Federal University of Rio Grande do Norte
Department of Informatics and Applied Mathematics (DIMAp)

Doctoral Proposal

Ernesto C. B. de Matos

ENTITLED

BETA: a B Based Testing Approach

Advisor Dr. Anamaria Martins Moreira

Co-Advisor Dr. Michael Leuschel

Natal July 16, 2015

Resumo

Sistemas de software estão presentes em grande parte das nossas vidas atualmente e, mais do que nunca, eles requerem um alto nível de segurança e confiabilidade. No mundo do desenvolvimento de software existem várias técnicas de Verificação e Validação (V&V) que se preocupam com controle de qualidade, segurança, robustez e confiabilidade; as mais conhecidas são Testes de Software e Métodos Formais. Métodos formais e testes são técnicas que podem se complementar. Enquanto métodos formais provêem mecanismos confiáveis para raciocinar sobre o sistema em um nível mais abstrato, técnicas de teste ainda são necessárias para uma validação mais profunda e são frenquentemente requeridas por orgãos de certificação. Nesta proposta de tese de doutorado nós propomos uma abordagem suportada por ferramenta para gerar casos de teste a partir de máquinas abstratas do Método B. O Método B é um método formal que usa conceitos de Lógica de Primeira Ordem, Teoria dos Conjuntos e Aritmética Inteira para especificar máquinas de estado abstratas que modelam o comportamento de componentes de um sistema. A abordagem proposta utiliza propriedades especificadas nestes modelos e aplica técnicas de teste bem fundamentadas pela comunidade de testes, como particionamento do espaço de entrada e critérios de cobertura lógica, para gerar casos de teste para a implementação do sistema. Os casos de teste gerados são testes de unidade que verificam se o código desenvolvido implementa o comportamento especificado nos modelos abstratos. Os requisitos para cada caso de teste são definidos usando fórmulas lógicas que são passadas como entrada para um solucionador de restrições para obter dados de teste. A ferramenta também provê funcionalidades para o cálculo de preâmbulos, verificação de oráculos e concretização dos dados de teste. Ela é capaz de gerar especificações de casos de teste e *scripts* de teste executáveis escritos em Java e C. A abordagem foi avaliada através de vários estudos de caso – utilizando modelos simples e complexos – e mostrou resultados promissores, sendo capaz de identificar defeitos inseridos durante o processo de refinamento e codificação, e defeitos criados por geradores de código defeituosos.

Palavras-chave: Métodos Formais; Teste de Software; Método B; Testes Baseados em Modelos.

Abstract

Software systems are a big part of our lives and, more than ever, they require a high level of security and reliability. In the software development world, there are many Verification and Validation (V&V) techniques that are concerned with quality control, security, robustness, and reliability; the most widely known are Software Testing and Formal Methods. Formal methods and testing are techniques that can complement each other. While formal methods provide sound mechanisms to reason about the system at a more abstract level, testing techniques are still necessary for a more in-depth validation of the system and are often required by certification standards. In this thesis proposal, we propose a tool-supported approach to generate test cases from B-Method abstract machines. The B-Method is a formal method that uses concepts of First Order Logic, Set Theory, and Integer Arithmetic to specify abstract state machines that model the behavior of system components. The proposed approach uses the properties specified in these models and applies well-established software testing techniques, such as input space partitioning and logical coverage criteria, to generate test cases for the system's implementation. The generated test cases are unit tests that verify if the system's code implements the behaviour specified in the abstract models. The requirements for each test case are expressed using logical formulas that are provided as input for a constraint solver to obtain test data. The tool also provides features for preamble calculation, oracle evaluation, and test data concretization. It is capable of generating test case specifications and executable test scripts in Java and C. The approach has been evaluated through several case studies – from simple to more complex models – and has shown promising results, being able to identify bugs inserted during the refinement and coding process; and bugs created by faulty code generators.

Keywords: Formal Methods; Software Testing; B-Method; Model-Based Testing.

Contents

1 Introduction **9**

1.1 Motivation . . . 10

1.2 Hypothesis and Research Questions . . . 13

1.3 Methodology . . . 14

1.4 Proposal Organization . . . 15

2 The B-Method **17**

2.1 Machines, Refinements and Implementations . . . 19

2.2 Generalized Substitutions . . . 19

2.2.1 Simple Attribution Substitution . . . 20

2.2.2 Multiple Attribution Substitution . . . 20

2.2.3 Parallel Substitution . . . 20

2.2.4 Conditional Substitution . . . 20

2.2.5 ANY Substitution . . . 21

2.3 B Notation Syntax . . . 21

2.4 Abstract Machines . . . 22

2.5 Proof Obligations . . . 26

2.5.1 Invariant Consistency . . . 26

2.5.2 Initialization Proof Obligation . . . 27

2.5.3 Operations Proof Obligation . . . 27

2.6 Tool Support . . . 28

2.6.1 AtelierB . . . 29

2.6.2 ProB . . . 29

2.6.3 Rodin . . . 29

3 Software Testing **31**

3.1 Testing Levels . . . 32

3.2 Functional and Structural Testing . . . 34

3.3 Basic Terminology . . . 35

3.3.1 Fault, Error and Failure . . . 35

3.3.2 Testing, Test Failure and Debugging . . . 35

3.3.3 Test Cases . . . 36
3.3.4 Coverage Criteria . . . 38
3.4 Input Space Partitioning . . . 39
3.4.1 Input Domain Model . . . 41
3.4.2 Criteria for block combination . . . 42
3.5 Logic Coverage . . . 43
3.5.1 Logic Expression Coverage Criteria . . . 45
3.5.2 Comments on MC/DC Coverage . . . 47

4 Related Work **49**
4.1 Current state of the art . . . 49
4.1.1 Amla and Ammann . . . 50
4.1.2 Dick and Faivre . . . 50
4.1.3 Marinov and Khurshid . . . 51
4.1.4 Ambert et al. . . . 52
4.1.5 Cheon and Leavens . . . 53
4.1.6 Satpathy et al. . . . 54
4.1.7 Gupta et al. . . . 55
4.1.8 Dinca et al. . . . 56
4.1.9 Cristiá et al. . . . 57
4.2 Final Discussions . . . 58

5 A B Based Testing Approach **60**
5.1 The Approach . . . 60
5.1.1 B Abstract Machine . . . 61
5.1.2 Logical Coverage . . . 61
5.1.3 Input Space Partitioning . . . 64
5.1.4 Use constraint solving to obtain test inputs . . . 67
5.1.5 Obtain oracle values . . . 68
5.1.6 Generate Test Case specifications . . . 69
5.1.7 Generate Test Scripts . . . 70
5.2 Comments on the evolution of BETA . . . 70

6 Recent contributions and improvements **72**
6.1 Support to Logical Coverage Criteria . . . 72
6.2 Automatic Oracle Evaluation and Oracle Strategies . . . 73
6.3 Preamble Calculation . . . 75
6.4 Generation of Test Scripts and Oracle Strategies . . . 76
6.5 Test Data Concretization . . . 77

7 Case Studies **78**
7.1 Previous Case Studies 78
7.1.1 The GDC Case Study 78
7.1.2 The FreeRTOS Case Study 79
7.2 Generating tests for the Lua API 80
7.2.1 Results 81
7.2.2 Final Remarks and Conclusions 83
7.3 Testing two code generation tools: C4B and b2llvm 86
7.3.1 Results 88
7.3.2 Final Remarks and Conclusions 89

8 Final Discussions and Next Activities **91**
8.1 Previous Activities 92
8.2 Next Activities 93

List of Figures

1.1 The B-Method development process 11

2.1 B-Method Development Process. 17
2.2 Train and metro lines around the world that employed the B-Method. 18

3.1 Usual software testing workflow. 32
3.2 Levels of Testing (the V model). 33

5.1 BETA approach overview 61

7.1 Testing Strategy for C4B and b2llvm. 87

8.1 Schedule for the next activities. 95

List of Tables

2.1 B Notation: Functions and Relations 21
2.2 B Notation: Logical Operators 22
2.3 B Notation: Set Operators 22
2.4 B Notation: Arithmetic Operators 23

4.1 Related Work Overview 59

7.1 Code coverage for the functions in the Lua API. 84
7.2 Branch coverage for the functions in the Lua API. 85
7.3 Machines used in the case study. 88

Chapter 1

Introduction

Software systems are a big part of our lives. They reside in all of our electronic devices; not only in our computers and smartphones but also in simple things, like our coffee machine, or in more complex ones, like the metro that we take every day to go to work.

There are many methods and processes to develop such systems. They usually involve activities like brainstorming requirements, modeling the system architecture, implementing the required code, deploying the system to be executed on hardware, etc. Among all these activities, there is a very important one that is usually called Verification and Validation (V&V). The main goal of V&V is to evaluate the quality of the system under development and answer questions like "Is the system correct according to functional requirements?", "Is the system safe enough?", "Can it recover from failures?", and others.

The V&V process is known to consume a great part of the resources involved during the development of software systems. According to [Myers, 2011], almost 50% of the time and money required to develop a system is spent on V&V.

The task of ensuring that a system is safe, robust and error-free is a difficult one, especially for safety-critical systems that have to comply with several security standards. There are many methods and techniques that can help with software V&V. The most widely known are *Software Testing* techniques which try to evaluate the reliability of a system by means of test cases that can reveal unknown faults. Another practice that is especially common in the development of safety-critical systems is the use of *Formal Methods*. In general, formal methods are based on models that serve as a representation of the systems behavior. These models are specified using logical and mathematical theories that can be used to prove their correctness.

Formal methods and testing are V&V techniques that can complement each other. While formal methods provide sound mechanisms to reason about the system at a more abstract level, testing techniques are still necessary for a more in-depth validation of the system and are often required by certification standards [Rushby, 2008]. For these reasons, there is an effort from both formal and testing communities to integrate these disciplines.

The B-Method [Abrial, 1996] is a formal method that uses concepts of *First Order Logic*, *Set Theory* and *Integer Arithmetic* to specify abstract state machines that represent the behavior of system components. These specifications can be verified using proofs that ensure their consistency. The method also provides a refinement mechanism in which machines go through a series of refinement steps until they reach an algorithmic level that can be automatically translated into code.

In [Matos, 2012], we proposed a tool-supported approach to generate test cases from B-Method abstract machines. The approach uses the properties specified in these models and applies well-established software testing techniques to enumerate test requirements for the system implementation. In this thesis proposal we improve upon the work developed in [Matos, 2012], improving the proposed testing strategy, adding new features, and performing new case studies to evaluate both the approach and the tool. The approach now refines the test requirements to complete, executable test cases containing test data, preamble calculation, oracle evaluation and test data concretization.

1.1 Motivation

The B-Method development process is presented in Figure 1.1. Beginning with an informal set of requirements, which is usually written using natural language, an abstract model is created. The B-Method's initial abstract model is called *Machine*. A machine can be refined into one or more *Refinement* modules. A Refinement is derived from a Machine or another Refinement and the conformance between the two modules must be proved. Finally, from a Refinement an *Implementation* module can be obtained. The Implementation uses an algorithmic representation of the initial model called B0. The B0 representation serves as basis for translation of the model to programming language code, which can be done either manually or by code generation tools.

As seen on the image, all steps between the abstract machine and the refinement to B0 implementation are verified using static checking and proof obligations. Nevertheless, even with all the formal verification and proofs, the B-Method alone is not enough to ensure that a system is error-free. In [Waeselynck and Boulanger, 1995] the authors present some limitations of the B-Method – and formal methods in general – that should encourage engineers to perform some level of software testing during system development. Some of these limitations are:

- Non-functional requirements are not addressed by formal methods;
- There are some intrinsic problems related to the activity of modeling. First, the model is necessarily an abstraction (and simplification) of the reality and has a limited scope. For example, the formal description takes into account neither the compiler used nor the operating system and the hardware on which it is executed. It also makes assumptions about the behavior of any component interacting with the software;

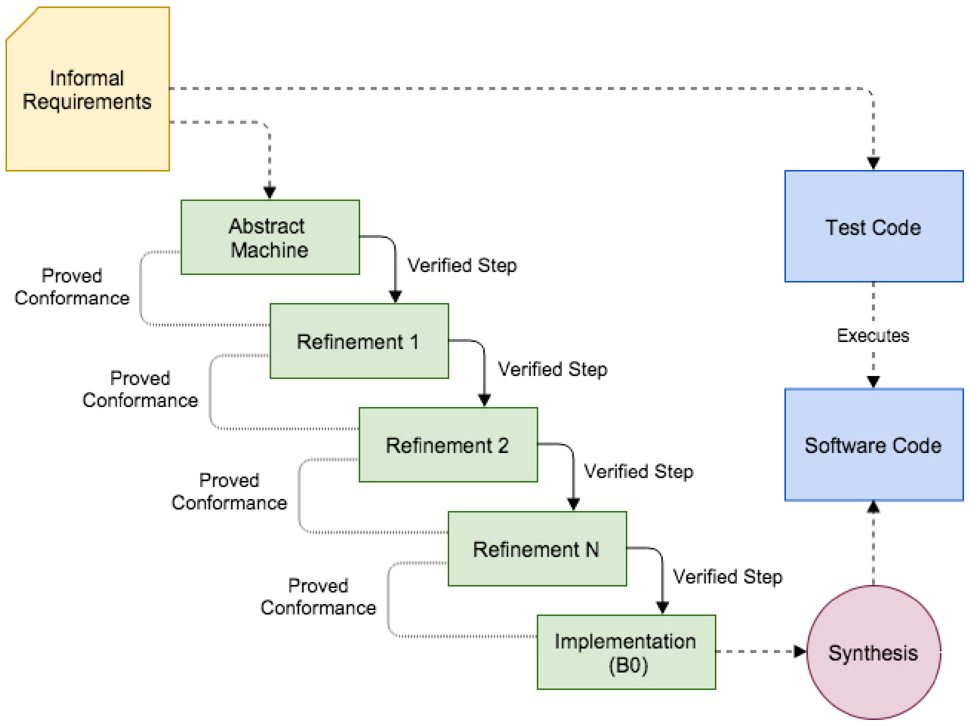


Figure 1.1: The B-Method development process and the usual testing procedure. Full line arrows represent formally verified steps and dashed line arrows represent steps that are not formally verified. Dotted lines show relationships between artifacts.

- The model can be wrong in respect to its informal requirements. In essence, there is no way to prove that the informal, possibly ill-defined, user needs are correctly addressed by the formal specification;
- Modeling is performed by a human that is liable to fail;
- Refinement of an abstract specification will always require a certain degree of human input, admitting possibilities of human errors;
- The formal system underlying the method may be wrong (the problem of validation of the validation mechanism);
- Ultimately, proofs may be faulty.

Besides the limitations pointed by Waeselynck and Boulanger, there are other aspects that could benefit from the use of software testing as a complement to the formal development process. These aspect are:

- The generation of tests from formal specifications can be particularly useful in scenarios where formal methods are not strictly followed. Sometimes, due to time and budget restrictions, formal methods are only used at the beginning of the development process – just for modeling purposes – and the implementation of the system is done in an informal way. In this scenario, tests generated from formal specifications could help to verify the coherence between specification and implementation, checking whether the implementation is in accordance with the specification or not.

– There is another problem related to the maintenance of the code base. Software code is always changing due to new or updated requirements. When the system needs to be updated the changes may be made directly into the source code. In this case, the effort to validate the model and its refinements is, at least, partially lost. Regression tests are essential to tackle this issue. They work as a safety net for programmers when they make modifications in the source code.

– It is also important to notice that the translation of B0 representation to code lacks formal verification. If done manually, the translation is obviously informal. The code generation tools for the B-Method are also not formally verified. So, in the end, the translation to source code cannot be entirely trusted. The code generated still need to be tested.

Given these limitations, software testing can complement a formal method like the B-Method, providing mechanisms to identify failures, exploiting possible defects introduced during refinement and implementation of the model, or during the maintenance of the code base.

As formal specifications usually describe requirements in a rigorous and unambiguous way, they can be used as basis to create good test cases. The automatic generation of test cases from formal models, using Model-Based Testing techniques, can also reduce testing costs.

Different research groups have then been researching the integration of formal methods and software testing in different ways. In the current literature, there are many publications targeting different types of tests, using different formal input models, and with different levels of automation ([Ambert et al., 2002, Satpathy et al., 2007, Gupta and Bhatia, 2010, Singh et al., 1997, Huaikou and Ling, 2000, Mendes et al., 2010, Burton and York, 2000, Marinov and Khurshid, 2001, Cheon and Leavens, 2002, Amla and Ammann, 1992, Dick and Faivre, 1993]).

Most of the current work in the field, we believe, have a deficiency in one or more of the following points:

– Their tests focus on finding problems in the model rather than on the respective implementation;

– They use *ad hoc* testing strategies instead of relying on well-established criteria from the software testing community;

– They lack on automation and tool support, something essential for the applicability of the proposed approaches.

In this thesis, we propose a Model-Based Testing approach to complement the B-Method development process. This approach is tool supported and partially automates the generation of test cases for a software implementation based on B-Method's abstract state machines. The test cases generated by this approach try to verify the conformance between the initial abstract model and the produced source code, checking if the behavior specified in the model is actually present in the software implementation. The tests generated are unit tests that test each operation in the model individually. They are generated using classic testing techniques that include input space partitioning and logical coverage.

1.2 Hypothesis and Research Questions

Given the proposal of this thesis, the following hypothesis need to be investigated:

1. Model-Based Testing can be used to complement the B-Method development process, providing a way to reduce the problem caused by the lack of formality in the code generation step of the process.

2. Well-established coverage criteria from the software testing community can be used to generate tests from B-Method models.

3. The entire process of the proposed approach can be automated by a tool.

To evaluate these hypothesis, we need to answer the following research questions:

Research Question 1: *How Model-Based Testing techniques can be used to complement the B-Method development process?*

As mentioned in the previous sections, our objective with this thesis is to propose a tool supported Model-Based Testing approach to complement the B-Method development process. The tests generated using this approach should help to validate the implementation of the system, checking if it is consistent with the abstract models. To answer this first question, we researched how software testing techniques could contribute to the B-Method development process. A review of the current literature in the field was performed to identify possible points of improvement in the current state of the art on the topic of Model-Based Testing using formal models. Once this points of improvement were identified, we proposed solutions for these problems and evaluated the proposed solutions through case studies.

Research Question 2: *What are the requirements for a proper Model-Based Testing approach for the B-Method?*

To generate test cases accordingly, there are some problems that need to be dealt with. For example, once test cases are defined, it is necessary to generate input data for them. Also, to properly compare the implementation and the model it is necessary to compare the outputs produced by both of them for a given set of test cases.

In some situations, it is also necessary to calculate preambles for the test cases. The preamble consists of a sequence of statements that will put the system in the state that we want to perform the test. So a mechanism for doing this preamble calculation is also welcome.

Ultimately, once the test cases are constructed, it is necessary to concretize them so they can be executed against the implementation code. Since the test cases are generated from an abstract model, that uses abstract data types, it is necessary to translate the test case values to concrete data types used in the implementation. This is a common problem in Model-Based Testing and has to be solved for tests generated using the B-Method as well.

Research Question 3: *How can we use well-established testing techniques to generate model-based tests in the context of the B-Method?*

There are many well-defined and well-established criteria to create test cases on the current software testing literature. In spite of that, most of the work on model-based testing using formal models use *ad hoc* testing criteria. In our approach we want to use testing criteria that was evaluated through time by the testing community so we can be more confident about the quality of our test cases. We also want our approach to provide some level of flexibility to allow new criteria to be introduced if needed.

1.3 Methodology

The chosen methodology to answer the questions from the previous section was to evaluate the approach and the tool through several case studies.

The first question we had to answer was how to use MBT to generate test cases that complement the B-Method development process. An initial version of the approach to generate the test cases was then designed [Souza, 2009] and evaluated by a case study [Matos et al., 2010]. The goal of this first evaluation was to assess the quality of the initial approach and determine the improvements that were necessary. The process of test case generation and evaluation of the test cases was performed completely by hand. After this first evaluation, the top priority of the project was defined: we needed a tool to automate the test case generation process. Performing the steps to generate the test cases by hand was a process very susceptible to errors and took a considerable amount of time to accomplish.

The focus now was the development of the first version of the tool [Matos, 2012]. Once we had an initial prototype, we evaluated it using a second case study [Matos and Moreira, 2012]. The target now was a model that was little more complex than the model used in the first case study. This second case study revealed some problems in the approach and bugs in the tool. It also helped to identify interesting features that could improve the tool, such as the generation of executable test cases and automatic oracle evaluation for the test cases.

Once the improvements were implemented in the initial version of the tool [Matos and Moreira, 2013], we took some steps towards more complex models. In the third case study, a model of the Lua programming language API was used to assess the second version of the tool [Souza Neto and Moreira, 2014]. But the goal now was not only to assess the tool but also to perform some analysis of the quality of the generated test cases. The models of the Lua API were much more complex and challenging for the tool. Once the test cases were generated, they were subjected to a code coverage analysis. This case study was also a good example of a case where model and source code were developed separately, ignoring the B-Method's formal process. The model was created based on the documentation of the API, and the tests were used to check the conformance of the code to the model (and vice-versa). In this case study, we also evaluated new features of the tool such as the test script generator.

To make a more in-depth study of the capabilities of the tool to detect discrepancies between the models and their respective (automatically generated) source code, we performed a case study with two code generators. In this case study, we used the test cases generated by BETA to check if the behavior of the code generated for several models was in accordance with their respective abstract models. This case study was also helpful to show the problems related to the generation of source code by non-verified code generators.

The next step proposed by this thesis is to perform another case study; this time focusing on the evaluation of the scalability of the tool and the approach. We plan to organize a case study that analyzes the performance of the BETA tool. It will measure aspects like the time needed to generate the test cases, the number of test cases generated, the number of test cases lost due to infeasibility, among other aspects, for each criterion implemented in the tool. This last case study will be performed using a large set of input models, that will vary in size and complexity.

1.4 Proposal Organization

The remainder of this thesis proposal is organized as follows:

Chapter 2: The B-Method This chapter gives a brief introduction to the B-Method, presenting all the elements necessary to understand the proposal;

Chapter 3: Software Testing This chapter presents the software testing terminology used in this document and all the testing criteria and techniques used in our testing approach;

Chapter 4: Related Work This chapter presents a review of the current work on model-based testing using formal models;

Chapter 5: A B Based Testing Approach This chapter presents an overview of our test case generation approach;

Chapter 6: Recent contributions and improvements This chapter presents our most recent contributions to the test case generation approach;

Chapter 7: Case Studies This chapter presents all the case studies performed until now;

Chapter 8: Next Activities and Schedule This chapter enumerates the next activities planned for the thesis and how they are scheduled.

Chapter 2

The B-Method

The B-Method is a formal method that can be used to model and develop safety-critical systems in a secure and robust way. It uses concepts of first order logic, set theory and integer arithmetic to specify abstract state machines that model the behaviour of the system. The method is also strongly characterized by the idea of model refinements.

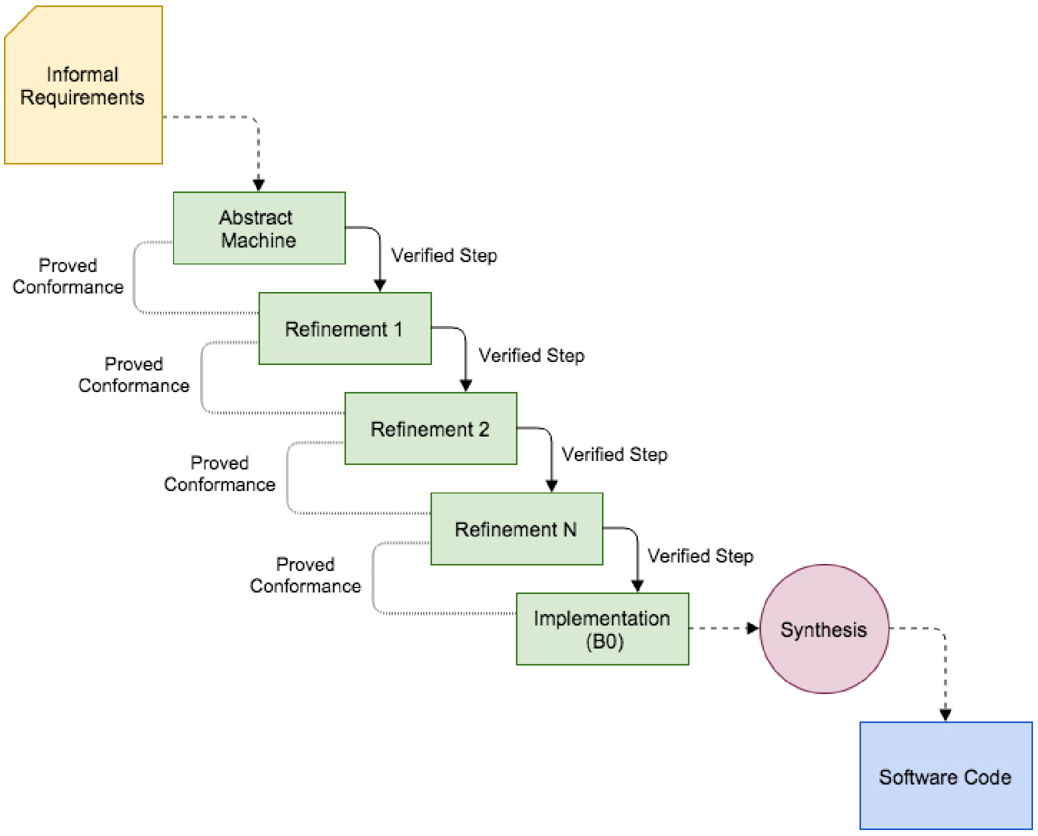


Figure 2.1: B-Method Development Process

An overview of the B-Method development process is presented in Figure 2.1. The process starts with an abstract *machine* specification, followed by incremental *refinements* until it reaches an *implementation*. In the implementation level, only imperative-like constructs may be used, this constructs are a subset of the B notation called B0. Such implementation is then translated to source code in a programming language, either manually or by code generation tools. More details about the machine, refinement and implementation modules are presented in the next section.

The consistency of the different modules is certified by proving that automatically generated verification conditions (called *proof obligations*) are valid. The refinement process is also subject to analysis through proof obligations. The proof obligations for the refinement process certify that a refinement is equivalent to its abstraction.

The B-Method is taught in many universities in formal method courses and has been successfully adopted in several industry projects. Its success in the industry could be mainly attributed to good tool support for both writing models and proving their correctness. Most of the projects where it has been used in industry are related to rail transport. Figure 2.2, which was adapted from Clearsy's[1] website, presents some of the trains and metro lines around the world that have employed the B-Method during their development.

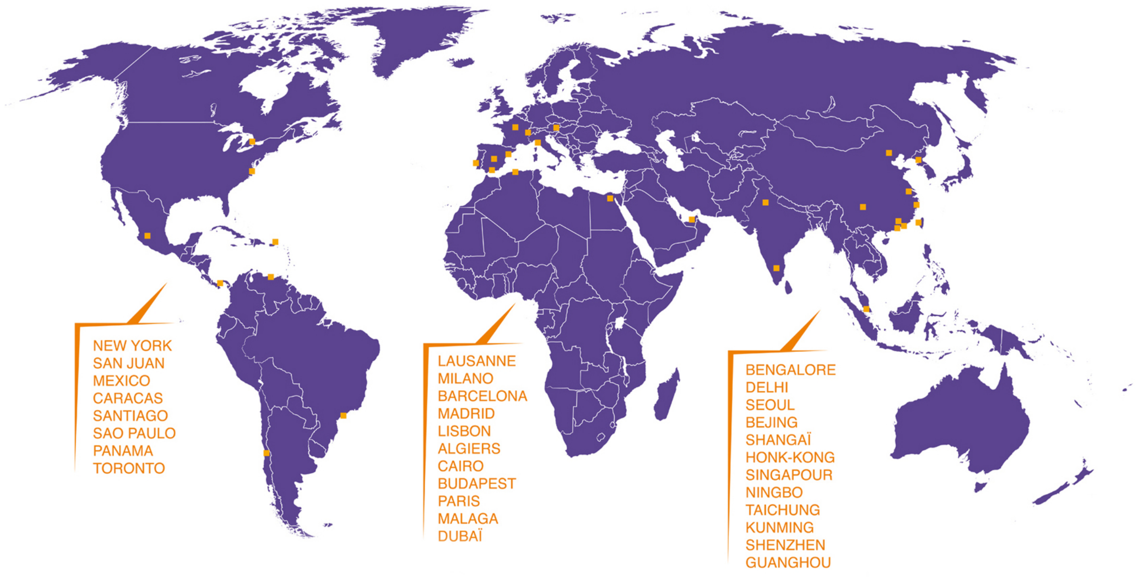


Figure 2.2: Train and metro lines around the world that employed the B-Method. Source: http://www.methode-b.com/en/b-method/

It is also important to mention that there is a variation of the B-Method called Event-B [Abrial, 2010]. The Event-B formal method might be considered as an evolution of the B notation, or *Classical B* as some might call the original B-Method notation. The Event-B notation is simpler and more minimal which makes it easier to learn and use. Its focus is mainly system-level modeling and analysis. Event-B development is supported by the

[1]Clearsy is a french company that is known for successful use of the B-Method in the development of safety-critical systems. It also maintains B-Method tools such as AtelierB. More information on their website: http://www.clearsy.com/en/

Rodin[2] tool.

2.1 Machines, Refinements and Implementations

B models are specified and structured in modules. The B-Method has a single notation encompassing abstract constructs suitable for specification and classic imperative constructs for computer programming. The B notation supports three different types of modules that are classified according to their level of abstraction. From the most abstract to the most concrete, they are *machine*, *refinement* and *implementation*.

A machine is an abstract state machine that has a set of *variables* that represent its state and a set of *operations* that can modify its state. Constraints on the state of the machine can be described in the machine's *invariant*. The method also has a *precondition* mechanism for the machine's operations. To ensure that an operation behaves as expected its precondition has to be respected.

A machine can also import other machines using one of the B-method modularization mechanisms. A machine can either *use, see, import, extend* or *include* another machine. These different types of imports provide different visibility of parts of the imported module.

A refinement is a module that refines a machine or another refinement. The idea of using this type of module is to refine the abstract model to something that is closer to the actual implementation of the system. A refinement module must always refer to a machine or another refinement. The refinement process can use one or more refinement modules. Each refinement step must be verified to guarantee the equivalence between the refinement and the module it refers to.

After one or more refinement steps, the specification can be refined to an implementation module. The implementation is the most concrete level of specification. It uses only a subset of the B notation called B0. The B0 notation consists of more concrete constructs that are similar to the ones available in most programming languages, so it makes the implementation more suitable for translation to a programming language.

2.2 Generalized Substitutions

Another concept used by the B-Method is the *Generalized Substitution*. Generalized substitutions are used to replace free variables in a predicate by expressions so the predicate can be analyzed and verified in a particular context.

The substitutions are written using the notation $[E/x]P$. This expression states that a variable x in the predicate P will be replaced by the expression E. For example, a substitution $[4/x](x+1>y)$ results in the predicate $4+1>y$.

[2]Rodin project website: http://www.event-b.org/

In this section only some of the B-Method substitutions are presented. The full list of substitutions can be found on [Clearsy, 2011].

2.2.1 Simple Attribution Substitution

$$[x := E]P \Rightarrow [E/x]P \tag{2.1}$$

When this substitution is applied, all free occurrences of x in P will be replaced by the expression E. In the machine context a predicate could be, for example, an invariant clause. This substitution is used to attribute values to state variables and return variables.

2.2.2 Multiple Attribution Substitution

$$[x := E, y := F]P \Rightarrow [E, F/x, y]P \tag{2.2}$$

The multiple attribution is a variation of the simple attribution. It works in the same way, and it is used to make attributions to two or more free variables simultaneously.

2.2.3 Parallel Substitution

$$S1 \parallel S2 \tag{2.3}$$

A very frequently used substitution in the B-Method is the parallel substitution. This substitution is used to represent two substitutions that, in theory, should occur exactly at the same time.

2.2.4 Conditional Substitution

$$[IF\ \ E\ \ THEN\ \ S\ \ ELSE\ \ T\ \ END]P \Rightarrow (E \Rightarrow [S]P) \wedge (\neg E \Rightarrow [T]P) \tag{2.4}$$

The conditional substitutions provide a mechanism to model behaviors that are based on a condition. In other words, given a logical expression, depending on what its outcome is, a particular substitution will be applied. In the notation presented above, if the expression E evaluates to $true$, the substitution S will be applied, if E evaluates to $false$, the substitution T will be applied instead.

2.2.5 ANY Substitution

$$[ANY\ X\ WHERE\ P\ THEN\ S\ END]R \Leftrightarrow \forall x.(P \Rightarrow [S]R) \tag{2.5}$$

This substitution allows the use of data declared in X, that is in accordance with the predicate P, to be used in the substitution S. If there is more than one set of values for the variables in X that satisfy P, then the substitution does not specify which set of values will be chosen. The ANY substitution is an example of non-deterministic substitution.

2.3 B Notation Syntax

In this section part of the B notation syntax is presented. Tables 2.1, 2.2, 2.3 and 2.4 present the syntax for functions, relations, logical operators, set operators and arithmetical operators.

Table 2.1: B Notation: Functions and Relations

Symbol	Description
$+->$	partial function
$-->$	total function
$-->>$	surjective function
$>+>$	partial injection
$>->$	total injection
$>+>>$	partial bijection
$>->>$	total bijection
$<->$	relation
$\|->$	mapping

Table 2.2: B Notation: Logical Operators

Symbol	Description
&	and
or	or
#	exists
!	for all
$=$	equal
$/=$	not equal
=>	logical implication
<=>	equivalence
$not(P)$	negation

Table 2.3: B Notation: Set Operators

Symbol	Description
:	belongs
/ :	does not belong
<:	included (subset of)
/ <:	not included (not subset of)
/\	intersection
\/	union
{}	empty set
POW	power set

2.4 Abstract Machines

In the B-Method, an abstract machine specifies a system or one of its modules. It models the state of the system and also operations that can act on this state. An abstract machine has many other elements besides variables and operations; the example presented on Listing 2.1 will be used to present some of this elements. The example is a model of a system that controls the students in a course and their grades. The features of the system are:

- It allows students to be registered in the course;
- It allows to register grades for the students registered in the course;
- It allows to register if a particular student was present in a practical lab class or not;
- Based on the grade of the student and its presence in the class the system can state the final result for the student in the course. A student can either pass, fail or be required to do a final test.

Table 2.4: B Notation: Arithmetic Operators

Symbol	Description
$+$	addition
$-$	subtraction
$*$	multiplication
$/$	division
$<$	less than
$<=$	less than or equal to
$>$	greater than
$>=$	greater than or equal to
INT	set of integers
NAT	set of natural numbers
$NAT1$	set of positive natural numbers

Listing 2.1: Example of B-Method machine

```
MACHINE Course

SETS
 STUDENT; result = {pass, final_exam, fail}

PROPERTIES
 card(STUDENT) = 15

VARIABLES
 students,
 grades,
 has_taken_lab_classes

INVARIANT
 students <: STUDENT &
 grades : students +-> 0..5 &
 has_taken_lab_classes : students +-> BOOL

INITIALISATION
 grades := {} ||
 students := {} ||
 has_taken_lab_classes := {}

OPERATIONS
  add_student(student) =
    PRE
      student : STUDENT
```

```
28      THEN
29        students := students \/ {student}
30      END;
31
32    add_grade(student, grade) =
33      PRE
34        student : students &
35        grade : 0..5
36      THEN
37        grades(student) := grade
38      END;
39
40    present_on_lab_classes(student, present) =
41      PRE
42        student : students &
43        present : BOOL
44      THEN
45        has_taken_lab_classes(student) := present
46      END;
47
48    rr <-- student_pass_or_fail(student) =
49      PRE
50        student : students &
51        student : dom(grades) &
52        student : dom(has_taken_lab_classes)
53      THEN
54        IF grades(student) > 3 & has_taken_lab_classes(student) =
             ↪TRUE THEN rr := pass
55        ELSIF grades(student) > 2 & has_taken_lab_classes(student) =
             ↪ TRUE THEN rr := final_exam
56        ELSE rr := fail
57        END
58      END
59  END
```

The name of a B machine is defined by the clause MACHINE in the specification. In the example presented the machine is named *Course* (line 1).

A B machine may also have a list of variables to represent its state. These variables are defined in the VARIABLES clause (lines 9-12). The *Course* machine has three variables: *students*, *grades* and *has_taken_lab_classes*. The variable *students* stores the set of students enrolled in the course, the variable *grades* stores their grades, and *has_taken_lab_classes* stores the information of whether or not they were present in the practical lab class.

An engineer can also define sets to be used in the model. These sets are specified in the machine's SETS clause (lines 3-5). Sets can be either *enumerated* or *deferred*. An enumerated set is a set which elements are stated as soon as it is declared. A deferred set is a set with unknown elements that can be specified with more details further in the modeling process. The *Course* example has two sets, an enumerated set *result* which elements represent the possible results for the student in the course, and also a deferred set *STUDENT* which is used to specify the students in the model. Using a deferred set to represent the students means that, in this level of abstraction, the engineer does not worry much about how a student will be represented in the system. In later refinements the student can be refined to be represented by a more concrete structure (e.g. integer identifier).

Constraints can be applied to sets using the PROPERTIES clause (lines 6-7). Properties can also be used to apply constraints to constants specified in the model. This particular example has no constants, but constants may be defined using the CONSTANTS clause.

The machine's variables may be constrained by predicates specified in the INVARIANT clause (lines 14-17). The machine's invariant is also used to define the types of its variables. The invariant must be satisfied in all of the machine's states so the model can be considered consistent. In the given example, the invariant defines that *student* is a subset of the deferred set STUDENT (line 15), that *grades* is a partial function that maps a student to an integer in the range between 0 and 5 (line 16), and that *has_taken_lab_classes* is also a partial function that maps students to a boolean value.

The initial state of the machine is defined by the INITIALISATION clause (lines 19-22). A machine can have one or more valid initial states. If a machine has variables, it must have an initialization. It is used to attribute the initial values for the state variables. In the given example, all the variables are initialized as *empty*.

A machine also needs a mechanism to modify its state. In the B-Method the only way to modify a machine's state is through operations. The machine's operations are listed in the OPERATIONS clause (lines 24-59). The header of an operation specifies its name and can also define its parameters and return variables. An operation can also have a precondition. A precondition can be used to define types for the operation's parameters and also other constraints that must be true so the operation can be executed correctly. The precondition acts as a contract for the operation. It can only guarantee that the operation behaves properly if the contract is respected. If the contract is broken, the method cannot foresee how the system will behave.

The *Course* example has four operations. The first three operations (*add_student*, *add_grade*, and *present_on_lab_classes*) are very similar. The *add_student* operation registers a student in the course, the *add_grade* operation register a grade for a given student, the *present_on_lab_classes* operation registers if a given student was present or not in the practical lab class. All of these three operations have parameters and preconditions. Their preconditions define types and other restrictions for the parameters, and they also act as guards for the operations. In their body they just make simple updates in one of the state variables.

The last operation is the most different one. It receives a student as a parameter and checks if he or she has passed or not in the course. The result is returned as an operation return variable (rr). The operation also has a conditional substitution on its body to check whether or not a student has passed the course. If a student's grade is greater than three and he or she has attended the lab class, then the operation returns 'pass'. If a student's grade is greater than two and less than or equal to three, and if he or she has attended the lab class, then the operation returns 'final_exam'. If a student does not fit in one of these cases than the operation returns 'fail'.

2.5 Proof Obligations

After the machine is specified, it has to be verified to certify coherence. In the B-Method, this verification is done using *Proof Obligations*. Proof obligations are logic expressions generated from the specification that should be validated to guarantee that the machine is coherent.

There are many types of proof obligations that can be generated from different elements of the model. Here, only some of the proof obligations are presented. The remainder of the proof obligations are found on [Abrial, 1996]. The proof obligations use the substitution notation presented on Section 2.2. The proof obligations presented in this section are: invariant consistency, initialization and operations proof obligations.

2.5.1 Invariant Consistency

The invariant consistency is proved by asserting that the machine has at least one valid state. It means that there should be at least one combination of values for the state variables that respect the machine's invariant. The formula used to do this verification is the following:

$$\exists v.I \tag{2.6}$$

Where v represents the machine's state variables and I represents its invariant.

To verify the invariant consistency of the *Course* machine one must prove the following:

$$\begin{align}
&\exists(students, grades, has\_taken\_lab\_classes). \tag{2.7}\\
&\quad(students \subseteq STUDENT \land \tag{2.8}\\
&\quad grades \in students \twoheadrightarrow 0..5 \land \tag{2.9}\\
&\quad has\_taken\_lab\_classes \in students \twoheadrightarrow BOOL) \tag{2.10}
\end{align}$$

If the free variables are instantiated respectively as $students = \{student1\}$, $grades = \{student1 \mapsto 5\}$ and $has\_taken\_lab\_classes = \{student1 \mapsto true\}$, one can prove that there is at least one state where the machine's invariant holds $true$.

2.5.2 Initialization Proof Obligation

The next proof obligation refers to the machine's initialization. This proof obligation is used to prove that the initial states of the machine satisfy the invariant. The formula for this proof obligation is the following:

$$[T]I \tag{2.11}$$

Where $[T]$ is the substitution made in the initialization, and I is the machine's invariant. For the *Course* example, one has to prove the following:

$$[students := \emptyset, grades := \emptyset, has\_taken\_lab\_classes := \emptyset] \tag{2.12}$$
$$(students \subseteq STUDENT \wedge \tag{2.13}$$
$$grades \in students \twoheadrightarrow 0..5 \wedge \tag{2.14}$$
$$has\_taken\_lab\_classes \in students \twoheadrightarrow BOOL) \tag{2.15}$$

After the substitution, the following predicate would need to be true:

$$\emptyset \subseteq STUDENT \wedge \emptyset \in students \twoheadrightarrow 0..5 \wedge \emptyset \in students \twoheadrightarrow BOOL \tag{2.16}$$

This predicate is actually $true$, hence the proof obligation for the initialization can be considered finished.

2.5.3 Operations Proof Obligation

The last type of proof obligation presented here refers to the machine's operations. The objective of this proof obligation is to prove that during the execution of a given operation, the machine's state changes to or remains at a valid state. The following formula defines this proof obligation:

$$I \wedge P \Rightarrow [S]I \tag{2.17}$$

Where I is the machine's invariant, P is the operation's precondition and S is the substitution used in the body of the operation.

This proof obligation states that, if the invariant and the precondition hold true before the execution of the operation, after the substitution S is performed, the invariant will still hold true (which means the machine will still be in a valid state).

For the operation *add_student* in the *Course* example the following proof obligation would be generated:

$$(students \subseteq STUDENT \wedge grades \in students \twoheadrightarrow 0..5 \wedge \tag{2.18}$$
$$has\_taken\_lab\_classes \in students \twoheadrightarrow BOOL) \wedge \tag{2.19}$$
$$(student \in STUDENT) \Rightarrow \tag{2.20}$$
$$[students := students \cup \{student\}] \tag{2.21}$$
$$(students \subseteq STUDENT \wedge grades \in students \twoheadrightarrow 0..5 \wedge \tag{2.22}$$
$$has\_taken\_lab\_classes \in students \twoheadrightarrow BOOL) \tag{2.23}$$

After the substitution, the following predicate would need to be true:

$$(students \subseteq STUDENT \wedge grades \in students \twoheadrightarrow 0..5 \wedge \tag{2.24}$$
$$has\_taken\_lab\_classes \in students \twoheadrightarrow BOOL) \wedge \tag{2.25}$$
$$(student \in STUDENT) \Rightarrow \tag{2.26}$$
$$(students \cup \{student\} \subseteq STUDENT \wedge grades \in students \twoheadrightarrow 0..5 \wedge \tag{2.27}$$
$$has\_taken\_lab\_classes \in students \twoheadrightarrow BOOL) \tag{2.28}$$

Since this implication is, in fact, true, the proof obligation for the operation is considered finished.

2.6 Tool Support

The B-Method is supported by some tools that make the process of specification and verification of the models easier. Among these tools, the most popular ones are *AtelierB*[3] and *ProB*[4].

[3]AtelierB's website: http://www.atelierb.eu/en/

[4]ProB's website: http://www.stups.uni-duesseldorf.de/ProB/index.php5/The_ProB_Animator_and_Model_Checker

2.6.1 AtelierB

AtelierB is a tool for specification and verification of B models. The tool is developed by Clearsy[5], a company that specialises in developing safety-critical systems using the B-Method. The tool has many features that can help the engineer through the various steps that are involved when developing using the B-Method, such as:

- an editor for the specification of modules using the B notation
- automatic generation of proof obligations
- automatic provers for the proof obligations
- an interactive prover for proofs that cannot be performed automatically.
- an assistant to help the refinement process

The tool also has some features to assist common practices in the software development world, such as, project management, distributed development and documentation.

AtelierB is free and has versions for Windows, Linux, and OS X. Also, some of its components are open source.

2.6.2 ProB

ProB is an animator and model checker for the B-Method. The tool allows models to be automatically checked for inconsistencies such as invariant violations, deadlocks, and others. It can also be used to write and animate models. Animations make it easier to experiment with the model while writing it and may help to find possible flaws in the specification earlier.

The tool also provides graphic visualizations for the models, has constraint solving capabilities, and also has some testing capabilities. The later will be discussed with further details on Chapter 4.

ProB also supports *Event-B* [Abrial, 2010], *Z* [Spivey, 1992], *TLA+* [Lamport, 2002] and *CSP-M* [Roscoe, 1997] formalisms. It is free, open source and has versions for Windows, Linux and OS X.

2.6.3 Rodin

Even tough our work does not focus on Event-B, we can not forget to mention the Rodin platform. Rodin is an Eclipse-based IDE for the development of Event-B projects. It provides effective support for specifying Event-B models and also supports refinements and mathematical proofs. Rodin is extendable with plugins and can be integrated with tools like ProB

[5]Clearsy's website: http://www.clearsy.com/en/

and AtelierB. The platform is free, open source and contributes to the Eclipse framework project.

Chapter 3

Software Testing

When developing any product, there is always a need to ensure the quality of what is being developed. Usually, there are activities in the development process that should be performed to assure and measure the quality of the product. These activities are necessary to ensure some level of safety and quality of the product for its users. The same concept is also present in software development. There are many techniques that can be used to assure and measure software quality. A formal method such as the B-Method presented in the last chapter is one example of a technique that can be used for this task. Another one is Software Testing. Software testing still is the primary method used by the software industry to evaluate software. The current literature on software testing has many techniques that can be used to test software. This chapter presents some of these techniques.

The basic idea behind software testing is to execute the software under test with the intent of finding errors Myers [2011]. Typically, this is done with the assistance of a set of test cases (also called test suite). A test case has the objective of executing a particular functionality of the software under test with a given test input. Then, the behavior of the software is observed, and the produced output is compared with the expected correct output to give the test case a verdict. If the software presents an unexpected behavior or produces an output that is different from the one expected, the test case fails. Usually, if a test case fails, it means that it found a problem in the software. A good test case has a high probability of finding hidden problems in the software.

The professional who performs testing activities is usually called a *Test Engineer*. Some of the test engineer responsibilities are: to document test requirements, design test cases, implement automated test scripts, execute test scripts, analyze test results and report them to developers and stakeholders.

Figure 3.1 presents the typical software testing workflow. The process usually begins with planning and designing test cases based on a list of test requirements. Then, the designed test cases are coded in a script or programming language so they can be executed automatically against the software implementation. The execution produces an output that is evaluated by an *oracle*. The oracle has the responsibility of saying if a test case passed or failed. After the execution of all test cases, a test report is produced to present the obtained results. These results provide feedback to the test engineer about the current state of the software, and can, hopefully, help the developers to find problems in the software that were previously unknown.

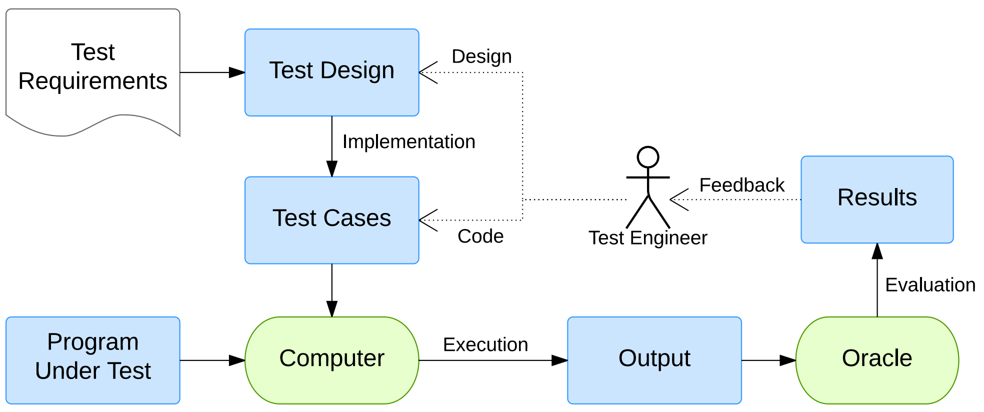


Figure 3.1: Usual software testing workflow.

3.1 Testing Levels

The software development process is composed by different phases, such as documenting requirements, planning and designing the software architecture and actual coding of the software. Faults can be introduced in any of these phases. That is why it is necessary to perform testing on different levels of the development process. This section presents the definition and the different levels of testing.

Each testing level is related to a different activity in the software development process. The relationship between the levels and the development activities are presented in Figure 3.2. In the current literature, there are different classifications for testing levels. The one presented here is presented in Ammann and Offutt [2010] and it classifies the levels in *acceptance*, *system*, *integration*, *module* and *unit*.

Acceptance testing is related to the requirements analysis phase of the software development process. This phase consists in identifying and documenting the needs of the users that should be attended by the software under development. Acceptance tests have the objective to verify if the software implemented is in accordance with the requirements. These tests can be performed either manually, by the final user interacting with the program or automatically, by tools that can simulate the user's actions.

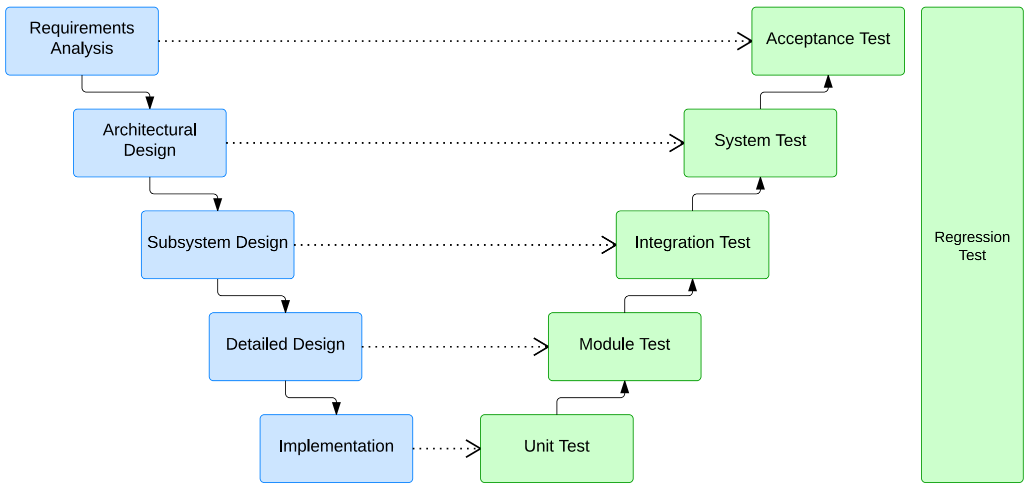


Figure 3.2: Levels of Testing (the V model).

System tests are related to the activity of architecture design. Typically, computer systems are composed by many software and hardware components that interact with each other and together provide a solution for the user's needs. The objective of system testing is to verify if the chosen composition attends the software's specification and to check for problems in the chosen architecture. System tests assume that each component was already tested individually and that they work properly.

The components used by the software architecture must communicate with one another to perform tasks. The communication between these components is done by their interfaces. These interfaces can send and receive information that will be processed by the component. The objective of integration tests is to verify if the interfaces can communicate properly.

Module testing is related to the activity of detailed design of the software. The objective of this activity is to determine the structure of the modules in the implementation. It is important to notice here the difference between *Modules* and *Units*. Modules are files that combine sets of functions and variables. Units are named instructions or procedures that can be called at some point of the program by its name. Taking as an example the Java programming language, classes are modules and methods are units. The objective of module testing is to analyze a module in an isolated way, verifying how its units interact with each other.

Unit testing is related to the most concrete level in the software development process: the actual implementation of units (methods, functions etc.). The objective of unit testing is to verify each unit isolatedly, without concerns about the rest of the scope that it belongs to. Unit tests are usually developed in parallel while coding the program, to ensure that the functions implemented behave as expected. In some bibliographies, module and unit tests are considered the same thing. In our work, we treat them differently.

Another constant activity present in the software development life cycle is the maintenance of the code. Code needs to be constantly corrected and updated and it is necessary to ensure that everything still works as it was working before the modifications. Regression tests should always be executed after some modification in the implementation is made and must ensure that the updated version has the same functionalities that it had before the update and that the software still works at least as good as it worked before the update.

3.2 Functional and Structural Testing

Two common terms used in software testing are Functional Testing (or black-box testing) and Structural Testing (or white-box testing). They represent the test's visibility of the system under test. For functional testing, the tests see the system as a black-box that receives an input and produces an output. There is no knowledge about the internals of the system. On the other hand, structural testing uses the knowledge about the system's internals to create test cases.

In structural testing, the internals of the software are available for the testing engineer. It means that all internal functions are available and can be explored and tested, either individually or integrated to other components.

One of the biggest advantages of structural testing is the possibility to create tests that can explore the internal structure of the code. For example, it would be easier to create tests to check all branches and execution paths in the program. It also makes it easier to see the coverage provided by a particular set of test cases. Usually, structural testing uses graphs or fluxograms to determine test requirements. Some examples of test requirements for this kind of test would be: "execute all paths in a graph" or "visit all nodes in a graph".

Functional tests are designed with no knowledge of the internal structure of the system. Typically, these tests are derived from external descriptions of the system, such as specifications, models or requirements. This type of test is usually performed when there is no access to the system's code. For example, in some cases the design and execution of the tests might be outsourced and performed by a different company. It might be the case that the developer doesn't want to provide the source code for the company that is performing the tests. In this cases, the test are designed looking at the system as a black-box, that can only be observed through inputs and outputs.

In most cases, structural and functional tests are not considered alternatives but are rather complementary. They complement each other and can find different types of problems.

3.3 Basic Terminology

This section introduces the basic software testing terminology that is going to be used throughout the remainder of this thesis.

3.3.1 Fault, Error and Failure

Some basic concepts that are also commonly mistaken are the definitions of *Fault*, *Error* and *Failure*.

Fault: a mistake made in the implementation that can result in problems during the execution. It is a static property in the software.

Error: an error is an incorrect state of the software that was caused by a fault.

Failure: is the external presentation of the fault when the error occurs. The manifestation of the error that is seen by the users.

Listing 3.1: Code example to illustrate the concepts of Fault, Error and Failure

```
int vector[] = new int[5];

for (int i = 0; i <= 5; i++) {
  System.out.println(vector[i]);
}
```

Let's consider the Java code presented in Listing 3.1. This code creates a vector of size 5 and iterates over it using a loop. As in the great majority of programming languages, the indexes of a vector in Java start with 0. So, the indexes for a vector of size 5 would range between 0 and 4. In the program presented, the vector is iterated from 0 to 5. That is the fault in the code. During the execution, the program will try to access the index 5, this will result in an error. Then, this error will be manifested as a failure for the programmer, in this case, the compiler will produce an *ArrayIndexOutOfBoundsException*.

3.3.2 Testing, Test Failure and Debugging

Once you know the difference between fault, error and failure, it is easier to understand the definition of *Testing* and *Test Failure*.

Testing: is the process of evaluating the software by observing its behavior during execution.

Test Failure: is an execution of a test on the software that results in failure.

The main objective of testing is to find faults in the software. The execution of a test case may result in a test failure. The test failure is then the starting point of a process called *debugging*. Debugging is the process of finding the fault in the software that caused a particular test failure.

3.3.3 Test Cases

The *Test Case* is another important concept behind software testing. It can be defined as follows:

Test Case: is a composition of *prefix values*, *test case values*, *expected results* and *postfix values* that are necessary for a complete execution and evaluation of one or more features of the software under test.

As can be seen, a test case is composed by many parts. The definition of each of these parts is presented next.

Test Case Values: are the input values necessary to complete some execution of the software under test.

It is important to mention here that these are not only inputs that are provided to a method or interface such as parameters or arguments. An input may also consist of a state that is needed to execute the test. It may even be the execution of a sequence of procedures that is required before you can execute the test case. When it is necessary to put the software in a particular state before it can be executed, the test case will need *Prefix Values*.

Prefix Values: are any inputs or actions necessary to put the software into the appropriate state to receive the test case values.

For a complete test case execution, it is also necessary to know what are the expected outputs for a particular set of test case values. These outputs are called *Expected Results*.

Expected Results: are the outputs that will be produced by a correct implementation of the software under test when executing a test case using a particular set of prefix values and test case values.

In some cases, it may also be necessary to provide some inputs to the software after the execution of the test cases. It can be done with *Postfix Values*.

Postfix Values: are any values that need to be sent to the software under test after the test case is executed.

Usually, postfix values are used when test cases need to be executed in sequence. For example, after the execution of each test case, the postfix values can be sent to the software under test so it can be put back into a stable state before the next test case is executed. Another example of usage of postfix value would be to send a command to terminate the program after the test case execution.

Test cases can be performed manually, but since it is a laborious, repetitive and error prone task in most cases, it is preferred to automate the test cases execution as much as possible. Typically, test case automation is done using *Executable Test Scripts*.

Executable Test Script: is a test case that is prepared to be automatically executed on the software under test and to produce a report about the test in the end.

There are many tools that can help in the process of writing executable test scripts, such as xUnit tools (e.g. JUnit[1]). The desirable scenario is one where the execution of a test case should be completely automated. This means, putting the software in the state necessary for the test, executing it with test case values, comparing the expected results with actual results produced by the software, sending postfix values if needed, and finally producing a report with test case results.

Finally, the last definitions about test cases that need to be explained are *positive* and *negative* test cases. These two definitions are based on the idea that there is a contract – that can be either implicit or explicit – about what is considered a valid input for the program (or units of the program). With this idea in mind, positive and negative test cases can be defined as follows:

Positive Test Case: is a test case that uses test values that are considered valid according to the program's input contract.

Negative Test Case: is a test case that uses test values that are considered invalid according to the program's input contract.

There are other definitions for positive and negative test cases in the software testing literature, but these are the ones we use in this thesis proposal. These definitions will be used

[1]JUnit project website: http://junit.org/

later in this document to speak about test cases that respect or disrespect the preconditions of the software under test.

3.3.4 Coverage Criteria

When testing a software it is important to ensure that it works for a wide range of scenarios. Ideally, it would be interesting to have test cases for all possible execution scenarios. Unfortunately, this is impossible in most situations since the number of necessary test cases would be too high (close to infinity in some cases). There are also computational limitations related to test cases generation and execution.

Since it is impossible to have test cases for every possible scenario, it is necessary create a good test set that has a high probability of finding faults with as few test cases as possible. That is when a good *Coverage Criterion* comes into place. A coverage criterion defines a set of rules for a test set based on test requirements. The definition of test requirement is the following:

Test Requirement: is a specific element of a software artifact that a test case must satisfy or cover.

Using the definition of test requirement, a coverage criterion can be defined as follows:

Coverage Criterion: is a collection of rules that impose test requirements on a test set.

If we make an analogy, a coverage criterion could be seen as a recipe to define test requirements for a test set. It should describe the test requirements in a complete and unambiguous way given some information on the software to be tested.

Let us use a simple example to explain these two definitions. Consider a simple coverage criterion which states "all methods of a class should be executed". Also, let us consider that the software under test consists of a single class with three methods *m1*, *m2* and *m3*. This criterion would yield three test requirements: "*m1* should be executed", "*m2* should be executed" and "*m3* should be executed".

A test set is said to satisfy a coverage criterion if, for each test requirement imposed by the criterion, there is at least one test case in the set of tests that satisfies it. Notice that the test engineer could write a single test that executes all three methods. It is ok to have one test covering more than one test requirement.

It is possible to calculate the level of coverage of a test set T by dividing the number of test requirements that it satisfies by the number of elements in the set of test requirements TR:

number of requirements covered by T / size of TR

There might be cases where some test requirements cannot be satisfied. It means that it is impossible to create a test case that covers a test requirement. When a test requirement cannot be satisfied, it is classified as an *infeasible* test requirement.

Infeasibility: a test requirement is considered to be infeasible when there is no combination of test case values that can satisfy it.

A coverage criterion can be used in two ways. It can be used directly to generate test case values that will satisfy the criterion. Or, it can be used after the creation of the test cases (using other mechanisms or even manually) to measure their quality and coverage. In the first way the coverage criterion is used as a *generator*, in the second way it is used as a *recognizer*.

Coverage criteria can also be related to one another. Some coverage criteria may yield test requirements that belong to a subset of a more in-depth coverage criterion. That is the idea of coverage criteria subsumption. A better definition would be the following:

Subsumption: a coverage criterion A subsumes a coverage criterion B if and only if every test set that satisfies criterion A also satisfies B.

These definitions will be used throughout the next sections of this chapter to present two categories of coverage criteria: *Input Space Partitioning* and *Logic Coverage*.

3.4 Input Space Partitioning

Fundamentally, software testing consists in choosing test case values from the software input space and then running the software with the chosen values to observe its behavior. Considering the importance of the chosen inputs to software testing there is a category of coverage criteria that focuses on selecting relevant test case values for a test set. This category is called *Input Space Partitioning*. Input Space Partitioning is a classic concept used in software testing. There are many definitions for this concept in the software testing literature. The concepts used here to describe Input Space Partitioning are based on the ones presented in [Ammann and Offutt, 2010], which explain it in terms of test requirements and coverage criteria.

The idea behind input space partitioning is that it is possible to divide the set that represents the universe of possible inputs for the software into subsets with inputs that are considered equivalent according to some test requirements. It means that if a subset contains inputs that are equivalent for the same test requirement, a test engineer could choose any input from this subset because they are all expected to produce the same behavior.

These subsets with equivalent inputs can also be called *equivalence classes*. From this point forward, we refer to them simply as *blocks*.

A *partition* defines how the universe of input values can be divided into blocks. The first thing that is necessary to define a partition is to identify the input parameters for the software under test. Input parameters can be function parameters, global variables, the current state of a program or even inputs provided by the user via GUI, depending on the kind of artifact that is being tested. The universe of all possible values that the input parameters can assume is called *input domain*.

The scope of the input domain for a particular input parameter can be defined by some restrictions. For example, if the input parameter is a boolean variable, its input domain is restricted to the values *true* and *false*. These restrictions are *characteristics* of the input parameter.

Input space partitioning consists in dividing the input domain of the software under test into blocks, taking into account the characteristics of the input parameters. Each characteristic can yield a set of blocks, and each one of the produced blocks represents a set of input data that is considered equivalent when testing the software in respect to the given characteristic.

To better understand how it is possible to use characteristics to partition the input space of a program, let us use an example to explain how the process works. Let us consider a program that controls the credits from an electronic card that is used to buy bus tickets. A card belongs to one of the three categories: *Standard*, *Student*, and *Free*. *Standard* cards are debited the normal price for a bus ticket; *Student* cards are debited half the price of a normal ticket; and *Free* cards aren't debited anything for a bus ticket.

If we were testing a function that debits the value of the ticket from one of these cards, the "category of card" would be an interesting characteristic to partition the input space of this function. This characteristic can be partitioned in three blocks:

- B1: Standard Cards;
- B2: Student Cards.
- B3: Free Cards.

Each one of this blocks represent a subset of the function's input domain and the elements in these subsets are considered equivalent when testing the function in respect to chosen characteristic. So, it would only be necessary to select one element from each one of these blocks to test this particular characteristic. Here we are using these tests just to explain the concept of blocks of equivalent data. To generate more meaningful tests one can use one of the coverage criteria presented later in this chapter.

3.4.1 Input Domain Model

The process of creating partitions for the software under test is called *Input Domain Modelling*. This process can be divided into the following three steps:

1. *Identifying testable functions:* first it is necessary to identify the functions that should be tested in the program. These functions can be methods in a class or use cases in a use UML use case diagram, for example;

2. *Identifying parameters that affect the behavior of the function:* after identifying the functions that should be tested it is necessary to identify for each one of them the variables that affect their behavior. These variables can be function parameters and/or global variables that are used by the function. They will compose the set of input parameters of the function under test;

3. *Modelling the input domain using characteristics and blocks:* the test engineer has to find characteristics that describe the input domain of the function under test. The characteristics describe the input domain of the function in an abstract way and are used by the test engineer to partition its input domain. Each partition is composed by a set of blocks. Then, test case values are selected from these blocks.

The blocks created using this process should comply with two properties:

1. The union of all the blocks in a partition must be equal to the input domain. It means that a partition must cover all the input domain of the function under test;

2. The blocks in a partition should not overlap. In other words, the intersection between the blocks must be empty. It means that one test case value cannot be used to cover more than one block of the same partition.

There are many strategies to partition the input domain of a testable function. Some of the most common strategies are:

– *Valid values:* include blocks with valid values according to the characteristics of the function;

– *Invalid values:* include blocks with invalid values according to the characteristics of the function;

– *Boundary values:* include values that are close to the boundaries in intervals for the variables in the input domain.

Once the input domain model is created, several coverage criteria are available to decide how to combine values from the blocks into test cases.

3.4.2 Criteria for block combination

After defining the input domain model and creating the partitions for each characteristic, it is necessary to define how the test case values will be selected from the blocks to be used on the test cases. To do it there are many coverage criteria available that can help the test engineer in this task. Three of these criteria are presented here: *All-combinations*, *Each-choice* and *Pairwise*.

Imagine a scenario where the test engineer created a input domain model with three partitions, two partitions with two blocks and one partition with three blocks, such as: [α, β], [1, 2, 3] and [x, y].

Combinatorial criteria can be used to combine these blocks into test requirements. The first idea that might cross the test engineer's mind is to test all possible combinations of blocks. That is exactly what the first criterion does:

All-combinations: All combinations of blocks from all characteristics must be used.

For the given partitions, if the test engineer wanted a set of tests that covered the All-combinations criterion, it would be necessary the following set test cases (TCs) which in this case are equal to the test requirements (TRs):

$$\begin{aligned} TRs = TCs = \{&(\alpha,\ 1,\ x),\ (\alpha,\ 1,\ y),\ (\alpha,\ 2,\ x),\ (\alpha,\ 2,\ y),\\ &(\alpha,\ 3,\ x),\ (\alpha,\ 3,\ y),\ (\beta,\ 1,\ x),\ (\beta,\ 1,\ y),\\ &(\beta,\ 2,\ x),\ (\beta,\ 2,\ y),\ (\beta,\ 3,\ x),\ (\beta,\ 3,\ y)\} \end{aligned}$$

The number of test cases necessary to cover All-combinations is equal to the product between the number of blocks of each partition (for given example it would be $2 \times 2 \times 3 = 12$). It is easy to see that if the number of partitions and blocks are too high the number of test cases necessary to cover this criterion grows considerable. That's why it is necessary to have combinatorial criteria that require fewer test cases but still produces interesting test requirements that have a good probability of finding faults. The second criterion presented here usually requires just a few test cases to satisfied:

Each-choice: One value from each block of each characteristic must be used in at least one test case.

For the given example, this criterion would yield the following test requirements:

$$TRs = \{\alpha, \beta, 1, 2, 2, y, x\}$$

It is possible to define different test sets that satisfy these requirements, for example:

$$TCs_1 = \{(\alpha, 1, x), (\beta, 2, y), (\alpha, 3, y)\}$$
$$TCs_2 = \{(\alpha, 1, x), (\beta, 2, y), (\beta, 3, x)\}$$

Both sets of tests satisfy the criterion and there are more test sets that could satisfy it as well. The minimum number of test cases required to satisfy this criterion is equal to the number of blocks of the partition that has the highest number of blocks. For the given example, it would be necessary at least three test cases to satisfy this criterion (the number of blocks in the second partition).

In most cases, this criterion is considered to be "weak" because it allows a lot of flexibility during the creation of test cases. It is considered too flexible because it does not require specific combinations of blocks from different partitions. The next criterion tries to fix this problem:

Pairwise: a value from each block from each characteristic must be combined with a value from every block for each other characteristic.

In other words, the pairwise criterion requires that all possible combinations two by two between blocks of two partitions should be tested. The number of tests generated using the pairwise criterion is usually higher than the number of tests generated using each-choice, and are significantly lower than all combinations for many scenarios. For the given example, this criterion yields the following test requirements:

$$\begin{aligned} TRs = \{&(\alpha, 1), (\alpha, 2), (\alpha, 3), (\alpha, x) \\ &(\alpha, y), (\beta, 1), (\beta, 2), (\beta, 3), \\ &(\beta, x), (\beta, y), (1, x), (1, y), \\ &(2, x), (2, y), (3, x), (3, y)\} \end{aligned}$$

The following tests can be used to cover all the pairs listed above (a "–" can be replaced by any block in the partition):

$$\begin{aligned} TCs = \{&(\alpha, 1, x), (\alpha, 2, x), (\alpha, 3, x), (\alpha, -, y), \\ &(\beta, 1, y), (\beta, 2, y), (\beta, 3, y), (\beta, -, y)\} \end{aligned}$$

Once combinations of blocks are obtained using one of these criteria, the actual test cases can be implemented. The blocks represent abstract sets of data, so to implement the concrete test cases actual values have to be selected from these blocks.

3.5 Logic Coverage

This section presents coverage criteria based on logical expressions. As with other types of coverage criteria, logical coverage can be applied on different kinds of artifacts such as code, models and specifications. This class of coverage criteria became popular since its incorporation on standards for safety-critical software [Hayhurst et al., 2001].

The two basic definitions used by logic coverage are *predicates* and *clauses*. A *predicate* is an expression that evaluates to a boolean value. An example of predicate would be the following expression:

$$((a > b) \vee C) \wedge p(x) \tag{3.1}$$

Predicates are composed of:

– boolean variables, such as the variable C in the given example;

– non-boolean variables that are compared using relational operators, such as in $a > b$;

– function calls, such as $p(x)$.

The internal structure of a predicate is created using *logical operators*. Some examples of logical operators are:

– Negation operator ($\neg$);

– AND operator ($\wedge$);

– OR operator ($\vee$);

– Exclusive OR operator ($\oplus$);

– Implication operator ($\Rightarrow$);

– Equivalence operator ($\iff$).

A *clause* is a predicate that does not contain any of these logical operators. For example, the predicate $(a = b) \vee C \wedge p(x)$ contains three clauses: a relational expression $(a = b)$, a boolean variable C and a function call $p(x)$.

As said before, this class of coverage criteria can be applied to different types of software artifacts. Some examples of places where predicates and clauses can be found are conditional statements on the source code, guards on model transitions and specification preconditions.

3.5.1 Logic Expression Coverage Criteria

Now that the concepts of predicate and clause are introduced, it is possible to show how they related to each other. Let P be a set of predicates and C be a set of clauses in the predicates in P. For each predicate $p \in P$, let C_p be the clauses in p, that is, $C_p = \{c | c \in p\}$. C is the union of the clauses in each predicate in P, that is, $C = \bigcup_{p \in P} C_p$ where $p \in P$.

The first two criteria presented in this section are the most basic ones: *Predicate Coverage (PC)* and *Clause Coverage (CC)*.

Predicate Coverage (PC): For each $p \in P$, the set of test requirements contains two requirements: p evaluates to *true*, and p evaluates to *false*.

Clause Coverage (CC): For each $c \in C$, the set of test requirements contains two requirements: c evaluates to *true*, and c evaluates to *false*.

These two criteria are not very effective because: 1) PC does not do verifications at the clause level and 2) tests that cover CC may not cover the actual predicate which the clauses belong. An important observation at this point is to note that PC does not subsume CC and vice versa. Let us take as an example the following predicate:

$$p = (a \vee b) \tag{3.2}$$

Given all combinations of values for a and b, we would have the following results for p:

	a	b	p
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

A test set using tests 2 and 3 would satisfy CC since both a and b have been tested with *true* and *false* values. But the same test set would not satisfy PC since for both test cases the predicate resulted in *true*, missing a test where p evaluates to *false*.

It might be interesting to have a coverage criterion that not only tests individual clauses, but also tests the predicate they belong to. The most direct way to do this would be testing all combinations of clauses. The next criterion presented here does exactly that:

Combinatorial Coverage (CoC): For each $p \in P$, the set of test requirements has requirements for the clauses in C_p to evaluate to each possible combination of truth values.

This criterion is also known as *Multiple Condition Coverage*.

Unfortunately, combinatorial coverage is impractical in many cases. So it is necessary to have a criterion that checks the effect of each clause, but does so requiring a reasonable number of test cases. To solve this problem, there are some criteria based on the notion of turning individual clauses “active” making it possible to test situations where this active clause determines the outcome of the predicate. These criteria are classified as *Active Clause Coverage* (ACC).

The ACC criteria use the concepts of *major clauses* and *minor clauses*. A major clause is the clause we are focusing in a particular test case. Each clause in the predicate is treated as a major clause at some point. The term c_i is used to refer to major clauses. Once you define a major clause for a particular test case, all other clauses are considered minor clauses. The term c_j is used to refer to minor clauses.

When using ACC criteria it is necessary to set the values for the minor clauses in a way that makes c_i determine the outcome of the predicate. In a more formal way, given a major clause c_i in the predicate p, it is said that c_i determines p if the minor clauses $c_j \in p$, $j \neq i$ have values so that changing the truth value of c_i changes the truth value of p.

The basic definition of ACC is the following:

Active Clause Coverage (ACC): for each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. Then, the set of test requirements has two requirements for each c_i: c_i evaluates to *true* and c_i evaluates to *false*.

There are some variations of the ACC criterion that are either more specific or more general about the values used for minor clauses in the test cases. From the most specific to the most general we have: *Restricted Active Clause Coverage*, *Correlated Active Clause Coverage* and *General Active Clause Coverage*.

The first variation of ACC forces c_j to be identical for both assignments of truth values for c_i.

Restricted Active Clause Coverage (RACC): For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. Then, the set of test requirements has two requirements for each c_i: c_i evaluates to *true* and c_i evaluates to *false*. The values chosen for the minor clauses c_j must be the same when c_i is *true* as when c_i is *false*.

It is important to notice that due to the restriction applied by RACC on the values for c_j (they must be the same for the test case which the predicate evaluates to true as for the test case which the predicate evaluates to false) there might be some infeasible test case requirements, mainly when there is dependence between variables of the predicate, something that is common in most software.

The second variation of ACC is a bit more general and does not force the values of c_j to be identical for both assignments of truth values for c_i, but they still require values for c_j that make the p evaluate to *true* and *false* in different test cases.

Correlated Active Clause Coverage (CACC): For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. Then, the set of test requirements has two requirements for each c_i: c_i evaluates *true* and c_i evaluates *false*. The values chosen for the minor clauses c_j must cause p to be *true* for one value of the major clause c_j and *false* for the other.

The third variation of ACC has no requirements at all when it comes to choosing values for the minor clauses c_j.

General Active Clause Coverage (GACC): For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. Then, the set of test requirements has two requirements for each c_i: c_i evaluates to true and c_i evaluates to *false*. The values chosen for the minor clauses c_j do not need to be the same when c_i is *true* as when c_i is *false*.

The problem with this criterion is that p might evaluate to *true* for both test cases, missing a test where p evaluates to *false*.

3.5.2 Comments on MC/DC Coverage

MC/DC (or MCDC) stands for *Modified Condition/Decision Coverage*. MC/DC is a logic coverage criterion that is used by many standards for safety-critical software development, such as the ones required for *Federal Aviation Administration* (FAA) approval. It is present in the RTCA/DO-178B document *Software Considerations in Airborne System and Equipment Certification* which is the primary means used by aviation software developers to obtain FAA approval [Hayhurst et al., 2001].

This criterion uses the concepts of *condition* and *decision* to state its test requirements. A condition is an expression containing no boolean operators (much like the *clause* definition used by the other logic expression coverage criteria). A decision is a boolean expression composed of conditions and zero or more boolean operators (like the previously defined *predicate*). If the same condition appears more than once in a decision, each occurrence should be considered as a distinct condition. A decision without a boolean operator is a condition.

The main objective of MC/DC is to show by execution that each condition in a decision affects the outcome of the decision independently. The independence requirement ensures that the effect of each condition is tested in relation to the other conditions. A more formal description of the MC/DC requirements is presented bellow:

Modified Condition/Decision Coverage: Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.

The definition presented above comes from [Hayhurst et al., 2001] and describes how the avionics industry sees MC/DC. It can also be called "unique-clause MC/DC". This definition corresponds to RACC presented previously.

The original ideas for MC/DC come from [Chilenski and Miller, 1994]. The definitions presented in this paper do not address whether or not the values for the minor clauses should be kept the same for both values of the major clause. It corresponds to GACC presented previously.

There is also another variation called "masking MC/DC" [FAA, 2001]. Using masking MC/DC the values for the minor clauses can change for different values of the major clause. However, the criterion requires the use of boolean logic principles to assure that no other condition is affecting the decision's outcome. This criterion corresponds to CACC that was also presented previously.

Chapter 4

Related Work

Formal methods and testing are two techniques used to develop systems that are more robust and reliable. For some years now there has been an effort from both formal methods and software testing communities to combine these two techniques. An important line of work regarding this combination focuses on model-based testing techniques to generate test cases from formal models.

The current work in this field varies a lot. The work we have found during our research covers different types of tests (such as unit testing, module testing, system testing etc.); they employ different coverage criteria; their goals may vary between testing the implementation or the model itself; they generate tests from different formal notations; and they have different levels of tool support.

The criteria used to select the work we reviewed in this chapter was the following:

- The paper has to describe the proposed approach in enough details so the reader can understand how it works;
- The approach should generate test cases for the software implementation or for the model itself;
- The approach has to be at least partially tool supported;
- Preferably, it should use formal notations that use abstract state machine concepts or share other B-Method characteristics (such as specifications based on invariants and preconditions). The formal notations considered were: ASM, Alloy, B, Event-B, JML, VDM and Z.

4.1 Current state of the art

In the next subsections we discuss the most relevant work found during our research. In each subsection, we summarize the proposed approach and the main contributions of a particular paper.

4.1.1 Amla and Ammann

In [Amla and Ammann, 1992], the authors present a method to perform *Category Partition Testing* [Ostrand and Balcer, 1988] based on Z specifications. Category Partition Testing is a testing strategy that uses informal specifications to produce test requirements for the system under test. These requirements are obtained by partitioning the system's input space. The partitions are defined by hand, using the test engineer own set of criteria. Once the partitions are defined, test case specifications are created using a language called TSL. Ultimately, these specifications serve as input for a tool that generates test case scripts for the system.

In this paper, the authors show how this technique can be applied using Z specifications instead of informal, natural language requirements. They believe that most of the work required to perform category partitioning is already done during the formal specification of the system. Invariants, pre- and post-conditions already restrict the values for parameters and variables to certain categories, in a more formal fashion. The use of formal specifications for this job avoids rework, and besides that, can provide a specification method that is more reliable than informal requirements.

The steps to create test case specifications for the system under test are the following:

1. *Identify testable units in the specification:* select Z schemas that represent units that can be tested separately;

2. *Identify parameters and variables:* identify parameters and variables that affect the behaviour of the unit under test;

3. *Identify categories:* identify significant characteristics of the parameters and variables that can be explored to create categories;

4. *Partition categories:* partition each category into sets of possible values;

5. *Define test results:* specify the possible results for each test case.

All these steps are performed manually to obtain the information necessary to write the test case specifications using TSL. Tool support only exists to translate the TSL specifications to test scripts.

4.1.2 Dick and Faivre

In [Dick and Faivre, 1993], the authors presented techniques to automate the partition analysis of operations in a formal model using *Disjunctive Normal Form* (DNF). They also showed how to use these partitions to build a *Finite State Automata* (FSA) that can be used to generate test cases.

In summary, their test case generation approach consists of two main steps:

1. *Perform partition analysis for each operation in the model:* the definition of each operation (a combination of *pre-*, *post-conditions* and *invariants*) is obtained and improved by eliminating redundant predicates and adding typing information. Then, this definition is parsed to disjunctive normal form to obtain disjoint sub-relations (or sub-operations);

2. *Build an FSA to find test cases:* once the sub-operations are created, they are used to build an FSA. Each sub-operation represents a transition in the FSA which then can be traversed to obtain a set of test cases. In the end, it is necessary to obtain valid input values to perform each transition or, in other words, find valid input data to execute each sub-operation.

In the proposed approach, a test case is a sequence of calls to sub-operations which begins in an initial valid state and covers a predefined number of sub-operations in the FSA at least once.

The authors presented examples of how the approach could be applied to VDM [Plat and Larsen, 1992] models. They also developed a tool to automate most of the process, but the generation of the FSA and the generation of test data for the sub-operations had to be done by hand.

The work of Dick and Faivre served as inspiration for many of the papers presented in this chapter. After this paper, the idea of using DNF to partition the input space of the system under test was used by many other authors in the field.

4.1.3 Marinov and Khurshid

In [Marinov and Khurshid, 2001], Marinov and Khurshid present *TestEra*, a framework for automated testing of Java programs. The framework uses the *Alloy* and *Alloy Analyzer* [Jackson et al., 2000] as tools to generate, execute and evaluate test cases for Java programs.

Alloy is used as a specification language to write invariants that describe the input space of the program that is being tested. These invariants are used by Alloy Analyzer to generate input instances to test the program; all generated instances should satisfy the specified invariant.

Since the instances generated by Alloy Analyzer use abstract data types, it is necessary to create functions that translate them to concrete data types. All necessary translation functions have to be implemented by hand. After the concrete data is obtained, it is used to execute test cases on the software implementation. The outputs generated by the program are then translated back to abstract data types – also using translation functions – so its correctness can be evaluated.

The correctness of the output produced by the program is evaluated by a *correctness criterion*, which is an Alloy function that verifies if the produced output is correct according to the tested input. If the evaluation fails, TestEra presents a counter-example that violates the correctness criterion.

The work of [Marinov and Khurshid, 2001] lacked a testing strategy to define the test cases, leaving it to the test engineer's own experience to specify interesting data for the test cases.

4.1.4 Ambert et al.

In Ambert et al. [2002], the authors present BZ-TT (B and Z Testing Tools), a tool for test case generation based on B and Z formal specifications. The method used by the tool was first presented in Legeard et al. [2002]. It is based on constraint solving, and its goal is to test every operation of the system at every reachable boundary state. A *boundary state* is a system state where at least one of the state variables has a maximum or minimum boundary value. The approach then follows with the execution of operations on every obtained boundary state; the operations are executed using extreme boundary values as their parameters.

The objective of BZ-TT is to test an implementation that was not derived via refinement from the formal model. The tool is capable of generating test cases from both B machines and Z schemas. To do this, it first translates the input models to an intermediate format that uses logical programming to represent the model's constraints. These files are used as inputs for a constraint solver that generates the test cases.

The proposed method relies on the CLPS-BZ constraint solver Bouquet et al. [2002] to generate the test cases. It is used to find the boundary states, and to animate the models to obtain an oracle verdict.

The generated test cases are sequences of operations calls that begin with the initialisation of the model and end in a boundary state. The method supports the generation of positive test cases and negative test cases. In a positive test case, all operations in the trace have their preconditions respected. On the other hand, a negative test case will violate the precondition of the last operation call in the trace.

Each generated test case has four parts:

1. *Preamble:* a sequence of operations that takes the machine to a boundary state;
2. *Body:* an operation call that modifies the current state of the machine;
3. *Identification:* an operation call that observes the state of the machine;
4. *Finalisation:* an operation call that takes the machine back to its initial state so other tests can be executed in sequence.

After taking the machine to a boundary state on the preamble, in the operation body, an operation which modifies the current state of the machine is executed. All operations that modify the state of the model are executed in every boundary state and are instantiated using extreme boundary values as parameters. On the identification step, operations that observe the state of the machine are instantiated so that a verdict about the test case can be given. A test case passes if all the output values returned by the concrete implementation of the trace are equivalent to the output values returned by the model during the simulation of the same trace. On the finalisation step, the machine is taken to its initial state again so other test cases can be executed.

After the test cases are generated, they are automatically translated into executable test scripts, using a test script pattern and a reification relation between the abstract and concrete operation names, inputs and outputs.

The tool establishes some restrictions on the specifications it can receive as input. It requires that all data structures must be finite. This means that any given set must be enumerated or of a known finite cardinality. It also requires that the specifications are monolithic, which means that they should be contained into a single file and cannot be structured in modules. Also, it requires that all operation's preconditions to be explicit, no preconditions can be left implicit on the body of the operation.

4.1.5 Cheon and Leavens

In [Cheon and Leavens, 2002] the authors present a tool that combines JML and JUnit to make the process of writing and maintaining unit tests easier. The tool is called *jmlunit* and it generates unit tests for Java programs based on JML specifications. Their approach focuses mainly on oracle implementation for Java programs using JML pre- and post-conditions. The authors state that the implementation of test oracles consists in coding the verification of post-conditions. With that in mind, they use JML post-conditions to perform oracle verifications automatically.

Jmlunit generates a test class for a JML annotated Java class. The test class contains one test method for each method in the class being tested. When a test case is executed, JML runtime assertion checker is used to verify if any exception is thrown during its execution. Instead of defining expected results and comparing it with the obtained outputs, the tests monitor the behaviour of the class being tested, observing exceptional behavior. If the method under test throws no exception, the test passes. If a precondition violation is thrown, the test is considered not important. If any other type of exception is thrown then, the test fails.

This work does not address test data generation or criteria to select test data. In this approach, the input data for the test cases is defined by hand – taking into consideration the test engineer own criteria – using JUnit test fixtures. A global test fixture has to be implemented in the test class. The data defined in the fixture is then used to execute the tests.

4.1.6 Satpathy et al.

In Satpathy et al. [2005], the authors developed the ProTest tool. ProTest is based on the ProB animator [Leuschel and Butler, 2003] and uses model checking to generate tests through a state graph for a B machine. In this graph, each node represents a machine state, and each edge represents an operation that takes the machine to another state. Each path beginning in the initial state of this graph is a test case.

To generate this graph, the input domain of the operations is partitioned into subdomains. Each one of these subdomains represents a possible scenario in which the operation can be instantiated. The process of input domain partitioning and test case generation is described by the following steps:

1. The operation precondition is converted to its *Disjunctive Normal Form* (DNF);

2. In case there are conditionals in the body of the operation, formulas representing the conditional choices are created and added to the precondition using conjunction. In case of conditional statements inside an ANY construct, the conditional is ignored since it might be related to bound variables that are not part of the initial state;

3. Possible contradictory clauses that might be added on the previous step are filtered using naive theorem proving. The result after this filter are the disjunctions C_1, C_2, ..., C_k which divide the input domain of the operation into k subdomains;

4. k instances of the operation are created. This way, each instance of the operation will be executed when the condition C_i is attended;

5. The process is repeated to every operation of the specification;

6. The full state of the B machine is explored to construct a *Finite State Machine* (FSM). Each node in the FSM represents a possible machine state and each edge is labeled by an operation instance. To explore the full state space it is assumed that all the sets of the specification are of finite type, and they are small in size;

7. The FSM is traversed to generate a set of operation sequences such that each operation instance in the FSM appears in the generated sequences at least once. Each operation sequence should start with the initial state. Each sequence constitutes one test case for the subsequent implementation. A set of test sequences represents a test suite.

ProTest allows the test cases to be translated into executable tests written in Java and does the verification of the results in a similar manner of Ambert et al. [2002]. It executes the tests and animates the specification in parallel to compare the results from both sources. This way, the state obtained after the execution of the concrete test case can be compared with the stated obtained after the animation of the specification (regarding the same test sequence) to give a verdict about the test case.

The method requires every operation of the specification to be implemented in the concrete code. The operations of the machine are divided into update operations, which are operations that can modify its state, and probing operations, which are operations that only query its state. The probing operations are used to query properties of both the specification state and the implementation state to handle the test verdict process. A mapping between the namespace of the state specification and concrete state is required so that comparisons can be made properly.

One of the advantages of this method is that it allows partial verification of the tests. It means that intermediate test results can be analyzed, not only the final results of the test.

The tool has some restrictions regarding the machines it receives as input. Machines should be monolithic (single file machines), operations should have only one return variable and should use only basic types (types that relate to the typing usually present on programming languages such as integers, booleans etc.) and simple data structures. Also, operations should not have non-deterministic constructs.

Even though the tool does not support non-deterministic constructs, the authors have suggested a solution to deal with non-determinism using a method they call testing on the fly. The method requires that operations with non-deterministic constructs make their choices visible through extra return variables added to the operation. For each non-deterministic choice on the operation body, an extra return variable is added. This way, during the test execution, these variables can be used to consult the choices made and guide the test execution.

The authors presented a simple case study to evaluate the tool that showed that a high number of partitions created could not be covered. The reason for this was that some of the partitions could have inconsistencies or contradictions that were not eliminated. Also, in some cases the operation instances obtained could not be reached from the initial state.

The same authors revisited the method in Satpathy et al. [2007], adapting it in a way so it can be used for other specification languages model oriented such as Z and VDM. They also went into more details about the solution for the problem of operations which have non-deterministic behavior.

4.1.7 Gupta et al.

The authors in Gupta and Bhatia [2010] proposed an approach to generate functional tests from B specifications. The proposed approach starts with informal requirements written in English. Each requirement is then manually translated into B constructs; each construct is annotated with an identifier that maps it to one of the informal requirements. These annotations are used later to verify which requirements are covered by the generated test cases. The specification is then validated on *AtelierB* and animated and tested on *ProB*.

Their test generation framework works as follows: *pre* and *post* conditions are extracted from *.mch* files using *AtelierB* and are parsed; then, according to a test selection criterion, the model is covered. The approach uses decision coverage as coverage criterion. This coverage criterion requires that, for each boolean decision, there should be a test case in which it evaluates *true* and a test case in which it evaluates *false*. In the end, a coverage matrix relating test cases and informal requirements is created so the test engineer can check the level of coverage.

Like other work in the field, the objective of this approach is to test all reachable paths present in the specification. On this approach, a test case consists of a sequence of operation calls. Each test case has four parts: a *preamble* that puts the system in the desired state for test execution, a *body* that executes the operation under test, a *observation* phase that uses query operations to check for tests results and a *postamble* that brings the system back to its initial state so other tests can be executed.

The authors did not provide more details about the approach in the paper, and the tool was not available for download, so it was not possible to perform a deeper evaluation. The proposed approach only concerns with testing a requirements artifact, trying to certify that errors introduced in a requirements artifact do not propagate into other phases of the software development life cycle. This approach does not deal with testing the actual software code; it only deals with the software requirements.

4.1.8 Dinca et al.

In [Dinca et al., 2012] the authors present a plugin for the Rodin tool[1] to generate conformance tests from Event-B models. The plugin implements a model learning approach that iteratively constructs an approximate automaton model together with an associated test suite. The authors also use test suite optimization techniques to reduce the number of test cases generated.

For a given Event-B model, the approach constructs, in parallel, a finite automaton and a test suite for the system. The finite automaton represents an approximation of the system that only considers sequences of length up to an established upper bound l. The size of the finite automaton produced for coverage is significantly lower than the size of the exact automaton model. By setting the value of the upper bound l, the state explosion problem normally associated with constructing and checking state-based models is addressed.

The tool takes as input an Event-B model and the finite bound l and outputs a finite automaton approximating the set of feasible sequences of events from the given model of length up to l and a test suite. The set of sequences includes test data that make the sequences executable.

[1]Rodin project website: http://www.event-b.org/

The automaton can also be improved by providing additional data, such as additional refinements, counter examples or new sequences that can increase the precision of the finite-state approximation.

There are many existing methods for test case generation from finite state models. In this work, the authors use internal information from a learning procedure. This procedure maintains an observation table that keeps track of the learned feasible sequences. The sets of sequences in this table provide the desired test suite. The obtained test suite satisfies strong criteria for conformance testing and may be large. If weaker test coverage like state-based, transition-based or event coverage are desired, optimization algorithms can be applied.

The tool was implemented in Java, uses ProB as a constraint solver to check feasibility of the test sequences and can be used on Event-B models with several levels of refinements.

4.1.9 Cristiá et al.

In Cristiá et al. [2014] the authors present an approach to generate test cases based on strategies to cover logical formulas and specifications. The authors discuss that most of the current work in the field focus on generating test cases that cover automatons. They argue that, in some cases, coverage criteria for logical formulas is more natural and intuitive. With that in mind, the authors propose a set of coverage criteria that they call testing strategies. The approach focuses on notations that rely heavily on logical formulas to specify models, such as the B and Z notations. In their work, the authors experimented with the approach using the Z notation.

The testing strategies presented in their paper define rules that indicate how to partition the input domain of a model specified using a logical expression. They analyze the structure, semantics and types of these logical expressions and identify relevant partitions to test them. The strategies are organized in a way that a test engineer could choose to perform weaker or more in-depth tests by combining them in different ways. Depending on the types of logical constructs used in the formula, different testing strategies may be employed. In total, the authors presented a set of eleven testing strategies.

The use of coverage criteria for logical expressions makes sense in many cases. Generating automatons from specifications that use logical formulas, in some cases, may result in a single transition that is not useful for test case generation, even though the same formula may not be trivial to cover.

The strategies presented in their work were implemented in a tool called FASTEST. The tool hides the partition process and uses a scripting language to allow test engineers to define new testing strategies as needed.

4.2 Final Discussions

Table 4.1 presents an overview of the work discussed in this chapter. Besides the work mentioned in the previous sections, many other papers were found in the current literature that dealt with model-based testing using formal models [Satpathy et al., 2007, Burton and York, 2000, Singh et al., 1997, McDonald et al., 1997, Fletcher and Sajeev, 1996, Bouquet et al., 2006, Xu and Yang, 2004, Aichernig, 1999, Nadeem and Ur-Rehman, 2004, Nadeem and Lyu, 2006, Mendes et al., 2010].

Most of the researched work use not well-defined criteria to generate test cases. As seen in Table 4.1, there are papers that use *ad hoc* criteria for test case generation. Another common problem in the work we found in the current literature is the lack of tool support. In general, the developed approaches are only partially supported by tools (in some cases third-party tools). Also, in many cases, these tools are not available to the public.

Also, we found no paper that addressed the problem of test data concretization in a more in-depth way. Some papers like [Marinov and Khurshid, 2001] and [Satpathy et al., 2005] used manually implemented functions to map the translation between abstract and concrete test data, but none of them provided a strategy that could be automated to solve this problem. Regarding oracle strategies for test case evaluation, most of the strategies used simple oracle solutions, not providing many options and configurations for the test engineer. Ultimately, few papers dealt with the problem of generation of executable test scripts as well. In our work, we tackle the aforementioned problems, implementing our solutions in a tool that is free, open-source and publicly available for evaluation.

Table 4.1: Related Work Overview: for each relevant work found during our research, the table presents: the level of tests generated by the proposed approach, the formal notation supported, the testing criteria used by the approach and the level of tool support.

Citation	Level	Notation	Testing Criteria	Tool Support
[Amla and Ammann, 1992]	Unit	Z	Category Partitioning/Equivalent Classes	There is a tool to translate the manually generated specifications into tests scripts (The tool was not available)
[Dick and Faivre, 1993]	Module	VDM	FSA Coverage	Partial tool support: the tool does not generate the FSA nor the test data for the test cases (The tool was not available)
[Marinov and Khurshid, 2001]	Unit	Alloy	*ad hoc*	Alloy Analyzer is used as a constraint solver to support the approach. There are steps of the approach that still need to be performed manually (The constraint solver is available)
[Ambert et al., 2002]	Module	B and Z	Coverage of boundary states	All the steps are tool supported but the tool applies some restrictions to the machines it can support (The tool was not available)
[Cheon and Leavens, 2002]	Unit	JML	*ad hoc*	There is a tool that generates JUnit class skeletons but the test data has to be created by hand (The tool was not available)
[Satpathy et al., 2005]	Module	B	FSA Coverage	The approach is supported by ProB (The tool is available)
[Gupta and Bhatia, 2010]	System	B	*ad hoc*	The approach is supported by Atelier-B and ProB
[Dinca et al., 2012]	Module	Event-B	FSA Coverage	Supported by a Rodin Plugin (The tool is available)
[Cristiá et al., 2014]	Unit	Z	Coverage of Logical Expressions	All the steps of the approach are tool supported (The tool is available)

Chapter 5

A B Based Testing Approach

BETA is a tool-supported approach to generate test cases using B-Method abstract state machines. The test cases generated by the approach are used to test a software implementation that was automatically generated or manually implemented using this type of models. BETA uses input space partitioning and logical coverage techniques to create unit test cases. This chapter presents the approach and its process to derive test cases from abstract B machines.

5.1 The Approach

Figure 5.1 presents an overview of the BETA approach and each of the steps of its test generation process. The approach is automated by the BETA tool. In summary, the approach starts with an abstract B machine, and since it generates tests for each unit of the model individually, the process is repeated for each one of its operations. Once an operation is chosen, the approach acts accordingly to the testing technique selected. If *Logical Coverage* is the chosen technique, it inspects the model searching for predicates and clauses to cover. Then, it creates test formulas that express situations that exercise the conditions/decisions which should be covered according to one of the supported logical coverage criteria. If *Input Space Partitioning* is the chosen technique, it explores the model to find interesting characteristics about the operation. These characteristics are constraints applied to the operation under test. After the characteristics are enumerated, they are used to create test partitions for the input space of the operation under test. Then, combinatorial criteria are used to select and combine these partitions in test cases. A test case is also expressed by a logical formula that describes the test scenario. To obtain test input data for each of these test scenarios a constraint solver is used. Once test input data is obtained, the original model may be animated using these inputs to obtain oracle data (expected test results). Test inputs and expected results are then combined into test case specifications that could be either in XML or HTML format. The test case specifications are used as a guide to code the concrete test cases. A separate module uses the XML specifications to generate partial test scripts that help the developer in the coding process.

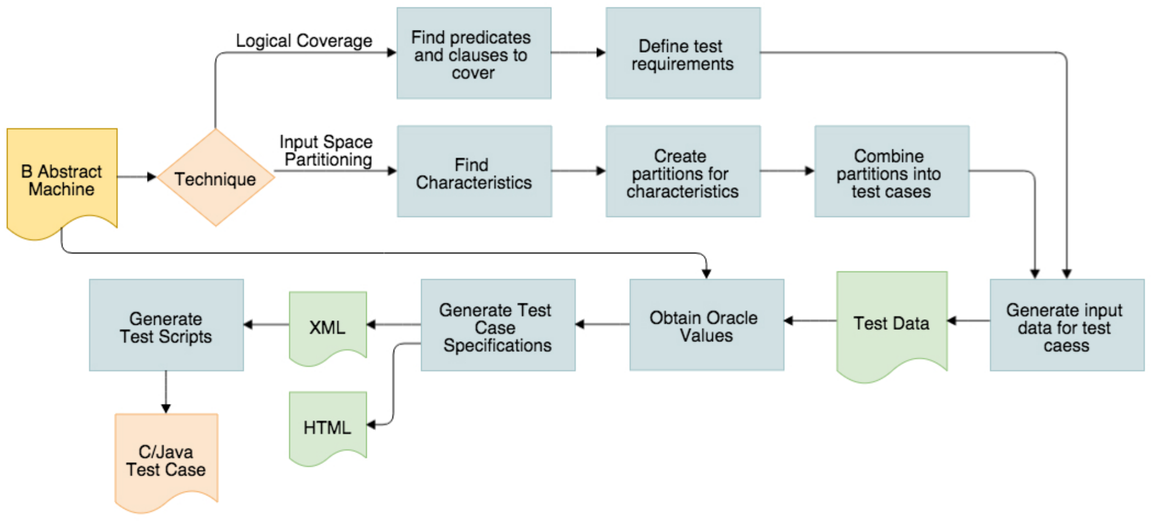


Figure 5.1: An overview of the BETA approach.

5.1.1 B Abstract Machine

An abstract machine is the highest level of model abstraction present in the B-Method. It is the specification that usually starts all the modeling process. BETA uses these machines to generate test cases.

One of the advantages of using the initial model as the basis for the test case generation process is that it is possible to make a connection between the abstract model and the actual implementation using the generated test cases. The test cases make it possible to check if the behavior specified at the beginning of the development process is present in the final product. The approach considers that the model used as input was previously verified on a proper modeling tool and did not contain any specification errors.

BETA currently supports all the notation used by abstract B machines with the exception of *structs* constructs. It does not support test case generation from *refinements* and *implementations*.

5.1.2 Logical Coverage

Logic-based coverage criteria use logical predicates and clauses to define test requirements. Thus, to generate test cases for this type of criterion, first it is necessary to obtain the predicates and clauses related to the operation under test. Once it is done, test requirements are defined based on the chosen coverage criterion. Lastly, test cases for these requirements are identified and expressed using logical formulas.

Obtaining predicates and clauses

The *B Language Reference Manual* [Clearsy, 2011] of the *AtelierB* tool was used as a reference to find interesting places with predicates to cover in a B machine. This document contains the B notation grammar which was used to find *Predicate* rules. This rule defines all kinds of predicates that can be written using the B notation. It can be found in many places of the grammar. Currently, BETA searches for predicates to cover on the following places:

– *Precondition* substitutions;

– *Assertion* substitutions;

– *If* substitutions;

– *Case* substitutions*;

– *Select* substitutions;

– *Any* substitutions.

There is one special case in this list: the *Case* substitution. It is considered a special case because instead of using predicates as guards, this substitution uses *Term* rules that should be considered as clauses for logical coverage. So for this substitution a set of clauses that have the form "$Term_a = Term_b$" (where $Term_a$ is the switch expression and $Term_b$ is a branch expression) is created.

Predicates present on *Constraints*, *Properties*, *Invariant*, and *Assertions* clauses are not covered. Currently, the focus is on substitutions that directly affect the behavior behavior of the operation. Also, covering clauses in places such as the invariant would result in test requirements that would break the invariant in some cases. This would result in a test cases that require the software to be in an inconsistent state during the its execution. Even tough it would be interesting for security testing, it is not one of the current objectives of the approach. Predicates used on *While* substitutions are also not covered since they are only used on the implementation level of the model.

Once a set of predicates to cover is defined, the predicates are broken into clauses so that some clause related coverage criteria can be applied. A clause should be a predicate that is not connected by any logical operator. In the grammar level, in most cases, a clause is also defined by a *Predicate* rule.

Defining test requirements

BETA currently supports four logical coverage criteria (more detailed information about the test criteria can be found on Section 3.5):

- *Predicate Coverage (PC)*: For each predicate p in the set of predicates to cover, the set of test requirements contains two requirements: p evaluates to *true*, and p evaluates to *false*;
- *Clause Coverage (CC)*: For each clause c in the set of clauses to cover, the set of test requirements contains two requirements: c evaluates to *true*, and c evaluates to *false*;
- *Combinatorial Coverage (CoC)*: For each predicate p in the set of predicates to cover, the set of test requirements has requirements for the clauses in p to evaluate to each possible combination of truth values;
- *Active Clause Coverage (ACC)*: for each predicate p in the set of predicates to cover, the set of test requirements has requirements to check that each clause of p individually affects the outcome of p. There are many variations of ACC (see Section 3.5 for more information). BETA currently uses *Correlated Active Clause Coverage*.

Logical formulas are used to define the test requirements. For PC, CC, and CoC criteria the process is straightforward. For PC, each predicate p gets two test formulas, one for p and another for $\neg p$. For CC, each clause c gets two test formulas, one for c and another for $\neg c$. For CoC, it is necessary 2^n test formulas (where n is the number of clauses in the predicate) to cover all possible combinations of truth values for the clauses in the predicate that is being covered. For example, to cover $p = a \vee b$, it would be necessary to test situations where: $a \wedge b$, $\neg a \wedge b$, $a \wedge \neg b$, and $\neg a \wedge \neg b$.

For ACC, the process is more complicated. ACC requires each clause of a predicate to be treated as a major clause (c_i) at some point. Once a major clause is defined all other clauses are considered minor clauses (c_j). The criterion requires that values for the minor clauses are chosen in a way that the value of the major clause determines the outcome of the predicate. By doing this, it is possible for the tests to swap the values of the major clause in a way that it actually affects the outcome of the whole predicate.

To find values for the minor clauses, BETA uses a process inspired by a technique presented in [Ammann and Offutt, 2010]. Consider a predicate p with a clause or boolean variable c. Let $p_{c=true}$ represent the predicate p with every occurrence of c replaced by $true$ and $p_{c=false}$ be the predicate p with every occurrence of c replaced by $false$. After these replacements, note that $p_{c=true}$ and $p_{c=false}$ will not contain any occurrence of the clause c. To find values for the minor clauses so that the major clause can determine the outcome of the predicate, one can use the following formula:

$$p_c = p_{c=true} \oplus p_{c=false} \tag{5.1}$$

This formula describes the exact conditions under which the value of c determines that of p. If the values for the clauses in p_c are chosen so that p_c is true, then the truth value of c will determine the truth value of p.

Let's use the predicate $p = x > 0 \lor y < 5$ as an example to better explain the process. By definition $p_{x>0}$ is:

$$p_{x>0} = p_{x>0=true} \oplus p_{x>0=false} \tag{5.2}$$

$$p_{x>0} = (true \lor y < 5) \oplus (false \lor y < 5) \tag{5.3}$$

$$p_{x>0} = true \oplus y < 5 \tag{5.4}$$

$$p_{x>0} = \neg(y < 5) \tag{5.5}$$

This evaluation states that if we want $x > 0$ to determine p the clause $y < 5$ must be $false$. It might seem obvious for a simple predicate like this one but the step by step process is very helpful when dealing with more complex ones.

The following two formulas describe the test case values for the test cases that verify if the major clause $x > 0$ determines the outcome of p:

$$x > 0 \land \neg(y < 5) \tag{5.6}$$

$$\neg(x > 0) \land \neg(y < 5) \tag{5.7}$$

These formulas can then be used on a constraint solving tool to find actual values for x and y.

5.1.3 Input Space Partitioning

Input Space Partitioning testing techniques use the concepts of characteristics and partitions to group equivalent values of test data. The idea is that values from the same group would trigger the same behavior in the software implementation, so it is only necessary to pick one value from each of these groups during the testing process. Usually, it can reduce the number of tests considerably.

The process of partitioning the input space of the operation under test starts with identifying its input parameters. Once the input parameters are identified, it is necessary to find characteristics (constraints) about them that are specified in the model. The characteristics are then used as the basis to partition the operation's input space using *Equivalence Classes* or *Boundary Value Analysis* techniques. The blocks created for the partitions are then combined using combinatorial criteria to define test requirements.

Obtain Characteristics

To define test scenarios the approach uses restrictions applied to the operation under test that are described in the abstract model. These restrictions are called *characteristics* of the operation. These characteristics are the basis for the test input selection process.

To find these characteristics first it is necessary to define the *input parameters* of the operation under test. The set of input parameters is composed by all of the operation's parameters and the machine's state variables. The parameters of the operation can be found just by looking at its list of parameters. To find the state variables requires more work. For optimization purposes, only a subset of the state variables is considered. Only variables mentioned in the operation's precondition, used on guards of conditional substitutions, and that relate to the previous variables via invariant clauses are considered. The last ones are required because their values may affect the other variables due to transitivity. The search on invariants is also extended to invariant clauses from other modules that are used by the operation's machine, via *sees*, *uses*, *includes* and *extends* clauses.

After the set of input parameters is defined, the next step is to define the characteristics for the operation under test. It is done by finding restrictions applied to the input parameters. On the B-Method this restrictions are usually expressed in the form of logical clauses. These logical clauses represent characteristics of the operation under test that should be tested. They might define types for parameters and variables or express constraints that should be respected by the software. The approach searches for characteristics on:

– *Invariant* clauses;

– *Properties* clauses;

– *Precondition* clauses;

– *If* substitutions used by the operation;

– *Case* substitutions;

– *Select* substitutions;

– *Any* substitutions.

As with the input parameters, the invariant and properties clauses from other modules are also included in the search for characteristics.

Each logical clause that mentions one or more input parameters will be added to the set of characteristics of the operation under test. These characteristics will guide the remainder of the approach. BETA uses them to define partitions that will be used by its test cases.

Create test partitions

After these characteristics are enumerated, the approach uses input space partitioning techniques to create test partitions (or blocks) based on them. Considering a characteristic as a particular property of a input parameter, each block extracted from this characteristic represents a set of values that are considered equivalent to test this property. These blocks are disjoint, meaning that a value can not belong to two blocks at the same time. The blocks are also complete, meaning that the union of all the blocks of a characteristic covers its entire domain.

Using the BETA approach, each characteristic can be partitioned up to four blocks of test data, depending on the predicate that defines it and the chosen partition strategy. The approach currently supports two partition strategies: *Equivalent Classes* and *Boundary Value Analysis* (Section 3.4). More details on how each type of predicate is partitioned into blocks can be found on BETA's website[1]

In most of the cases, the approach generates two blocks for each characteristic: one block for positive tests and another block for negative tests. There are two exceptions to this rule:

- cases in which the formula states that some variable in the input space accepts values from a range of values, also called interval (e.g., $x \in 10..25$). In this case the partition may be composed of three or four blocks, depending on the chosen partition strategy.
- cases in which the negation of the formula corresponds to situations that normally do not generate interesting tests. In this case the characteristics correspond to a trivial one block partition.

If the characteristic represents an interval and equivalence classes are used to partition the characteristic, the partition is composed of three blocks: one block for values inside the interval, one block for values preceding the interval and one block for values following the interval.

In cases where the boundary value analysis technique is chosen, the proposal is to cover it with four values, corresponding to the two valid limits of the range and its two invalid limits (immediately below and above the range). This can correspond to four blocks: one block containing the values below the limit, one block containing the values above the limit, and two blocks for the valid range, where each one contains one of the valid limits.

The approach considers two situations to be "not interesting" when partitioning a characteristic in blocks:

1. when a characteristic declares the typing of a variable, it does not create a block for invalid values since they will most likely represent values of a different type;
2. when the origin of a characteristic is the invariant or the properties of the machine, do not generate a block of invalid values as well. The approach currently considers that a test begins in a valid state so the values for the state variables must not be forced to be invalid by a negative block.

[1]BETA User's Guide: http://beta-tool.info/user_guide.html

For the first situation, the approach avoids the creation of a negative block because it will most likely result in a compilation error in the concrete test case (considering that they will be implemented in a strongly typed programming language). For the second situation, although this might be important for security analysis, currently it is not an objective of BETA to test the system when it is in an inconsistent state.

Combine partitions into test cases

After the definition of the blocks, it is necessary to decide how they are going to be used in the test cases. The first thought might be to test all the possible combinations of blocks. Unfortunately, in most cases, due to a high number of blocks created, to test all possible combinations of blocks is impractical. Therefore, the approach has to provide more efficient strategies to combine these blocks. To do this it relies on combinatorial criteria. BETA currently supports three combinatorial criteria:

– *Each-choice*: one value from each block for each characteristic must be present in at least one test case. This criterion is based on the classical concept of equivalence classes partitioning, which requires that every block must be used at least once in a test set;

– *Pairwise*: one value of each block for each characteristic must be combined to one value of all other blocks for each other characteristic. The algorithm used by BETA for this criterion is the *In-Parameter-Order Pairwise* presented in [**?**];

– *All-combinations*: all combinations of blocks from all characteristics must be tested. As mentioned before, this criteria is usually impractical to perform if the partitioning has a high number of blocks. The approach still provides this option in case a test engineer wants to use it. It might be useful if the number of blocks in the partitioning is not too large.

The final result of this combination will be a set of formulas that represent test cases. Each formula is a conjunction of blocks and represents a portion of the input domain of the operation under test. Each formula specifies the input data requirements for a different test case.

5.1.4 Use constraint solving to obtain test inputs

After the definition of input data requirements using logical formulas, it is necessary to find test case values for the input parameters of the operation under test that cover these requirements.

BETA relies on constraint solving tools for this task. It currently uses ProB's constraint solver. ProB [Leuschel and Butler, 2003] is an animator and model checker for the B-Method. BETA uses ProB interfaces (command-line and java API interfaces) to interact with its constraint solver.

When providing a test formula as input for the constraint solver it may result in one of the following: 1) the constraint solver provides values for the input parameters that satisfy the test formula; or 2) the constraint solver can not find any combination of input parameters that would satisfy the test formula.

In the first situation, the values provided by the constraint solver are used as abstract test input values for the test cases. If different combinations of values satisfy the formula, any of them can be selected since they all satisfy the same requirement. The second situation can happen for two reasons: either the constraint solver could not generate test case values due to its limitations, or the test formula represented an infeasible test case scenario. An infeasible test scenario can be a result of some inconsistency in the test formula, for example, the formula may require two conditions that can not be true at the same time.

5.1.5 Obtain oracle values

Obtaining the test inputs for each test case is just the first part of the test creation process. Once the input values are defined, it is necessary to find out what are the expected results for the test. To do this, BETA uses the original model to check what would be the correct results for a particular test case according to what was specified.

This step can either be done manually or automatically, depending on the kind of machine that is being used to generate the test cases. If the machine contains only primitive values, such as integers and booleans, the generation of oracle values can be done automatically. If the machine has any abstract values, such as abstract sets, the process has to be done manually. It is necessary due to limitations of ProB's constraint solver.

When doing it automatically, BETA interacts with ProB's API to animate the original model, calling the operation under test and passing as parameters the inputs for the test. When the process has to be done manually, the test engineer has to open the original model using ProB's interface, and animate the operation under test using the test inputs.

That is the general rule applied to positive test cases. Negative tests, on the other hand, violate the operation's preconditions, and this means that the expected behavior of the operation is not specified for them. In this case, the criteria used by the oracle to evaluate the results must be defined by the test engineer.

Still regarding test oracles, the BETA approach supports the implementation of some oracle strategies which determine what kind of verifications are done by the test oracle. It supports four strategies that can be used separately, to make weaker verifications, or combined, to make stronger verifications. These strategies were implemented in the test case generator by [Souza Neto and Moreira, 2014] and are inspired by the ones presented in [Li and Offutt, 2014]. The four strategies are:

– *Exception checking*: executes the test and verifies if any exception is raised;

– *Invariant checking*: executes the test and after its execution verifies if the invariant is preserved. The invariant checking is done using a *checkInvariant()* method;

– *State variables checking*: executes the test and verifies if the values for the state variables are the ones expected;

– *Return variables checking*: executes the test and verifies if the values returned by the operation are the ones expected.

The oracles generated in this step are used to check if the software implementation actually behaves as specified in the abstract model.

5.1.6 Generate Test Case specifications

Once the test cases are generated, BETA creates test case specifications that can help the test engineer to code the concrete tests.

Each test case specification contains a set of tests for an operation of the model. This tests were created using a particular combination of partition strategy and combinatorial criteria for the blocks. Three parts compose a test case:

1. *Prefix*: in some cases it is necessary to put the system in a particular state before the execution of the test case. In this case the test engineer can use values generated for the state variables to set the state variables of the system into the desirable state before the test execution;

2. *Execution*: the second part consists on executing the operation under test. If it contains input values the approach uses the generated test input values to execute the operation. If it doesn't have input values then it's done by a simple operation call;

3. *Oracle evaluation*: in this part the approach uses the generated values for the test oracles to verify if the software behavior was the one expected.

The tool also supports the generation of test specifications in XML format. The XML specifications contain the same information present on the HTML test specifications, but presented in a structured way so they can be used by other tools to translate the test cases into executable test code.

5.1.7 Generate Test Scripts

To reduce the effort needed to code the concrete test cases, BETA has a separate module that can translate a XML test case specification into partial test scripts implemented in Java or C++. The generated test script still has to be adapted before it can be executed. Adaptations are required when there are differences between data structures used in the abstract model and the final implementation. Also, names of variables and methods may change during the refinement and development process.

The code generation module also implements the oracle strategies mentioned in Section 5.1.5.

5.2 Comments on the evolution of BETA

The BETA approach and tool has been under development since the author's masters course. We believe it is important to clarify what were the contributions made during the period of this doctorate, showing the evolution of the approach since the masters dissertation [Matos, 2012] was presented. The major contributions were:

- *Improvements on the partition strategies:* some improvements were made in the way the blocks are generated for each type of characteristic that can be extracted from B models;

- *Improvements on the test data generation:* we integrated the new ProB API into the tool so we can now use it to obtain test data for our test cases. This integration helped to improve the architecture of the tool making it easier to maintain the code base;

- *Integration of logical coverage criteria:* we added support to some logical coverage criteria. Before the doctorate started, only input space partitioning techniques were supported by the approach;

- *Oracle automation:* during the period of the doctorate we also automated the process of oracle evaluation using the ProB API. Previously, the process to obtain oracle data for the test cases was performed manually;

- *Generation of test scripts:* the work of [Souza Neto, 2015] contributed with a generator of test scripts for the tool. The scripts are only partially implemented but are a considered big improvement when compared with the test case specifications that were generated by the initial versions of the tool;

- *Usability improvements:* after case studies were performed we obtained feedback about the usability of the tool and improved it.

More details about these new contributions and other improvements that are still under development are presented in the next chapter.

Chapter 6

Recent contributions and improvements

In this chapter, the most recent contributions to the BETA approach and tool are presented. These contributions are updates, improvements and new additions to the approach presented in Chapter 5. Among some minor improvements, the major additions to the approach were: the support of logical coverage criteria, automatic oracle evaluation, automatic preamble calculation, generation of executable test scripts and partial concretization of test data.

6.1 Support to Logical Coverage Criteria

In the initial versions of the BETA approach, only Input Space Partitioning criteria was supported. Test cases were generated using either Equivalent Classes or Boundary Value Analysis techniques. Even tough these criteria are very common and well-established in the software testing community, they still presented some problems. The combination of partitions using combinatorial criteria to define test requirements resulted in many infeasible test cases (test cases that were impossible to generate test data for).

During the development of the initial versions of the approach and the tool, there was always the intention to make a tool that could support different types of testing techniques and coverage criteria. It would give the users more options to work with when generating test cases. Having a tool that implemented different criteria would also help to study and compare the effectiveness of different criteria. These studies could reveal the situations where one particular criterion could be better than another when generating test cases.

Logical coverage is an obvious next step when we talk about coverage criteria for tests generated from formal models. Formal notations usually define the behavior of a system using logical theories. They use logical expressions to express preconditions, invariants, conditional statements, and other types of constructs of the model. Test cases designed to attend the requirements defined by logical coverage criteria can be used to cover many scenarios expressed in a particular model using logical predicates and clauses.

During our case studies, we analyzed the code coverage achieved by our test cases that used input space partitioning criteria. We noticed that, in some cases, these tests were not enough to cover all the possible execution paths defined in the model. We expect that test cases designed for logical coverage could be more a more suitable choice to achieve a higher level of branch coverage.

Ultimately, there is a demand in the industry for tools that offer support to logical coverage. There are certification standards that require companies to provide logical coverage for the code they produce [FAA, 2001], especially for companies that develop safety-critical software. For example, standards like DO-178B require software to be checked for MC/DC coverage [Hayhurst et al., 2001]. The MC/DC criterion requires that each condition in a decision be shown by execution to independently affect the outcome of the decision. The independence requirement ensures that the effect of each condition is tested relative to the other conditions.

Considering this demand, improvements and adaptations were made in the approach and the tool to add support to MC/DC and other logical coverage criteria. BETA now supports the main logical coverage criteria presented in [Ammann and Offutt, 2010]. The supported criteria are: Predicate Coverage (PC), Clause Coverage (CC), Combinatorial Clause Coverage (CoC) and Active Clause Coverage (ACC). The ACC is equivalent to the MC/DC criterion as explained on Section 3.5.2 from Chapter 3. More details about all the implemented logical coverage criteria are presented on Section 3.5.1 of the same chapter.

We believe that the additional support of such coverage criteria could push forward the tool, making it more appealing to be adopted by the industry. The logical coverage criteria may help to generate more interesting test cases and could help software companies to achieve the coverage requirements that meet the certification standards.

6.2 Automatic Oracle Evaluation and Oracle Strategies

As mentioned in previous sections, the approach now supports different types of oracle evaluation strategies [Souza Neto and Moreira, 2014]. These strategies were first presented in [Li and Offutt, 2014]. They do different types of verifications and can be combined to do stronger or weaker verifications. The strategies are: *Exception Checking*, *Invariant Checking*, *State Variables Checking* and *Return Variables Checking* (more information about these strategies are found in section 5.1.5 of chapter 5).

The oracle evaluation for the test cases also lacked automation in the initial versions of the tool. The oracle implementation was a manual process that required the user to animate the original model and simulate the test case execution. The process consisted in loading the model on an animation tool (ProB was the suggested tool for this job), and executing the operation under test using the test values defined by the test case specification. Once the simulation was done, the user had to observe what was the expected behaviour for that particular test case according to the model.

This process could be automated for most of the scenarios if BETA was integrated with an animation tool like ProB. With that in mind, we decided to automate the oracle evaluation process by integrating ProB into BETA using ProB's Java API that is currently under development.

The integration works as follows:

1. To obtain the expected values for the oracle, first it is necessary to put the model in the state where we intend to execute the test case. To do this, the ProB API provides a method to obtain a trace to a state where a given predicate holds true. So, first of all, it is necessary to define this predicate that specifies the intended state. Since a test case already defines the values of all the state variables, the definition of this predicate is straightforward. The tool will write it as: $var_1 = value_X \wedge var_2 = value_Y \wedge ... \wedge var_N = value_Z$;

2. Once the model is in the intended state to execute the test, we use ProB API methods to execute the operation under test passing test values as parameters, if the particular operation has any parameters;

3. After the operation under test is executed, the model will transition to a new state. Then, depending on the oracle strategy selected, the tool will do one or more of the supported oracle verifications. If *invariant checking* is selected, a method that verifies if the invariant was violated is executed. This method has to be implemented manually by the test engineer. If *state variables checking* is selected, the tool uses methods of the ProB API to query the current state of the model and compares it with the implementation. For *exception checking*, the tool only verifies if the test case raised any exception during its execution.

Unfortunately, there are still some limitations in the ProB side that affect our implementation for the automatic oracle evaluation. The first limitation is that it is currently not possible to obtain return values from operations after their execution using the API. The second limitation is that it is not possible to work with elements from deferred sets when performing animations. So, for example, if we need to pass an element from a deferred set as an operation parameter, we cannot do it using ProB's API. As a workaround, we currently request the user to replace deferred sets by enumerated sets before a model is loaded on BETA (somewhat similarly to the way ProB enumerates deferred sets in its graphical interface).

Another limitation, that is not related to ProB, is the generation of oracles for negative test cases. Currently, it is not possible to create oracles for this type of tests automatically because it is not possible to foresee what is the expected behaviour for a test case that simulates scenarios that are not specified in the model. These test cases most likely break the model's preconditions, and according to the B-Method philosophy, the behaviour of an operation which the precondition was broken is unknown.

6.3 Preamble Calculation

Another aspect that still needed improvements in the approach and the tool was the definition of the test case preambles. Initially, to define preambles, we required the test engineer to use "set" methods to set the state variables with the values that the test case specification required.

After some case studies, the need to elaborate better preambles was identified. In some cases, the test engineer cannot rely on set methods to put the system in the state where the test case has to be executed. "Set" methods are not always available. They could be easily implemented but, in some situations, they can't, as we noticed with the Lua API case study (more information found on Section 7.2 from Chapter 7). In a situation like this, the best way to put the system in the state that is needed is to use the own model operations and formulate a sequence of operation calls that will put the system in that state.

A lot of work might be required to perform this process manually depending on the complexity of the model and the test case. So we decided to use ProB's constraint-based test case generator (also called CBC test case generator) to help with this task. The CBC test case generator receives a list of operations that should be covered by a test case and uses constraint solving techniques to find paths that cover these operations. For this tool, a test case is a sequence of operation calls that begins with the initialization and covers all the operations requested by the test engineer. The test engineer can also set more parameters in the tool, such as a predicate that should hold true at the end of each sequence of operation calls.

The CBC test case generator was integrated into BETA and used to automate the creation of test case preambles as follows:

1. The objective is to elaborate a sequence of operation calls that begins with the initialization and terminates in the state necessary to execute the test. To do this, first BETA creates an auxiliary machine that contains all the operations from the original model, plus an auxiliary operation that will act as a goal for the CBC's algorithm;

2. The CBC's algorithm will then find a path that leads to a state where the auxiliary operation is enabled. The auxiliary operation states in its precondition the state necessary to execute the test case. So, when this operation is enabled, it means that the system has reached the state that we needed;

3. The CBC test case generator then generates a XML file that contains the sequence of operation calls that establishes the preamble for the test case. BETA reads this XML

file and adds the information to its own test case specification.

There are some limitations regarding the complexity of the models that could work using this approach. If the models are too complex, the process can result in a state space explosion and the tool will not be able to calculate the preamble.

This line of work is similar to the work of [Ambert et al., 2002] and [Dick and Faivre, 1993] where the test cases were sequences of operation calls. In our work, we use this idea just for the calculation of the preambles. The test cases themselves are defined first, and we use this technique just to find paths that lead to a suitable state for the test case execution.

6.4 Generation of Test Scripts and Oracle Strategies

Another improvement was made in the last step of the BETA approach: the implementation of the concrete test cases. In the last step, the test engineer had to read the test specification generated by the tool and translate it to a concrete, executable test case using a programming language and test framework of his choice. In the past, this last step had to be performed manually. It required a lot of effort, especially when the test specification defined too many, too complex test cases. With that in mind, we developed a generator of executable test scripts that automates part of the process of translation to concrete test cases [Souza Neto and Moreira, 2014] [Souza Neto, 2015]. These executable test scripts still have to be adapted and complemented with some information before they can be executed.

The test script generator was developed using Rascal [Klint et al., 2009], a domain-specific language for metaprogramming. This language was used because it has features for analysis, transformation and generation of code in a single language, being very helpful in the development of code generators and compilers. Besides, Rascal was developed using Java, the same language used by BETA, which facilitates the integration of both tools.

The generator receives as input a test case specification in XML format (which is generated by BETA). Then it extracts from the XML file all the information that it needs to generate the executable test scripts. The information extracted is then passed to a module that implements a test template for a particular programming language and testing framework. Currently, the test script generator supports the Java and C language, using the JUnit[1] and CuTest[2] frameworks, respectively. This module will format the test code.

The test script generator also supports all the oracle strategies presented on section 5.1.5 from chapter 5. The generated test script still has to be adapted before it can be executed. Since BETA generates code from abstract machines, that uses abstract data structures, the data structures used in the test cases may need to be adapted to use the data structures used in the actual implementation. Also, depending on the chosen oracle strategy, the test engineer might have to fill some test assertions with data.

[1]JUnit Project's website: http://www.junit.org/
[2]CuTest Project's website: http://cutest.sourceforge.net/

Currently, the test script generator only supports the JUnit and CuTest frameworks, but we are implementing it in way that more programming languages and test frameworks can be supported by the tool in the future.

6.5 Test Data Concretization

A very common problem when we talk about Model-Based Testing is the concretization of test data. Since models may represent data in a more abstract way, using abstract data structures, it is usually necessary to adapt these data during the implementation of the concrete test cases. The same problem occurs in the BETA approach. Since it generates test cases based on B-Method's abstract machines, the test data is described in the test case specifications using B-Method's abstract data structures (e.g. functions and relations).

Our proposed solution to this problem uses a B-Method feature called *gluing invariant*. The gluing invariant is a special type of invariant that is used during the refinement of the models. It is used to express the relation between the variables of two different levels of refinement. The gluing invariant is written in the more concrete level and states how some of its variables can be mapped to an abstract variable from the module it refines.

To concretize the test data of its test cases, BETA relies on the gluing invariant to map the abstract test data to the data structures used on the implementation level.

Since BETA test cases are expressed in the form of predicates that can be solved using a constraint solver to obtain test data, we use the same predicates to concretize the test data. Some adaptations in the test predicate are required for this. For each one of the variables we need to concretize, we have to identify the gluing invariant for the variable and add it to test predicate using a conjunction. By doing it, when the test predicate is solved, the constraint solver will give us the values not only for the abstract data described in the formula but also for the concrete variable described on the gluing invariant.

This method still has to be evaluated more thoroughly, exploring possible limitations. Also, it still has to be implemented in the tool. Currently, there is only a separate prototype that automates the process described here.

Chapter 7

Case Studies

This chapter presents all the case studies performed using BETA. The main objective of these case studies was to obtain relevant information to answer the research questions discussed in Section 1.2. For each case study, a brief introduction is made to present its context, followed by the results obtained and the lessons learned from it. Until now, four case studies were performed: the *GDC*, the *FreeRTOS*, the *Lua API*, and the *c4b and b2llvm* case studies. The first two are presented in a single section and are just a review of what had been done before the author's doctorate started. The last two case studies belong to the scope of this doctorate and are presented with more details.

7.1 Previous Case Studies

7.1.1 The GDC Case Study

In this first case study, a model of a metro *"General Door Controler"* system (from now on referred just as GDC) was used to validate the initial testing generation approach proposed by [Souza, 2009].

The GDC system is developed by the *AeS Group*[1], a Brazilian company which specialises in the development of safe-critical systems for the railway market. The GDC system controls the state of the doors in a metro and has operations to open and close them. These operations must obey a series of safety restrictions that take into consideration, for example, the current speed of the train, its position on the platform, and possible emergency signals.

The model used in this case study was specified by [Barbosa, 2010]. It is composed of 19 operations (each operation containing at least one precondition clause), 29 variables, and 46 invariant clauses.

The objective of this case study was to evaluate the original version of the approach. A user, which had no familiarity with the approach or in-depth knowledge about formal methods, was invited to apply the approach to the main machine of the model. It is important

[1]AeS Group website: http://www.grupo-aes.com.br/

to mention that, when this case study was performed, there was no tool to automate the approach yet, so all the test generation process was done manually.

In the end, the case study showed that the approach could be used even by people with no expertise in formal methods. It also showed that, if done manually, the process was very susceptible to errors. If we wanted to facilitate the adoption of the approach, a tool had to be developed to automate it.

Unfortunately, this case study had some limitations related to the scope of the model. Since GDC is a signaling system, all of its variables and parameters were of the boolean type. Because of this, there were some features of the approach, like boundary values analysis, that could not be explored in it.

Even with these limitations, the obtained results were promising. The *AeS Group* was interested in the obtained results, showing particular interest in the possibility of generating negative test cases. The case study also provided information about what should be the next research directions for the project. More details about the GDC case study can be found on [Matos et al., 2010].

7.1.2 The FreeRTOS Case Study

In this case study, a model of the FreeRTOS[2] microkernel was used to evaluate the reviewed approach and the first version of the tool developed to automate it.

FreeRTOS is a free and open source microkernel for real-time systems. It provides a layer of abstraction between an application and the hardware that runs it, making it easier for the application to access hardware resources. This abstraction is implemented using a library of types and functions. The library has no more than two thousand lines of code and is very portable, supporting 17 different architectures. Some of the features provided by FreeRTOS are: task management, communication and synchronization between tasks, memory management, and input/output control.

The model used in this case study was specified by [Galvão, 2010]. The model encompasses FreeRTOS *tasks*, *scheduler*, *message queue* and *mutex* components. The case study focused on the *message queue* component of the specification. This component controls the communication between tasks using messages that are sent to and retrieved from a queue.

Some characteristics of the queue module were interesting for evaluation of features of the approach that could not be explored during the GDC case study. These characteristics are: more variety of data types, modules composed of many machines, and operations with conditional statements.

This case study showed a significant increase in the number of test cases generated after improvements were made in the approach based on the results of the first case study. Some of these improvements were 1) treat guards on conditional statements as characteristics

[2]FreeRTOS website: http://www.freertos.org/

that should be tested; 2) take into consideration characteristics from imported machines; and 3) generate better partitions for typing characteristics.

The case study also revealed problems related to test case infeasibility. The combinations obtained using the implemented combinatorial criteria resulted in many infeasible test cases. Infeasibility occurs in a test case when it combines contradictory blocks. For example, if one block requires $x > 1$ and another block requires $x < 0$ and they are both combined in the same test case, there is no value for x that satisfies the test case requirements. Infeasible test cases were also detected in the first case study, but in the FreeRTOS case study, the number was higher.

During this case study there was also a change in the way invariant clauses are treated when partitioning characteristics into blocks. Previously, characteristics obtained from the invariant were treated the same way as the ones obtained from preconditions and conditional statements when generating negative blocks. After some consideration, it was decided that the approach should not generate negative blocks for invariant characteristics. The negation of an invariant clause results in a test case that requires the system under test to be in an inconsistent state before the test case execution. Since the focus of this work is on unit testing, it was decided that the approach should consider the system to be in a consistent state before a test case is executed. In the end, this decision also helped to reduce the number of infeasible test cases generated by the tool.

Ultimately, this case study showed that the approach still had to be improved to reduce the number of infeasible test cases. This problem could be solved by implementing different coverage criteria or by improving the current combinatorial algorithms to consider contradictory blocks during the combinations. Also, the case study showed some problems related to the usability of the tool and the readability of the generated reports.

7.2 Generating tests for the Lua API

In this case study, BETA was used to generate tests for the Lua[3] programming language API. The tests were generated using a model specified by [Moreira and Ierusalimschy, 2013]. The case study presented here was performed in [Souza Neto, 2015] and provided valuable feedback to improve BETA.

Lua [Ierusalimschy et al., 1996] is a scripting language that was designed to be powerful, fast, lightweight, and embeddable. Its syntax is simple but has powerful data description constructs based on associative arrays and extensible semantics. Lua is a dynamic typed language, which supports many programming paradigms such as object-oriented, functional and data-driven programming.

Nowadays, Lua is a robust and well-established programming language in the market.

[3]http://www.lua.org

It is used in projects such as *Adobe Photoshop Lightroom*[4], the *Ginga*[5] middleware for digital TV, and in games such as *World of Warcraft*[6] and *Angry Birds*[7].

One of the main reasons for Lua's success is its embeddable characteristic, which is made possible by its API. The Lua API is written in C and provides a set of macros that allow a host program to communicate with Lua scripts. All its functions, types, and constants are stored in a header file *lua.h*. The API has functions to execute Lua functions and code snippets, to register C functions to be used by Lua, to manipulate variables, among other things.

The communication between the host program and a Lua script is done using a stack. If the host program wants to execute a function in a Lua script, it has to use an API function to load the script, and then use the API stack to call the function. First, it has to put in the stack the function it wants to call, and then the parameters it requires in the right order (the first parameter must be put in the stack first). Once the function and its respective parameters are in the stack, the host program can call another function in the API that executes the Lua function.

A model of the Lua API was specified by [Moreira and Ierusalimschy, 2013] using the B-Method. The model focuses on the typing characteristics of the Lua language and the consistency of the API's stack. The model was developed in incremental cycles, first modeling the typing system, and after that the state and operations for the API. It is based on the reference manual for Lua 5.2 [Ierusalimschy et al., 2014] and, altogether, is composed of 23 abstract machines.

The objective of this case study was to evaluate BETA with more complex specifications, which would possibly reveal problems and areas of the approach/tool that still needed to be fixed or improved. It was also the first time that the whole test generation process was evaluated, from the generation of abstract test cases to the implementation and execution of concrete test cases. Additionally, this case study could also provide a way to refine the Lua API model and possibly find discrepancies between the model of the API and its implementation.

7.2.1 Results

This case study started with the complete version of the model for the Lua API. Unfortunately, due to limitations of ProB, the scope of the model had to be reduced. Because of the complexity of the original model, mainly in the parts related to the Lua typing system, ProB was not able to load it properly. The way the model specifies the typing of variables is through an extremely large cartesian product that was just too much for ProB to compute. Since BETA uses ProB as a constraint solver, it was not able to generate test cases for the

[4]http://www.adobe.com/products/photoshop-lightroom.html
[5]http://www.ginga.org.br/
[6]http://us.battle.net/wow/en/
[7]https://www.angrybirds.com/

complete model.

Some modifications were made to the model as an attempt to make ProB load it, but it did not work. In the end, as a workaround for this problem, the original model was reduced removing some of the Lua types. The reduced version of the model only deals with the types *nil*, *number* and *boolean* (three of the original eight types), which are simple primitive types that do not require complex models to specify. In the end, 11 of the original 23 abstract machines were kept. We removed all machines related to the five types we were not taking into consideration.

After the model was reduced, ProB was able to load it and BETA could finally generate test case specifications for some of its operations. But it still couldn't generate test case specifications for some operations. The case study detected two problems in the BETA tool that caused this issue:

1. The tool did not support the B-Method's *definitions* clause that was used in many machines of the model;

2. For structured specifications, in some cases, the tool was not considering the whole context of the model when generating the tests. It did not take into consideration modules that were imported by the machine under test.

Once these problems were fixed, BETA was able to generate test case specifications for all operations in the model. The only missing operations were *lua_status*, which is specified as a non-deterministic operation that randomly returns a natural number, and *lua_gettop*, which returns the current value for the state variable *stack_top*. BETA could not generate tests for these two operations because it was not able to identify their input space variables since they are simple return statements and the identification process of input space variables in the BETA approach requires preconditions. In this case, a trivial test has to be implemented. For the other operations, all the test case specifications were generated using *equivalent classes* as a partition strategy and one or more combinatorial criteria.

After the test case specifications were generated, they were implemented as concrete executable test cases using the *Check*[8] framework, which is a unit testing framework for C. At this point, differences between the model and the implementation caused by the model-based testing gap started to appear. The tests generated from the abstract model made use of variables that were not present in the actual implementation of the API. Thus, some inspection had to be made to find the relation between the variables in the model and the variables in the implementation.

Another issue found during the implementation of the concrete tests was that some of the variables of the API could not be directly modified (using a "set" function for example). Because of this, if a test required the API to be in a particular state, it was necessary to

[8]http://check.sourceforge.net/

use functions of the API to carry the system manually to the state where the test should be executed.

Once these problems were solved, the concrete tests were implemented. In the end, only one test case specification for the operation *lua_setglobal* could not be implemented. It was impossible to implement the concrete tests for this operation because the implementation did not correspond to what was specified in the model.

As a remark, it is important to mention that negative tests generated by BETA were not considered during this case study. The documentation for the Lua API solely relies on the concept of preconditions and does not describe what is the expected behavior for inputs that violate the these preconditions, so it was impossible to determine the oracle for negative test cases. For this reason, we decided to only use the positive test cases.

To evaluate the quality of the test cases generated for the Lua API, we decided to measure the extent of the coverage they provided for the code base. To do this, we used two of the most common techniques in the industry to measure coverage: *line* and *branch* coverage.

Tables 7.1 and 7.2 present an overview of the coverage obtained by the tests generated for each API function that was tested. They present respectively the numbers for line and branch coverage. Table 7.1 shows the number of lines for each function and the number of lines covered by the tests generated for each combinatorial criterion. Table 7.2 shows the number of branches for each function and the number of branches covered by the tests generated for each combinatorial criterion.

For us the most interesting results were the ones revealed by the branch coverage analysis. The input space partitioning criteria used in this case study could not cover many of the branches present in the API's code. These results were expected and this problem was one of our motivations to integrate logical coverage criteria in the tool. Logical coverage criteria are more suitable to generate tests that can explore the many branches (or decision points) inside a program. This case study used only input space partitioning criteria because by the time it was performed the logical coverage was not part of the tool. We are planning to do a revision Lua API case study in the future, generating tests using logical coverage and comparing the new results with the ones presented here.

7.2.2 Final Remarks and Conclusions

In this case study BETA was used to generate tests for the Lua programming language API. The tests were generated based on a model of the API specified using the B-Method.

During this case study, some problems were identified in the BETA tool. The most serious problems, which made BETA not generate test case specifications for the model, were caused by the lack of support of the B-Method's *definitions* clause, and an incomplete support of structured models. These problems were fixed and now BETA supports the use of definitions and all model structuring mechanisms of the B-Method.

Table 7.1: Code coverage for the functions in the Lua API

Function	Lines	All Combinations	Each Choice	Pairwise
lua_checkstack	10	6 (100%)	5 (50%)	6 (60%)
lua_copy	3	3 (100%)	3 (100%)	3 (100%)
lua_insert	6	6 (100%)	6 (100%)	6 (100%)
lua_pushboolean	3	3 (100%)	3 (100%)	3 (100%)
lua_pushinteger	3	3 (100%)	3 (100%)	3 (100%)
lua_pushnil	3	3 (100%)	3 (100%)	3 (100%)
lua_pushnumber	3	3 (100%)	3 (100%)	3 (100%)
lua_pushvalue	3	3 (100%)	3 (100%)	3 (100%)
lua_remove	5	5 (100%)	5 (100%)	5 (100%)
lua_replace	4	4 (100%)	4 (100%)	4 (100%)
lua_settop	10	9 (90%)	4 (40%)	9 (90%)
lua_arith	13	12 (92.3%)	12 (92.3%)	9 (69.2%)
lua_absindex	1	1 (100%)	1 (100%)	1 (100%)
lua_compare	9	3 (66.6%)	0 (0%)	0 (0%)
lua_rawequal	4	4 (100%)	4 (100%)	4 (100%)
lua_toboolean	3	3 (100%)	3 (100%)	3 (100%)
lua_type	2	2 (100%)	2 (100%)	2 (100%)

There were also some limitations identified in ProB as a constraint solver to generate the test data. The original model was reduced as a workaround for this problem. Recently there were some significant improvements implemented on ProB's kernel that may have solved part of this problem. More experiments are necessary to see how it behaves when using the complete version of the model, but since the model is known to be quite complex, we don't expected to have this problem completely solved.

This case study also helped to identify future directions for the project. Such as:

– *Implementation of a complete preamble for the test cases:* as mentioned in the previous section, some of the variables in the Lua API could not be directly modified. If a test case required one of these variables to have a particular value, it was necessary to use the API functions to make the variable assume the desired value. Currently, BETA assumes that it is possible to modify state variables directly (using a "set" function, for example). This case study showed that it is not always possible to do such a thing. So, in some cases, it is necessary to provide a complete preamble for the test case. The preamble would have to state the complete sequence of function calls that will carry the system to the state that the test case requires. This feature is already under development and will use ProB's constraint-based testing capabilities to find out what

Table 7.2: Branch coverage for the functions in the Lua API

Function	Branches	All Combinations	Each Choice	Pairwise
lua_checkstack	8	4 (50%)	3 (37.5%)	4 (50%)
lua_copy	1	1 (100%)	1 (100%)	1 (100%)
lua_insert	4	3 (75%)	3 (75%)	3 (75%)
lua_pushboolean	2	1 (50%)	1 (50%)	1 (50%)
lua_pushinteger	2	1 (50%)	1 (50%)	1 (50%)
lua_pushnil	2	1 (50%)	1 (50%)	1 (50%)
lua_pushnumber	2	1 (50%)	1 (50%)	1 (50%)
lua_pushvalue	2	1 (50%)	1 (50%)	1 (50%)
lua_remove	4	3 (75%)	3 (75%)	3 (75%)
lua_replace	2	1 (50%)	1 (50%)	1 (50%)
lua_settop	8	5 (62.5%)	2 (25%)	5 (62.5%)
lua_arith	12	8 (66.6%)	5 (41.6%)	5 (41.6%)
lua_absindex	2	1 (50%)	1 (50%)	1 (50%)
lua_compare	8	3 (37.5%)	0 (0%)	0 (0%)
lua_rawequal	4	3 (75%)	1 (25%)	1 (25%)
lua_toboolean	4	3 (75%)	3 (75%)	3 (75%)
lua_type	2	1 (50%)	1 (50%)	1 (50%)

is the required preamble for a test case.

– *Test data refinement:* some of the test case specifications in this case study could not be translated directly into concrete test cases. This is a known issue of model-based testing approaches. This problem occurred because the tests were generated based on a abstract model that expresses data differently than the actual implementation. To make the process of implementing concrete test cases easier, it is necessary to have a mechanism that facilitates the translation of abstract test data (generated from the abstract model) to concrete test data (that will be used in the executable test cases).

– *Support to different coverage criteria*: even though most of the API functions obtained 100% of code coverage, some of them still had a low coverage percentage. Some of these functions were not completely covered because of branches that the current tests generated by BETA – using equivalent classes and boundary value analysis – couldn't reach. Adding support to logical coverage criteria can help with this problem and increase the code coverage in these situations.

Ultimately, the case study also helped to improve the model of the API, helping to find discrepancies between the model and the implementation.

7.3 Testing two code generation tools: C4B and b2llvm

In this case study [Moreira et al., 2015], BETA was used to test two code generation tools for the B-Method: *C4B* and *b2llvm* [Déharbe and Medeiros Jr., 2013].

C4B is a code generation tool that is distributed and integrated with AtelierB 4.1. It automatically generates C code based on B implementations. The input to *C4B* is a specification written using the B0 notation.

The other tool tested during this case study was b2llvm, a compiler for B implementations that generates LLVM code. LLVM is an open-source compiler infrastructure used by many compiling toolchains. It has a complete collection of compiler and related binary programs and provides an intermediate assembly language upon which techniques such as optimization, static analysis, code generation and debugging may be applied.

The input to *b2llvm* is also a B implementation written using the B0 notation. When this case study was performed, the tool was under development and did not support the entire B0 notation. The input to the tool is given as XML-formatted files representing the B implementation, and the output files it produces are in LLVM's intermediate representation, also called *LLVM IR*. The XML input files are generated by AtelierB.

When designing tests for a code generation tool, the test engineer will try to find answers for the following two questions:

1. *Is the tool capable of generating code for the wide range of inputs it can receive?*

The input for a code generation tool is usually a complex artifact such as a model or even another program. It means that the variety of inputs the tool can receive possibly tends to infinity. With that in mind, the test engineer has to design a good set of input artifacts that can provide reasonable input coverage for the tool.

2. *Is the code generated by the tool actually correct according to the source artifact?*

Another aspect that has to be tested is the translation of the input artifacts. For example, if the code generation tool that is being tested generates code based on a model, the test engineer has to design tests that check if the code produced by the tool implements what was specified in the model. This part requires knowledge of the semantics of the input artifacts and how they should be translated in the output format.

In this case study, BETA was used to assist the testing process for this second aspect. The tests generated using BETA were used to verify if the code produced by the code generation tools behaved as specified by the respective input models. Figure 7.1 shows the testing strategy for this case study.

The testing strategy is divided in two levels. For the first level, a set of models was selected to test the first aspect (which answers the first question). The criterion used to select the models for this part was to choose a set of models that cover a high percentage

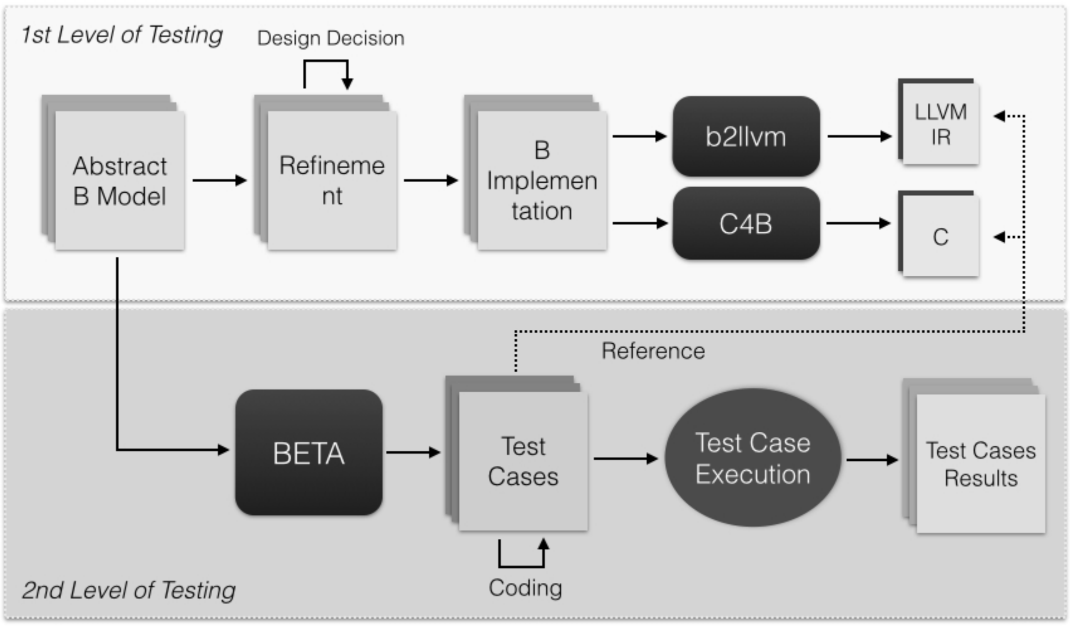


Figure 7.1: Testing Strategy for C4B and b2llvm

of the B0 grammar. Once the models were selected, they were given as inputs to C4B and b2llvm to generate C and LLVM code. The quality of the coverage provided for the first level is related to the rigor of the employed coverage criteria. Since grammar-based testing is not the focus of our work, we won't get into further details. More information about this can be found on the case study paper [Moreira et al., 2015].

In the second level, which is directly related to BETA, the respective abstract machines for the B implementations were used to generate tests cases for the automatically generated C and LLVM code. These test cases were used to verify the conformance between the generated code and the abstract model. The test cases were generated in the form of BETA test drivers. These drivers partially implement executable test cases on a target programming language, which for this case study were written in C, since both code generators tested in this case study generate APIs in C to allow the integration of the generated code with other programs.

Because of the model-based testing gap mentioned before, the test drivers generated by BETA still had to be adapted before they could be executed. These adaptations were also related to the encoding of test case data.

After these adaptations were made, the tests had to be linked with the C or LLVM code to be executed. The testing code and the generated code were compiled, generating an executable program. This program was then run to execute the tests. After the execution, the tests results were evaluated to verify if the code under test was in conformance with the abstract machine. If the generated code behaved as specified by the abstract model, it was expected that all tests should pass. A test could fail for one of the two reasons: 1) a problem in the code generation tool that caused a wrong translation, or 2) some mistake was made during the refinement process and the B implementation was not equivalent to its respective abstract machine.

This case study also evaluated the whole test case generation process, from generation of abstract test cases to implementation and execution of the concrete test cases. It was particularly important to evaluate the capability of BETA to complement the B-Method as a model-based testing tool that uses unit tests to verify the conformance between abstract models and the actual implementation. It was directly related to the first research question presented in Section 1.2 and it helped us to reason about it.

7.3.1 Results

The same set of B models was used to test both code generation tools. Table 7.3 presents some information on these models, the tests, and the obtained results, such as the number of lines of each abstract machine, the number of operations, the number of test cases generated by BETA and the number of tests that passed for b2llvm and C4B generated code.

Table 7.3: Machines used in the case study

N.	Machine	Lines	Operations	Test Cases	b2llvm	C4B
1	Counter	51	4	10	10	10
2	Swap	18	3	3	3	3
3	Calculator	48	6	10	10	10
4	Wd	27	3	5	4	4
5	Prime	10	1	4	4	4
6	Division	12	1	2	2	2
7	Team	36	2	3	3	3
8	BubbleSort	35	1	2	2	2
9	TicTacToe	67	3	13	13	13
10	Fifo	22	2	2	0*	2
11	Calendar	40	1	2	0*	2
12	ATM	28	3	3	0*	3
13	Timetracer	47	6	7	0*	4

This set of machines was chosen in a way that a reasonable part of the B0 notation was covered. An analysis using *Terminal Symbol Coverage* (TSC) [Ammann and Offutt, 2010] was performed to verify how much of the B0 grammar was being covered by them. The *LGen* tool [Moreira et al., 2013] was used as an auxiliary tool to compute the terminal symbols for the part of the B0 grammar corresponding to the modeling of operations.

Overall, the process of generating the test drivers with BETA, adapting the code, and executing it, was done in a few minutes for each one of the tested operations. The overall effort for all the models was approximately one day of work.

The tests for the machines 1 to 9 had the same results for both C4B and b2llvm. One of the tests for the *Wd* machine failed for both tools. At first, one might think it was a problem in the code generation tools but, after further inspection, the problem was found in the refinement process. The B implementation for the *Wd* machine was not properly validated during its refinement and, because of that, the generated code was wrong according to the abstract model. This refinement problem was not noticed until the tests were executed. It was an interesting result for BETA. It was capable of identifying problems that occurred in the refinement process, and not just in the translation to source code. Since it is a problem that happens with some frequency when developing with the B-Method, due to the difficulties of formal verification, this is another useful application for the approach.

For the machines number 10 to 13, b2llvm was not able to generate code because it does not support some of the B0 constructs used on these machines. Because of this, these tests were not performed for b2llvm. In this situation, the tests created by BETA can be used to guide the implementation of these missing features in b2llvm, similarly to a test-driven development approach. In contrast, C4B was able to generate code for these machines. The tests for the machines *Fifo*, *Calendar* and *ATM* have passed, but three tests for the *Timetracer* machine have failed. After further analyzes, the fault was found in the the generated C code. The B implementation for *Timetracer* imports other B modules, so it is expected that the C code generated from it also calls other correspondent C modules. C4B was capable of generating code for all the modules of *Timetracer* but it did not import them where they were needed. That is why these three tests failed. This problem was reported to *Clearsy*, the company that develops AtelierB and C4B.

7.3.2 Final Remarks and Conclusions

In this case study, BETA was used to support the testing process of b2llvm and C4B, two code generation tools for the B-Method. LLVM and C programs generated by b2llvm and C4B were verified using tests that certified their conformance to their respective B specification.

As a remark, it is important to mention that even though BETA is capable of generating positive and negative test cases, the negative ones were not considered in this case study. That was the chosen approach because the intention was to verify if the code produced behaved as foreseen by the input model. Negative tests cases verify how the implementation behaves in situations that were not foreseen by the model. Since C4B and b2llvm directly translate the information in the model into executable code, it would not be fair to expect it to behave "properly" (according to what was expected from the software) in unexpected scenarios, since we don't have a description on what is the expected behavior for this scenarios.

This case study was important to evaluate how BETA performed as a model-based testing tool that can complement the B-Method development process. The B-Method has a lack of formality in the code generation step and in this case study we could assess the effectiveness of the tests generated by BETA when it comes to identifying faults inserted during the code generation process.

The case study showed that the tests were able to identify problems caused by the implementation of the code generation tools. One example of this was the problem related to the use imports in the code generated by C4B. But not only this, it was also able to identify problems caused in other parts of the B-Method development process, such as faults made in the refinement process.

In the end, the case study helped us to be more confident about the answers to the research questions presented in the first chapter. We concluded that BETA could in fact be used as complement for the B-Method, partly reducing the verification gap caused by the lack of formality in the code generation process (*Research Question 1*). It showed us how the features of the tool attended some of the requirements to achieve this goal, and what were the features that still had to be implemented or improved (*Research Question 2*). Ultimately, it also showed the effectiveness of the software testing techniques used by BETA in detecting problems in the context of the B-Method development process (*Research Question 3*).

Chapter 8

Final Discussions and Next Activities

In this thesis proposal, we proposed an approach that complements the B-Method with model-based testing. As explained in the first chapter, the B-Method has a verification gap in the code generation process. In the B-Method, the code generation step can be performed either manually or automatically, using code generation tools. If done manually, the code generated is clearly not verified. This problem could be addressed if the code were generated by code generators that were formally verified, but that is not the case. There are no code generators for the B-Method that are verified and proven to generate correct code. The proposed approach relies on model-based testing techniques to verify the conformance between the model and the implementation's code, trying to lessen the problem caused by the lack of formality in the code generation process.

As mentioned in the first chapter, formal methods alone cannot guarantee that a software is bug free. Testing is still necessary to complement the verification process. BETA provides a cheap way to generate tests from these B models, taking advantage of effort done in the specification phase, since it relies on the information present in these models to generate the test cases.

The approach can also be used in scenarios where the formal development process is not followed rigourously. Since formal methods such as the B-Method are quite expensive and time consuming, they might be used only as modeling tool in some projects. In these scenarios the code is not formally derived from the model but is produced manually instead. BETA can be used to check the conformance between the initial model and the manually implemented code.

Our test generation approach is tool supported and uses testing criteria that were validated through time by the software testing community. The tool also supports testing criteria required by certification standards for safety-critical systems such as MC/DC and other types of logical coverage.

The tool is already implemented and has been evaluated through several case studies (see Chapter 7). It is free and open source and is available for download and evaluation on http://www.beta-tool.info.

So far, the work developed during this project resulted in three publications:

– In [Matos and Moreira, 2012] we presented the first relevant results obtained by our research. We described our test generation approach and how it was evaluated through a case study using a model for the FreeRTOS libraries. Also, in this paper we released the first version of the tool to the public;

– In [Matos and Moreira, 2013] we presented the tool in its first tool workshop. This version of the tool was an improvement made based on the feedback obtained after the first publication. The tool also presented new features such as the first version of the module to generate executable test scripts and improved test case specifications in HTML and XML format;

– In [Moreira et al., 2015] we presented a case study and a test strategy that used BETA to evaluate AtelierB's C4B and B2LLVM, two code generation tools for the B-Method. Using a set of test models, BETA was employed to check the conformance between the models and the code generated by both code generation tools. At the end of the case study, BETA was able to identify problems in both code generation tools. This case study was also important to evaluate the effectiveness of BETA as an approach to complement the B-Method using unit tests. The tests were able to identify problems caused not only in the code generation process, but also in other steps of the process such as refinements.

Besides these three publications, there is also the master's work of [Souza Neto, 2015] [Souza Neto and Moreira, 2014] that is part of the BETA project as a whole. In this master's work a case study was performed to evaluate BETA while generating tests for the Lua API implementation. This case study was the first to evaluate the BETA approach as a whole, from the generation of test case specifications to the implementation and execution of concrete test cases. It provided valuable feedback and pointed out some interesting aspects that could be improved in both the tool and the approach.

8.1 Previous Activities

Here we present a list of the previous activities carried out during this doctorate.

– *Improvements in the partition strategies:* we improved the partition strategies changing the way blocks are generated for some characteristics extracted from the model. This strategies were documented in detail and are publicly available in the tool's website;

– *Research internship:* during the period from May 2014 to March 2015 the author of this thesis was in a research internship at *Heinrich-Heine Universität* where he worked with the STUPS group on integrations between ProB and BETA. During this internship he also worked on the implementation of logical coverage criteria;

- *Support to logical coverage:* we researched and implemented logical coverage criteria for the approach and the tool. BETA now supports four different types of logical coverage criteria besides the input space partitioning criteria that was already implemented;

- *ProB API integration:* we integrated BETA with the new ProB API. In the best all interactions with ProB were made using the CLI interface. Now we use the Java API to perform most of the necessary interactions with ProB;

- *Oracle automation:* we automated the process of oracle evaluation using the ProB API. Previously, the process to obtain oracle data for the test cases had to be performed manually;

- *Generation of test scripts:* we improved the generation of test cases by adding support to the test script generator developed by [Souza Neto, 2015];

- *New case studies:* we performed two new case studies. One in which we generated tests for the Lua API [Souza Neto, 2015] and another in which we evaluated two code generation tools for the B-Method [Moreira et al., 2015];

- *Preamble calculation:* we researched and started to implement the automatic preamble calculation for BETA test cases;

- *Test data concretization:* we researched a strategy to concretize test case data;

- *Other improvements in the tool:* we also made many minor improvements in the tool regarding usability and configurations for the test case generation process.

8.2 Next Activities

So far we were able to provide some answers to our research questions, but we still need to improve these answers by doing better evaluations of the results obtained in the case studies that were already performed, and also by performing a last experiments (this time focusing on performance). Also, there are still some aspects of the tool and the approach that need improvements, and some missing features that we want to see implemented in the tool.

The preamble calculation has an initial prototype implemented, but this prototype needs deeper testing to be integrated in a tool release. The same thing applies to the test data concretization feature. Currently, it is just a tested theory that still needs to be implemented in the tool. We also need to describe more precisely the theory behind these features so it can be added in the definition of the approach.

There is also a new module that generates executable test scripts that was implemented using *Rascal*. This module was implemented separately in a master's work and still needs to be integrated into the tool.

In summary, the next activities of this project are:

– *Task 1 - Writing papers:* we are allocating some time to write papers about our most recent achievements and contributions;

– *Milestone 1 - Proposal presentation*

– *Task 2 - Formalization of the preamble calculation:* we need to rigorously and precisely explain the theory and the process of preamble calculation so it can be added in the approach description;

– *Task 3 - Implementation of the preamble calculation:* we will further develop the preamble calculation feature, evaluate it through case studies and integrate it in the next release of the tool;

– *Task 4 - Formalization of the test data concretization approach:* we still need to rigorously and precisely describe the theory and the process of test data concretization so it can be added in the approach definition;

– *Task 5 - Automatic test data concretization:* we will implement the test data concretization feature, evaluate it and integrate it in the next release of the tool as well;

– *Task 6 - Final experiments:* we are planning to perform some final experiments with the tool. We need to continue with the work developed in [Souza Neto, 2015] to analyze the quality of the test cases generated by BETA. The work already developed was very important for the evaluation of the tool but we still want to improve it by experimenting with different machines. We also want to perform some final experiments regarding the scalability of the tool. These experiments will act as a benchmark for the tool. We will evaluate all BETA's test case generation features in a large set of models. We have access to a repository[1] with many examples of models and we are planning to use them to check how BETA scales when we consider aspects such as the time required to generate test cases, the number of test cases generated, and other quantifiable aspects;

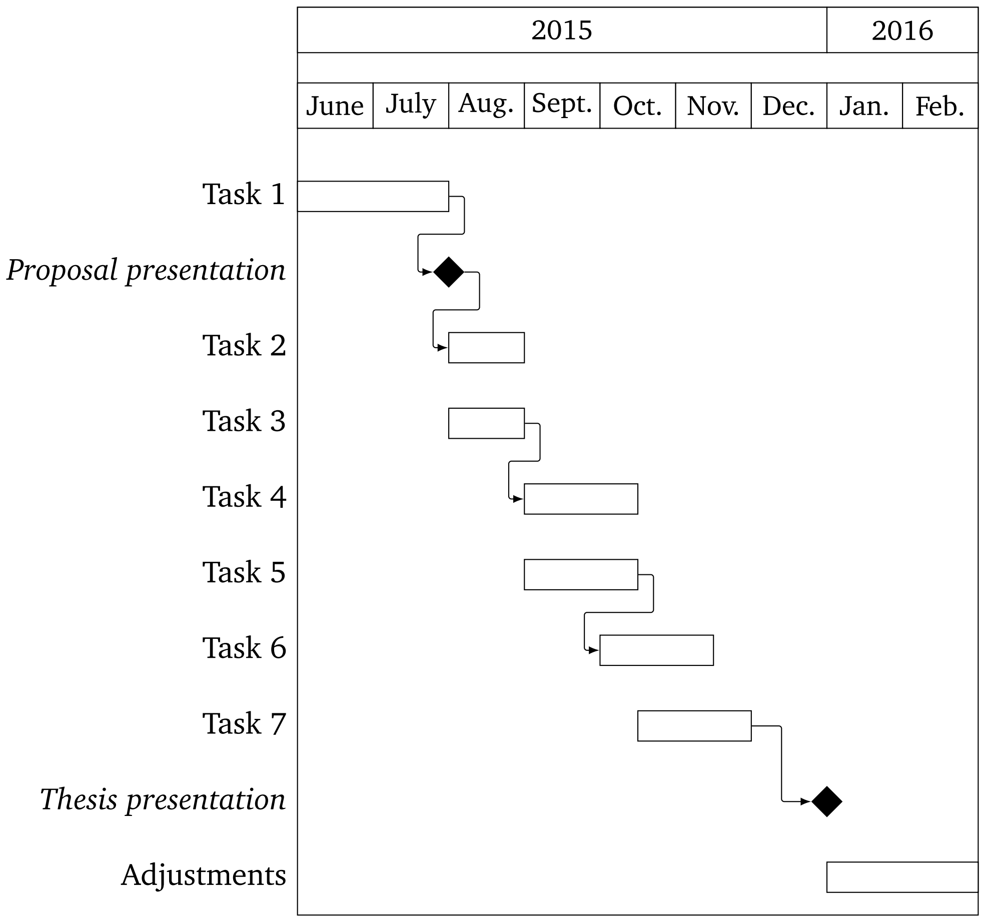
– *Task 7 - Writing of the thesis:* we will write the final version of this thesis;

– *Milestone 2 - Thesis presentation*

[1]The repository of models is provided by the STUPS research group (http://stups.hhu.de/w/STUPS_Group) located at Heinrich-Heine Universität.

The approximate time estimated for each one of these activities is presented in the gantt chart in Figure 8.1. We still have seven months until the end of the doctorate. We are planning to finish the thesis and present it by December. We will leave the rest of the time as a safety time that could be used in case the time planned for any of the activities end up being not enough.

Figure 8.1: Schedule for the next activities.



Bibliography

G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, New Jersey, third edition, 2011.

J. Rushby. Verified software: Theories, tools, experiments. chapter Automated Test Generation and Verified Software, pages 161–172. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-69147-1.

J. R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5. URL http://portal.acm.org/citation.cfm?id=236705.

E. C. B. Matos. BETA: Uma ferramenta para geração de testes de unidade a partir de especificações B. Master's thesis, Natal, 2012.

H. Waeselynck and J. L. Boulanger. The role of testing in the B formal development process. In *Proceedings of Sixth International Symposium on Software Reliability Engineering*, pages 58–67, 1995.

F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. *Proc. of Formal Approaches to Testing of Software, FATES 2002 (workshop of CONCUR'02)*, pages 105–120, 2002.

M. Satpathy, M. Butler, M. Leuschel, and S. Ramesh. Automatic testing from formal specifications. *TAP'07: Proceedings of the 1st international conference on tests and proofs*, pages 95–113, 2007.

A. Gupta and R. Bhatia. Testing functional requirements using B model specifications. *SIGSOFT Softw. Eng. Notes*, 35(2):1–7, 2010.

H. Singh, M. Conrad, S. Sadeghipour, H. Singh, M. Conrad, and S. Sadeghipour. Test Case Design Based on Z and the Classification-Tree Method. *First IEEE International Conference on Formal Engineering Methods*, pages 81–90, 1997.

M. Huaikou and L. Ling. A Test Class Framework for Generating Test Cases from Z Specifications. *Engineering of Complex Computer Systems, IEEE International Conference on*, page 0164, 2000.

E. Mendes, D. S. Silveira, and M. Lencastre. TESTIMONIUM: Um método para geração de casos de teste a partir de regras de negócio expressas em OCL. *IV Brazilian Workshop on Systematic and Automated Software Testing, SAST*, 2010.

S. Burton and H. York. Automated Testing from Z Specifications. Technical report, York, 2000. Report: University of York.

D. Marinov and S. Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. *International Conference on Automated Software Engineering*, 0:22, 2001.

Y. Cheon and G. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *ECOOP 2002 - Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 1789–1901. Springer Berlin / Heidelberg, Berlin, 2002.

N. Amla and P. Ammann. Using Z specifications in category partition testing. In *Computer Assurance, 1992. COMPASS '92. 'Systems Integrity, Software Safety and Process Security: Building the System Right.', Proceedings of the Seventh Annual Conference on*, pages 3–10, 1992.

J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer Berlin Heidelberg, 1993. doi: 10.1007/BFb0024651. URL http://dx.doi.org/10.1007/BFb0024651.

F. M. Souza. Geração de Casos de Teste a partir de Especificações B. Master's thesis, Natal, 2009.

E. C. B. Matos, A. M. Moreira, F. Souza, and R. de S. Coelho. Generating test cases from B specifications: An industrial case study. *Proceedings of 22nd IFIP International Conference on Testing Software and Systems*, 2010.

E. C. B. Matos and A. M. Moreira. BETA: A B Based Testing Approach. In R. Gheyi and D. Naumann, editors, *Formal Methods: Foundations and Applications*, volume 7498 of *Lecture Notes in Computer Science*, pages 51–66. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-33296-8_6. URL http://dx.doi.org/10.1007/978-3-642-33296-8_6.

E. C. B. Matos and A. M. Moreira. Beta: a tool for test case generation based on B specifications. *CBSoft Tools*, 2013.

J. B. Souza Neto and A. M. Moreira. Um estudo sobre geração de testes com BETA: Avaliação e aperfeiçoamento. In *WTDSoft 2014 - IV Workshop de Teses e Dissertações do CBSoft*. 2014.

J. R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, first edition, 2010.

Clearsy. B language reference manual. http://www.tools.clearsy.com/images/0/07/Manrefb_en.pdf, 2011. version 1.8.7.

J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, New Jersey, 2 edition, 1992.

L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, MA, USA, 2002. ISBN 032114306X.

A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, New Jersey, 1 edition, 1997.

P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, first edition, 2010.

K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. A Practical Tutorial on Modified Condition/Decision Coverage. Technical report, 2001.

J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

FAA. Rationale for Accepting Masking MC/DC in Certification Projects. Technical report, 2001.

T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31:676–686, June 1988.

N. Plat and P. G. Larsen. An overview of the ISO/VDM-SL standard. *SIGPLAN Not.*, 27: 76–82, August 1992.

D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. *Proceedings of the 22nd. International Conference on Software Engineering*, 2000.

B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. pages 221–236. 2002. URL http://www.springerlink.com/content/pqug5lfud3gnud43.

F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - A Constraint Solver for B. In J. P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2002.

M. Satpathy, M. Leuschel, and M. Butler. ProTest: An Automatic Test Environment for B Specifications. *Electronic Notes in Theoretical Computer Science*, 111:113–136, January 2005. doi: 10.1016/j.entcs.2004.12.009. URL http://dx.doi.org/10.1016/j.entcs.2004.12.009.

M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer Berlin / Heidelberg, Berlin, 2003.

I. Dinca, F. Ipate, L. Mierla, and A. Stefanescu. Learn and test for event-b âĂŞ a rodin plugin. In *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 361–364. Springer Berlin Heidelberg, 2012.

M. Cristiá, J. Cuenca, and C. Frydman. Coverage criteria for logical specifications. In *Proceedings of the 8th Brazilian Workshop on Systematic and Automated Software Testing (SAST'14)*, pages 11–20, 2014.

J. McDonald, L. Murray, and P. Strooper. Translating Object-Z specifications to object-oriented test oracles. *Asia-Pacific Software Engineering Conference*, 1997.

R. Fletcher and A. Sajeev. A framework for testing object oriented software using formal specifications. In *Reliable Software Technologies - Ada-Europe '96*, volume 1088 of *Lecture Notes in Computer Science*, pages 159–170. Springer Berlin / Heidelberg, Berlin, 1996.

F. Bouquet, F. Dadeau, and B. Legeard. Automated test generation from JML specifications. In *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 428–443. Springer Berlin / Heidelberg, Berlin, 2006.

G. Xu and Z. Yang. JMLAutoTest: A novel automated testing framework based on JML and JUnit. In *Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 1103–1104. Springer Berlin / Heidelberg, Berlin, 2004.

B. Aichernig. Automated black-box testing with abstract VDM oracle. In *Computer Safety, Reliability and Security*, volume 1698 of *Lecture Notes in Computer Science*, pages 688–688. Springer Berlin / Heidelberg, Berlin, 1999.

A. Nadeem and M. J. Ur-Rehman. A framework for automated testing from VDM-SL specifications. *Proceedings of INMIC 2004, 8th IEEE International Multitopic Conference*, pages 428–433, 2004.

A. Nadeem and M. R. Lyu. A framework for inheritance testing from VDM++ specifications. *Pacific Rim International Symposium on Dependable Computing, IEEE*, pages 81–88, 2006.

N. Li and J. Offutt. An empirical analysis of test oracle strategies for model-based testing. In *IEEE Seventh International Conference on Software Testing, Verification and Validation, ICST 2014*, pages 363–372, 2014.

J. B. Souza Neto. Um estudo sobre geração de testes com BETA: Avaliação e aperfeiçoamento. Master's thesis, Natal, 2015.

P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, pages 168–177, 2009.

H. Barbosa. Desenvolvendo um sistema crítico através de formalização de requisitos utilizando o Método B. Bachelor Dissertation, DIMAp/UFRN, 2010.

S. S. L. Galvão. Especificação do micronúcleo FreeRTOS utilizando método B. Master's thesis, Natal, 2010.

A. M. Moreira and R. Ierusalimschy. Modeling the Lua API in B, 2013. Draft.

R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

R. Ierusalimschy, L. H. Figueiredo, and W. Celes. Lua 5.2 Reference Manual. http://www.lua.org/manual/5.2/, 2014.

A. M. Moreira, C. Hentz, D Dehárbe, E. C. B. Matos, J. B. Souza Neto, and V. Medeiros Jr. Verifying Code Generation Tools for the B-Method Using Tests: a Case Study . In *Tests and Proofs, TAP 2015*, Lecture Notes in Computer Science. Springer, 2015.

D. Déharbe and Valério Medeiros Jr. Proposal: Translation of B Implementations to LLVM-IR. In *Brazilian Symposium on Formal Methods (SBMF)*, 2013.

A. M. Moreira, C. Hentz, and V. Ramalho. Application of a Syntax-based Testing Method and Tool to Software Product Lines. In *7th Brazilian Workshop on Systematic and Automated Software Testing*, 2013.