Federal University of Rio Grande do Norte

Department of Informatics and Applied Mathematics (DIMAp)

Doctoral Thesis

Ernesto Cid Brasil de Matos

ENTITLED

BETA: a B Based Testing Approach

Advisor Dr. Anamaria Martins Moreira

Co-Advisor Dr. Michael Leuschel

Natal, RN, Brazil

2016

Ernesto Cid Brasil de Matos

BETA: a B Based Testing Approach

Thesis submitted to Coordenação do Programa de Pós-Graduação em Sistemas e Computação of Universidade Federal do Rio Grande do Norte as a requirement to obtain the degree of Doctor in Computer Science

Advisor: Dr. Anamaria Martins Moreira

Co-Advisor: Dr. Michael Leuschel

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Natal, RN, Brazil
2016

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial Especializada do Centro de Ciências Exatas e da Terra – CCET.

Matos, Ernesto Cid Brasil de.
Beta: a B based testing approach / Ernesto Cid Brasil de Matos. – Natal, RN, 2016.
169 f. : il.

Advisor: Dra. Anamaria Martins Moreira.
Co-advisor: Dr. Michael Leuschel.

Thesis (Ph.D) – Federal University of Rio Grande do Norte. Center of Exact Sciences and Earth. Departament of Informatics and Applied Mathematics. Graduate Program in Systems and Computation.

1. Software engineering – Thesis. 2. Formal methods – Thesis. 3. Software testing – Thesis. 4. B-method – Thesis. 5. Model-based testing – Thesis. I. Moreira, Anamaria Martins. II. Leuschel, Michael. III. Title.

RN/UF/BSE-CCET CDU 004.41

ERNESTO CID BRASIL DE MATOS

BETA: uma Abordagem de Testes Baseada em B

Esta Tese foi julgada adequada para a obtenção do título de doutor em Ciência da Computação e aprovado em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

_____

Profa. Dra. Anamaria Martins Moreira – UFRN
Orientadora

_____

Prof. Dr. Uirá Kulesza – UFRN
Coordenador do Programa

Banca Examinadora

_____

Profa. Dra. Anamaria Martins Moreira – UFRN
Presidente

_____

Prof. Dr. Marcel Vinícius Medeiros Oliveira – UFRN
Examinador

_____

Prof. Dr. Alexandre Cabral Mota – UFPE
Examinador

_____

Prof. Dr. Michael Leuschel – HHU
Examinador

_____

Profa. Dra. Patrícia Duarte de Lima Machado – UFCG
Examinadora

Abril, 2016

This thesis is dedicated to my parents and my sister, for their love and endless support.

Acknowledgements

First, I would like to thank God, the one who created, loved, cared for, and always inspired me. If it weren't for Him, I would have no purpose in my life. I would like to show the deepest gratitude to my parents and sister. They are my safe place, the ones I can always rely on and who provide me unconditional support. I would also like to thank my girlfriend, Rebeca, for always encouraging me to be my best version. I love you. I cannot forget to say how grateful I am for my church, Videira, for being my second family during these years in Natal. Furthermore, I would like to express my gratitude to my advisor, Prof. Anamaria Moreira. She cares about her students like no one I ever met and, without her support, this thesis would not be the same. I would also like to say *"danke sehr"* to Prof. Michael Leuschel for supervising my work during my time in Düsseldorf. The support provided by him and the STUPS team had a great impact on this project. Likewise, I would like to say how I appreciate the life of every professor who had some influence in my education. They helped to shape who I am today. For this, I'm forever grateful. Also, I would like to thank all my ForAll colleagues, especially João, for all the collaborations and insights we shared during this work. Ultimately, I would like to thank CAPES and INES for the financial support.

“Yet I am learning.”
— Michelangelo

Resumo

Sistemas de software estão presentes em grande parte das nossas vidas atualmente e, mais do que nunca, eles requerem um alto nível de confiabilidade. Existem várias técnicas de Verificação e Validação (V&V) de software 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 frequentemente requeridas por orgãos de certificação. Levando isto em consideração, BETA provê uma abordagem de testes baseada em modelos para o Método B, suportada por uma ferramenta, que é capaz de gerar testes de unidade a partir de máquinas abstratas B. Nesta tese de doutorado apresentamos melhorias realizadas em BETA e novos estudos de caso realizados para avaliá-la. Dentre as melhorias, integramos critérios de cobertura lógicos à abordagem, revisamos os critérios de cobertura baseados em espaço de entrada que já eram suportados e aperfeiçoamos as últimas etapas do processo de geração de testes. A abordagem agora suporta a geração automática de dados para os oráculos e preâmbulos para os casos de teste; ela também possui uma funcionalidade para concretização dos dados de teste e um módulo para gerar scripts de teste executáveis automaticamente. Outro objetivo desta tese foi realizar estudos de caso mais complexos utilizando BETA e avaliar a qualidade dos casos de teste que a abordagem produz. Estes estudos de caso foram os primeiros a avaliar o processo de geração de testes por completo, desde a especificação dos casos de teste até a sua implementação e execução. Em nossos últimos experimentos, analisamos a qualidade dos casos de teste gerados por BETA, considerando cada critério de cobertura suportado, utilizando métricas de cobertura de código como cobertura de instruções e ramificações. Também utilizamos testes de mutação para avaliar a capacidade dos casos de teste de detectar faltas na implementação dos modelos. O resultados obtidos foram promissores mostrando que BETA é capaz de detectar faltas introduzidas por programadores ou geradores de código e que a abordagem pode obter bons resultados de cobertura para a implementação de um sistema baseado em modelos B.

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 reliability. There are many software 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. Taking this into consideration, BETA provides a tool-supported, model-based testing approach for the B Method that is capable of generating unit tests from abstract B machines. In this thesis, we present improvements made in the BETA approach and tool, and new cases studies used to evaluate them. Among these improvements, we integrated logical coverage criteria into the approach, reviewed the input space criteria that was already supported, and enhanced the final steps of the test generation process. The approach now has support for automatic generation of oracle data and test case preambles, it has a feature for test data concretization, and a module that automatically generates executable test scripts. Another objective of this thesis was to perform more complex case studies using BETA and assess the quality of the test cases it produces. These case studies were the first to evaluate the test generation process as a whole, from test case design to implementation and execution. In our last experiments, we assessed the quality of the test cases generated by BETA, considering each coverage criteria it supports, using code coverage metrics such as statement and branch coverage. We also used mutation testing to evaluate the ability of the generated test cases to identify faults in the model's implementation. The results obtained were promising, showing that BETA is capable of detecting faults introduced by a programmer or code generation tool and that it can achieve good coverage results for a system's implementation based on a B model.

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

Contents

List of Figures

List of Tables

Glossary

Abstract Test Case a test case generated from an abstract model, that uses abstractions from the said model to define some of its test data.

Concrete Test Case an executable test case implementation.

Logical Clause a predicate that does not contain any of the following logical operators: $\neg$, $\wedge$, $\vee$, $\oplus$, $\Rightarrow$ and $\Leftrightarrow$. This is non-standard according to the terminology for predicate logic where a *clause* is defined as a *disjunction of literals*.

Logical Predicate an expression that evaluates to a boolean value.

Test Suite a set of test cases.

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. One of the most widely known are *Software Testing* techniques, which try to evaluate 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 (or machines) 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 the author's master's dissertation [Matos, 2012], a tool supported approach to generate test cases from B Method abstract machines was developed. The approach is an improvement upon the work developed in [Souza, 2009]. It uses the properties specified in B models and applies well-established software testing techniques to enumerate test requirements for the system implementation.

The main contributions of the work developed in [Matos, 2012] were: 1) a review of the test case generation process proposed by [Souza, 2009], which resulted in significant changes in the said process; 2) the first version of the BETA tool, which automated the process up to the generation of abstract test case specifications; 3) two case studies that evaluated the initial version of the BETA approach and tool.

More specifically, the following research questions were addressed in [Matos, 2012]:

Research Question 1 *Can the approach proposed by [Souza, 2009] be performed by someone who is not familiar with the B Method?*

During the master's period, one of the first things that we did was to perform a case study to evaluate if the approach could be performed by someone who had no previous background in formal methods or the B Method. The author of this thesis, which had a profile that matched our requirements for this case study, used the test case generation approach proposed by [Souza, 2009] to generate tests from a B model. The model used in this experiment was a model of a metro's door controlling system [Barbosa, 2010]. The case study showed us that the approach could be performed by someone who was not familiar with the B Method. It also showed us that the approach was very susceptible to errors when performed by hand. Therefore, it motivated the development of a tool to automate the test case generation process [Matos et al., 2010].

Research Question 2 *How the approach proposed by [Souza, 2009] could be improved?*

We evaluated the test generation approach proposed by [Souza, 2009] through two preliminary case studies, using models of industry systems. These case studies confirmed some problems and deficiencies in the test generation process, and provided the feedback that we needed to propose possible improvements. Some of the problems we tackled were:

– The lack of clarity on how the coverage criteria was used to generate the test cases. We addressed this problem by reformulating the description of the test case generation

approach so it could match the definitions of input space coverage criteria described in [Ammann and Offutt, 2010];

– The lack of support for modular specifications. The first version of the approach required specifications to be contained in a single component. This kind of restriction cannot be imposed on models used in the industry since they usually use components to improve cohesion and maintainability. We improved this aspect by adding support to the different types of modularisation constructs supported by the B Method;

– The first version of the approach did not consider the behaviour of the operation when partitioning the input space of the operation. In the initial proposal, the approach only took into consideration the precondition of the operation and the invariant of the machine when creating partitions. We improved the partition strategies so they could take into consideration conditional statements to create more interesting partitions for the test cases, exercising different execution paths specified by the operation.

Research Question 3 *Can we develop a tool to automate the test generation process?*

The case studies also revealed that, if we wanted to use the approach in an industry setting efficiently, we needed a tool to support it. So we developed a tool that partially automated the test case generation process. The tool was capable of generating test case specifications that could be manually translated into executable test scripts by a test engineer. The test case specifications consisted of simple input data for the test cases and values for the state variables that had to be set before the test case was executed. All this data was generated based on a coverage criterion. The test cases generated still missed some features, such as a proper preamble that contained a sequence of operation calls that could put the system in the desired state for test case execution, and an automatic oracle evaluation process.

In this thesis we continued the work developed in [Matos, 2012]. Our objective with this thesis was to improve both the BETA approach and the tool that supports it, improving the test generation strategies, adding new features to the tool, and performing new case studies to evaluate both the approach and the tool.

Some of the improvements that were made in the approach are: a process to generate oracle data and different strategies for oracle evaluation that can be used to perform weaker or stronger verifications; a strategy to calculate preambles for the test cases, which defines sequences of operation calls that put the system in the desired state for test case execution; and a strategy for test data concretization that translates abstract data generated from B Machines into concrete test data that can be used to implement executable test cases. Furthermore, we improved the strategies for creating partitions using input space partitioning criteria and also added support to logical coverage criteria.

Regarding the tool, we updated it so it could support the automation of the new aforementioned features. We also worked on the integration with ProB, using more features available on its kernel and adding more configuration parameters to the BETA tool. BETA now also integrates with the recently developed ProB Java API[1]. Another feature that was added to the tool was a new module that is capable of generating partial executable test scripts written in Java and C [Souza Neto, 2015]. We also redesigned its user interface.

Ultimately, we performed two new case studies to evaluate the whole test case generation approach and did new experiments to assess the quality of the test cases generated by BETA.

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, which is written using a subset of the B notation 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;

[1]ProB API webpage: http://nightly.cobra.cs.uni-duesseldorf.de/prob2/prob2-handbook/nightly/devel/html/

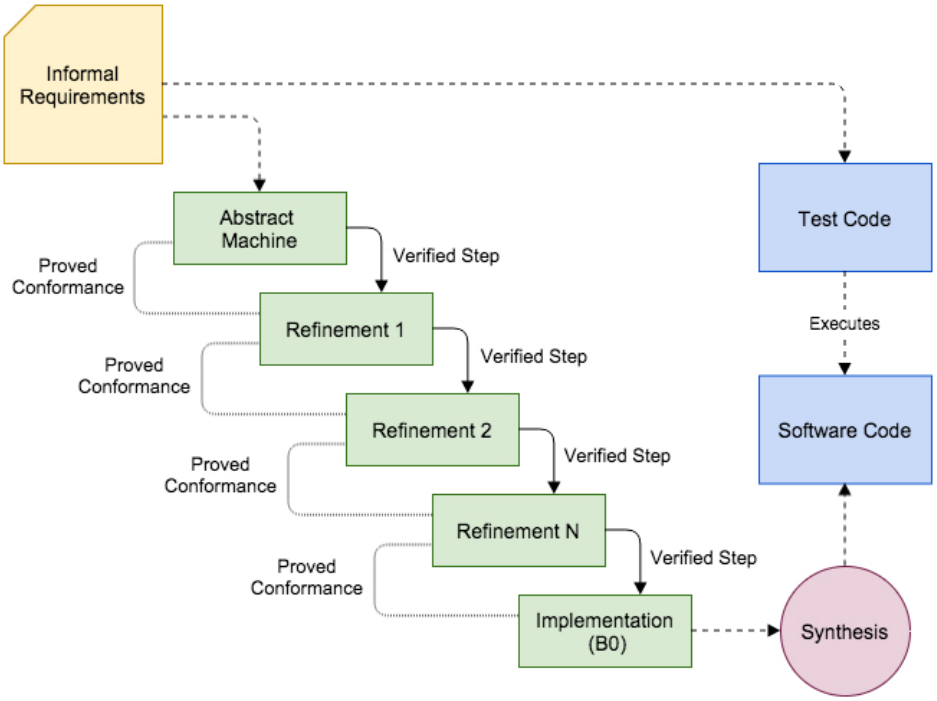

Figure 1.1: The B Method development process and the usual testing procedure. Full line arrows represent formally verified steps and dashed line arrows represent steps that are not formally verified. Dotted lines show relationships between artifacts.

- 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;
- Although refinements are a verified process, it will always require a certain degree of human input, admitting possibilities of human errors during the proofs;
- The formal system underlying the method may be wrong (the problem of validation of the validation mechanism);
- Ultimately, proofs may be faulty.

All these points make a case for the use of software testing to complement the formal development process, and we believe the last three can be addressed by a formal testing approach like BETA.

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 aspects 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 being refactored to improve readability and maintainability. Refactorings are usually made directly into the source code. So, tests generated from the specification can act as a safety net to ensure that the behaviour of the implementation did not change during code refactorings.

– 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 needs 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.

Model-Based Testing (MBT) is a software testing technique to generate (and sometimes execute) tests based on models or specifications of the system. As formal specifications usually describe requirements in a rigorous and unambiguous way, they can be used as basis to create test cases using MBT. The automatic generation of test cases from formal models 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, specially in the B Method's context, we believe, has a deficiency in one or more of the following points:

– 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.

Another point that is not a deficiency but a different view on the subject is that there are a lot of work in the current state of the art that focus on finding problems in the model rather than on the respective implementation.

In this thesis, we build up on a previously proposed [Matos, 2012] 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 check 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.

In [Marinescu et al., 2015] the authors present a recent overview of the state of the art and a taxonomy for the classification of model-based testing tools for requirement-based specification languages. According to their taxonomy, which has six dimensions, BETA can be classified as a tool that:

– Uses a *state-based* or *pre/post* notation as a *modeling notation*;

– Generates test cases based on *artifacts* that model *functional behavior*;

– Relies on *structural model coverage* as a *test selection criteria*;

– Uses *constraint solving* as a *test generation method*;

– Uses *offline* test cases (tests are created and executed in two different steps) as a test execution *technology*;

– Has a *mapping* mechanism that translates *abstract test cases* into *executable test cases*;

1.2 Hypotheses and Research Questions

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

1. The test case generation approach and the tool developed in [Matos, 2012] can still be improved if we add important features such as test script generation capabilities, automatic oracle and preamble calculation, and test data concretization;

2. The proposed test case generation approach and the tool that supports it are flexible enough to allow the integration of other software testing techniques and coverage criteria;

3. The test cases generated by BETA are efficient enough to detect problems introduced during the process of implementation of B models.

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

Research Question 1: *How can we improve the test generation process, mainly the last steps of the approach?*

The last experiments performed in [Matos, 2012] confirmed that both our approach and the tool still needed improvements in the last phases of the test generation process. The tool was only able to generate abstract test cases and still missed features like preamble calculation and automatic generation of oracle data. In this thesis, we worked on fixing these issues.

We added a test script generation module in the tool that is capable of generating partial test scripts written in Java and C. The test scripts still need some adaptations before they can be executed, but they save some of the effort necessary to translate the abstract test cases into concrete test cases. This module was one of the contributions made by [Souza Neto, 2015].

Another issue that we worked on was the automatic generation of oracle data. In the past, we required the test engineer to manually define the oracles for the test cases. The oracle data could be obtained by animating the original model and checking what was the expected behaviour of the system for a particular test case according to the model. In this thesis, we made this process automatic. Thanks to the integration with ProB [Leuschel and Butler, 2003], an animator and model-checker for the B notation, BETA is now capable of animating the models to obtain the oracle data for its test cases automatically. Also on the subject of oracles, we defined and implemented new oracle strategies that can be used to perform different types of verifications during the execution of the test cases [Souza Neto, 2015].

Our experiments also confirmed that, in some cases, we need to calculate preambles for the test cases. We used to require *set* functions in the system's implementation so that we could use these functions to put the system in the state we wanted to before the execution of the test cases. Unfortunately, we cannot always rely on the availability of these functions. Because of this, we needed to calculate preambles for the test cases generated by BETA. The preamble consists of a sequence of operation calls that will put the system in the state that we want to perform the test. In this thesis, we developed a preamble calculation strategy for BETA so that the preambles can be calculated automatically.

Ultimately, we also worked on a test data concretization strategy for the approach. One of the big challenges of model-based testing is the gap between abstract test cases and concrete test cases [Utting and Legeard, 2007]. Usually, abstract test cases use abstract data structures that are different than the ones used by concrete test cases in the implementation level. So, it is necessary to concretize the test cases generated using model-based testing before they can be executed against the system's implementation. To solve this problem, we developed a test data concretization strategy that uses the B method's gluing invariant to find the relation between abstract test data and concrete test data.

Research Question 2: *How the testing criteria supported by the approach can be improved?*

Another aspect that we worked on was improving the software testing criteria supported by BETA. We refined the previously supported criteria and added support to new ones.

Initially, BETA only supported input space partitioning as a strategy to generate test cases. It supported both equivalence classes and boundary value analysis to generate partitions for its test cases. During this Ph.D., we worked on improvements for the partition strategies. We reviewed the old partition strategies and also added new strategies so that more interesting test cases could be generated.

We also added support to logical coverage criteria. Support to logical coverage is important for the approach because there is a demand from certification standards for this kind of criteria, particularly in the field of safety-critical systems. This work was also important to evaluate the flexibility of the approach when it comes to adding new coverage criteria and modifying the ones it already supports.

Research Question 3: *How can we measure the quality of the test cases generated by BETA?*

In this thesis, we also evaluated the quality of the test cases generated by BETA. We performed more in-depth experiments to assess the properties and the effectiveness of the test cases generated using different types of coverage criteria. To evaluate the effectiveness of the test cases we did experiments using mutants. We used mutation tools to generate mutants of implementations derived from B models and checked if the test cases generated by BETA were able to detect the faults introduced by these mutants.

During this Ph.D., we also performed two new case studies that, for the first time, exercised the whole test case generation process, from the generation of test case specifications to implementation and execution of concrete test cases. In the first case study, we generated tests cases for the Lua programming language API using a model developed by [Moreira and Ierusalimschy, 2013]. In the second case study, we used BETA to generate test cases to test the code produced by two code generators for the B Method: b2llvm[Déharbe and

Medeiros Jr., 2013] and C4B[2].

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 and experiments.

The first question we had to answer was how we could improve the test generation process that we already had. The approach and the tool developed in [Matos, 2012] were evaluated through some preliminary case studies. Unfortunately, these case studies were not enough to evaluate the whole test case generation and execution process. These two initial case studies focused only on the generation of test case specifications, missing the implementation and execution of the concrete test cases. Because the test cases were not implemented and executed, it was not possible to assess the quality of the test cases regarding code coverage and capability to find bugs.

With that problem in mind, we designed a new case study to evaluate the whole test case generation process, from the generation of abstract test cases to the implementation and execution of the concrete test cases [Souza Neto and Moreira, 2014]. This case study used a model of the Lua programming language API specified by [Moreira and Ierusalimschy, 2013]. The models of the Lua API were much more complex and challenging for the approach. This case study provided important feedback to improve many aspects of the approach and the tool and helped to identify limitations of our approach. Some of the aspects that were improved after this case study were related to oracle evaluation and the generation of test scripts. It also confirmed the need for better preambles in the test cases generated by BETA.

Regarding the third research question, the Lua API case study was also important to assess the quality of the test cases generated by BETA. Once the test cases were generated, they were subjected to a code coverage analysis. This analysis was important to assess the level of coverage provided by each coverage criterion supported by the approach. It also helped us to identify some execution paths that the tests generated by BETA could not cover.

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 *a posteriori*, based on the documentation of the API, and the tests created were used to check the conformance of the model with the API's code.

Another decision that we made during this Ph.D. was to improve the supported test criteria and add support to logical coverage to both the approach and the tool (addressing second research question). This work helped us to evaluate their flexibility when it comes to adding new criteria and modifying the ones already implemented.

[2]AtelierB website: http://www.atelierb.eu/en/

Once we improved the supported coverage criteria, we decided to perform a second case study where we generated tests for two code generators for the B Method. This case study also helped us to experiment with the capabilities of the approach to detect discrepancies between the models and their respective (automatically generated) source code.

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.

Ultimately, also regarding the third research question, we performed more experiments to evaluate the quality of the test cases generated by BETA. In these last experiments, we measured the statement and branch coverage achieved by test suites generated using different coverage criteria supported by BETA. Besides, we also evaluated the tests using mutation testing. We used mutation tools to generate mutants of implementations derived from B models and verified if the test cases generated by BETA were able to identify the bugs introduced by these mutants.

1.4 Summary of the results

In this thesis, we were able to improve many aspects of the BETA approach and tool. We enhanced old parts of the approach and features of the tool, and also implemented new ones such as the support for logical coverage, the test script generator, the automatic oracle data generation and preamble calculation features, new oracle strategies, and the test data concretization strategy.

We were also able to evaluate both the approach and the tool through more in-depth case studies. The case studies were performed using more complex and challenging models and experimented with aspects that had not been evaluated before. For the first time, we were able to evaluate the approach as a whole from the design of the test cases to their implementation and execution. Besides, we assessed the quality of the test cases generated for the several case studies, measuring aspects such as statement and branch coverage. We also assessed the ability of the test cases to identify bugs using mutation testing.

The results obtained after these experiments were promising. The test cases generated by BETA were capable of identifying interesting problems in the targets of our case studies. The tests also achieved a reasonable percentage of code coverage (95% of the statements and 86.1% of the branches for the highest coverage achieved) and a fair mutation score (79.1% for the highest mutation score achieved). More details about these results are presented in Chapter 6.

In the end, we believe we developed a model-based testing approach that contributes to the current state of the art. The approach is mature enough to be used in other projects, and many of its aspects can be reused by other approaches in different scenarios, or for different

formal notations. The tool developed during this Ph.D., and its source code is also available to the public so that it can be explored, extended and reused by other research projects.

1.5 Thesis Organization

The remainder of this thesis 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 thesis;

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

Chapter 4: Related Work This chapter presents a review of the current state of the art on model-based testing using formal models, and draws some comparisons between our work and the work of other researchers;

Chapter 5: A B Based Testing Approach This chapter presents an overview of our test case generation approach, explaining in details each step involved in the test generation process;

Chapter 6: Case Studies This chapter presents all the case studies and experiments that we have performed to evaluate the approach and the tool;

Chapter 7: Conclusions and Future Work This chapter presents our final dicussions, conclusions and future work.

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.

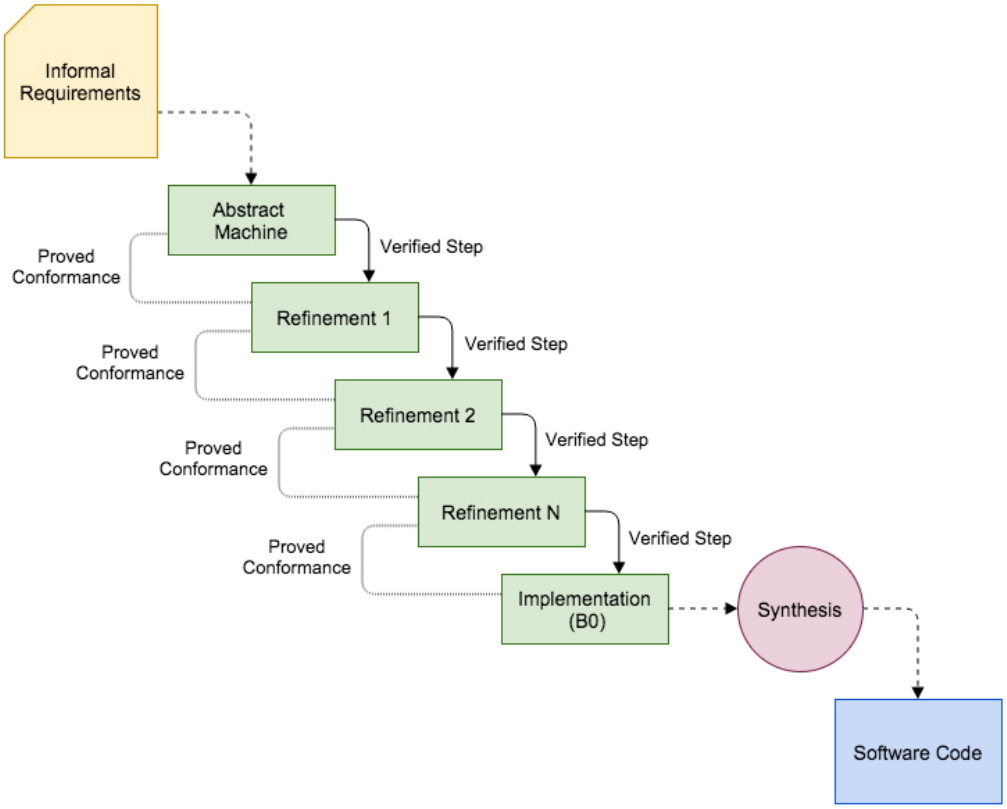

Figure 2.1: B-Method Development Process

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, these 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 consistent with 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 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.

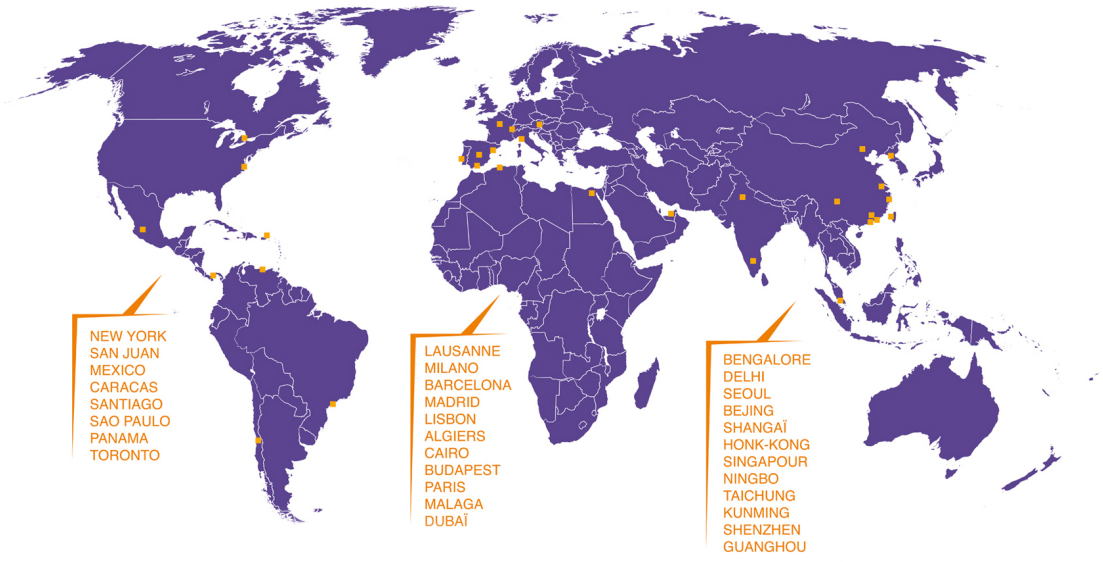

Figure 2.2: Train and metro lines around the world that employed the B-Method. Source: http://www.methode-b.com/en/b-method/

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 easier to learn and use.

The classical B Method and Event-B focus on different aspects of system modelling. While the focus of classical B is on the specification of software components and how they

[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/

work internally, the focus of Event-B is on system-level events. In some cases, projects can use both notations to model different aspects of the system. Event-B development is supported by the 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 modularization mechanisms 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 consistency 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 using specific values.

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

The substitutions are written using the notation $[E/x]P$. This expression states that any free occurrence of 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 $4+1>y$.

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

Simple Attribution Substitution

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

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.

Multiple Attribution Substitution

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

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.

Conditional Substitution

$$[IF\ \ E\ \ THEN\ \ S\ \ ELSE\ \ T\ \ END]P \Leftrightarrow (E \Rightarrow [S]P) \wedge (\neg E \Rightarrow [T]P) \tag{2.3}$$

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.

ANY Substitution

$$[ANY\ \ X\ \ WHERE\ \ P\ \ THEN\ \ S\ \ END]R \Leftrightarrow \forall x.(P \Rightarrow [S]R) \tag{2.4}$$

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 a non-deterministic substitution.

Parallel Substitution

$$S1 \parallel S2 \tag{2.5}$$

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 simultaneously. We do not present the exact substitution in the notation above because $S1$ and $S2$ could be replaced by any of the substitutions supported by the B Method.

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
$\mid\!-\!>$	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 classroom and their grades. The features of the system are:

- It allows students to be registered in the classroom;
- 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

```
1  MACHINE Classroom
2
3  SETS
4   STUDENT; result = {pass, final_exam, fail}
5
6  PROPERTIES
7   card(STUDENT) = 15
8
9  VARIABLES
10  students,
11  grades,
12  has_taken_lab_classes
13
14 INVARIANT
15  students <: STUDENT &
16  grades : students +-> 0..5 &
17  has_taken_lab_classes : students +-> BOOL
18
19 INITIALISATION
20  grades := {} ||
21  students := {} ||
22  has_taken_lab_classes := {}
23
24 OPERATIONS
25   add_student(student) =
26     PRE
27       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 *Classroom* (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 *Classroom* 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 *Classroom* 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 with the *empty set*.

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*[3]. 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 part of 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 *Classroom* 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 specify the conditions under which the operations can be applied. In their body they just make simple

[3]This is a crucial concept, especially when dealing with animations and test case generation.

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 *Classroom* machine one must prove the following:

$$\exists(students, grades, has\_taken\_lab\_classes). \tag{2.7}$$
$$(students \subseteq STUDENT \wedge \tag{2.8}$$
$$grades \in students \nrightarrow 0..5 \wedge \tag{2.9}$$
$$has\_taken\_lab\_classes \in students \nrightarrow BOOL) \tag{2.10}$$

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 describing the initialization, and I is the machine's invariant.

For the *Classroom* example, one has to prove the following:

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

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

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

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

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 *Classroom* example the following proof obligation would be generated:

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

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

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

Since this implication is, in fact, true, the proof obligation for the operation is easily proved.

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*[4], *ProB*[5], and *Rodin*[6].

2.6.1 AtelierB

AtelierB is a tool for specification and verification of B models. The tool is developed by Clearsy[7], a company specialized 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
- code generators that can translate B implementations into programming languages code

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. It also has a more complete and robust paid version.

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.

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

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

[6]Rodin's website: http://www.event-b.org/

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

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 though 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 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 technique 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 the definition of the test requirements, which can be produced using some special criterion. The test cases are then designed according to the 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, which, usually, is not a trivial task. 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.

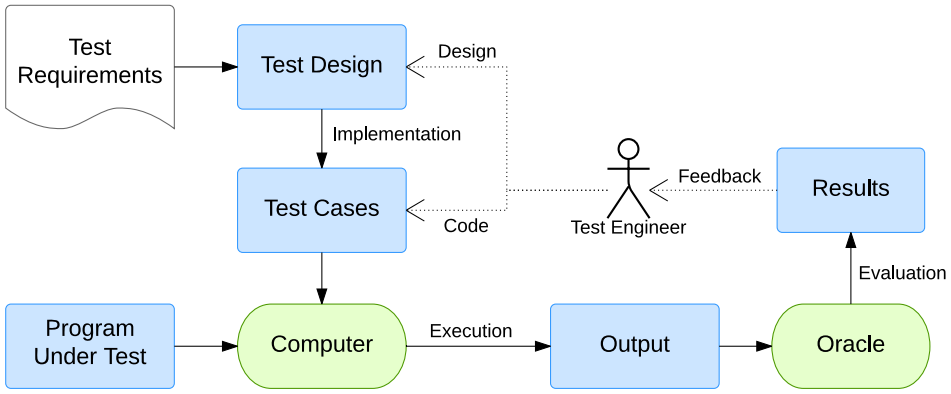

Figure 3.1: Usual software testing workflow.

3.1 Testing Levels

The software development process is composed by different phases, such as defining the requirements and documenting them, planning and designing the software architecture and actual coding of the software. Faults (or bugs) 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

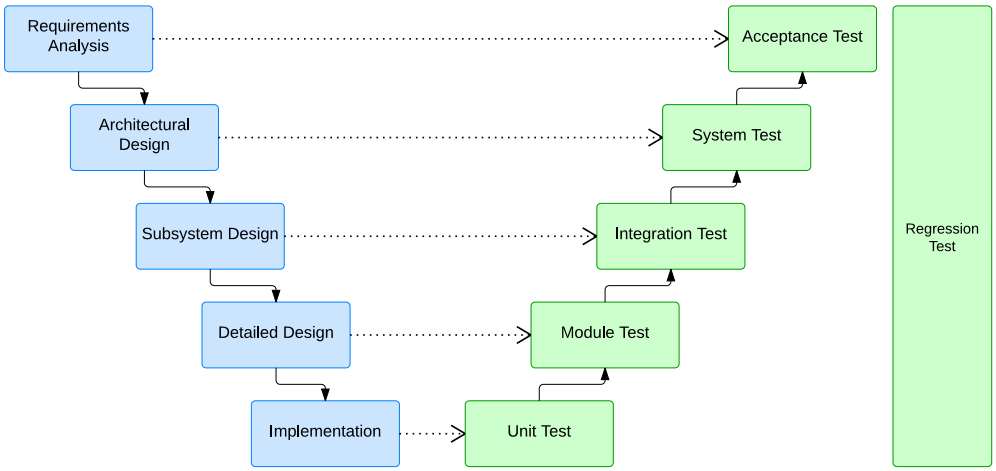

Figure 3.2: Levels of Testing (the V model).

automatically, by tools that can simulate the user's actions.

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 their 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 flowcharts 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. Even though they use abstractions as a source to define the test cases, they can be used in different test levels, from high-level system testing to low-level unit testing. 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

```
1  int vector[] = new int[5];
2
3  for (int i = 0; i <= 5; i++) {
4    System.out.println(vector[i]);
5  }
```

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 (or Test Case Data): 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. 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. These definitions will be used later in this document to speak about test cases that respect or disrespect the preconditions of the software under test.

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

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 to 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 possible 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:

$$\textit{number of requirements covered by T / size of TR}$$

There might be cases where some test requirements cannot be satisfied. It means that it is impossible to create a test case that satisfies 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*, 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 of 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 are not 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 these blocks represents a subset of the function's input domain and the elements in these subsets are considered equivalent when testing the function in respect to the 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 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. 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].

Combination 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 require the following set of test cases (TCs), which, in this case, is equal to the set of 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 the 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 combination 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 satisfy:

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.

Ultimately, it is important to mention that there is a subsumption relation between the combination criteria we just presented. A coverage criterion C_1 subsumes a criterion C_2 if, and only if, every test set that satisfies criterion C_1 also satisfies C_2. In this case, we can say that *All-Combinations* subsumes *Parwise*, which in turn subsumes *Each-Choice*. With that in mind, the test engineer can choose the coverage criterion that is more suitable for the environment he is working on.

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 *atomic formulas* (which we refer to as *clauses*). A *predicate* is an expression that evaluates to a boolean value. An example of a 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 ($\Leftrightarrow$).

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 $p(x)$.

It is important to emphasize that the term *clause* is used here *differently* from the standard terminology for predicate logic. What we call a *clause* here, is in fact, a *positive literal* according to predicate logic. We decided to keep using the term clause in a non-standard way because this is how [Ammann and Offutt, 2010] uses the term and because our implementations of logical coverage criteria are based on this reference.

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 relate 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 set of clauses in each predicate in P, that is, $C = \bigcup_{p \in P} C_p$.

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*.

For the predicate presented previously, to satisfy CoC, we need all four tests shown in the table above.

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 $c_j \in p$, for $j \neq i$, 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, with $j \neq i$, 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*.

For the given predicate $p = a \vee b$, we would end up with the following test requirements, two for clause a and two for clause b:

	a	b
$c_i = a$	**T**	F
	F	F
$c_i = b$	F	**T**
	F	**F**

For clause a, a will determine the outcome of p if, and only if, b is $false$. So we have test requirements TRs $= \{(a = T,\ b = F),\ (a = F,\ b = F)\}$ for $c_i = a$. The same principle is applied to the clause b, resulting in the test requirements TRs $= \{(a = F,\ b = T),\ (a = F,\ b = F)\}$ for $c_i = b$. Since there are two identical requirements in the set of requirements for each major clause, we end up with a set of three requirements for the predicate p: TRs $= \{(a = T,\ b = F),\ (a = F,\ b = T),\ (a = F,\ b = F)\}$.

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*.

To explain these variations, let us use the following predicate, which is more interesting:

$$p = a \wedge (b \vee c)$$

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, with $j \neq i$, 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*.

For the given example, we could achieve RACC, for $c_i = a$, with the following test requirements:

a	b	c	$a \wedge (b \vee c)$
T	T	T	**T**
F	T	T	**F**

The table above presents only one of the possibilities to satisfy RACC, the engineer could use others.

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, with $j \neq i$, 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.

For the given example, we could achieve CACC, for $c_i = a$, with the following test requirements:

a	b	c	$a \wedge (b \vee c)$
T	T	F	**T**
F	F	T	**F**

The table above presents only one of the possibilities to satisfy CACC, the engineer could use others.

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, with $j \neq i$, 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*.

For the given example, the engineer could use any of the two previous tables to satisfy GACC for $c_i = a$.

The problem with this criterion is that p might evaluate to the same boolean value for all the test cases, missing a test where p evaluates to an opposite value (e.g. we end up with two test cases where p evaluates to *true* and no test cases where p evaluates to *false*). Consider the predicate $p = a \leftrightarrow b$, to achieve GACC for p we can only use the set of inputs: $\{(a = T, b = T), (a = F, b = F)\}$. For both tests, p evaluates to *true*, so predicate coverage is not satisfied.

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 (corresponding to the previous definition of *clause* or *atomic formula* presented in Section 3.5). A decision is a boolean expression composed of conditions and zero or more boolean operators (corresponding to the *predicate* definition presented in Section 3.5). A decision without a boolean operator is a condition.

The main objective of MC/DC is to test situations where a condition is independently affecting a decision. The independence requirement ensures that the effect of each condi-

tion 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 tested in a situation where it is independently affecting the 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 work in this field varies a lot [Marinescu et al., 2015]. 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, contracts, and pre- and post-conditions). The formal notations considered were: ASM, Alloy, B, Event-B, JML, VDM and Z.

4.1 History and 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 (Test Specification Language). 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.

An interesting aspect of this work is how the authors define a clear, step by step, process to identify the test case categories. This particular characteristic inspired the way we described our test generation case generation approach for BETA.

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 Automaton* (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. In our work, we decided not to use DNF versions of the models under test. Even though predicates in the DNF form are simpler to partition, we believe there is an advantage in not requiring the engineer to change the model before he or she can use our approach (e.g. by changing predicates to DNF form, we lose traceability of the covered predicates in the original model).

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 the 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. Also, the focus of this work is different than ours. Their approach uses Alloy specifications to specify test case scenarios. The single purpose of these specifications is to define the test cases. In our approach, we generate test cases from models that were created for development purposes, using a process that can be done either manually or automatically. In other words, we do not require additional specifications to generate our 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.

Our work shares some similarities with the work presented by [Ambert et al., 2002]. It not only uses the B notation as its input model but it also generates negative test cases and uses boundary values as a test case generation strategy. Unfotunately, since their tool is not available to the public, we could not make more detailed comparisons between their tool and ours.

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.

One of the highlights of this work is the concern with the implementation and execution of the concrete test cases. This particular aspect shares some similarities with our tool, which is also capable of generating test scripts in Java.

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 main difference when compared to [Ambert et al., 2002], is that ProTest does not generate negative test cases as BZ-TT.

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, assuming names are the only difference between the abstract specification and the concrete implementation.

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 it can be used for other model oriented specification languages such as Z and VDM. They also went into more details about the solution for the problem of operations which have non-deterministic behavior. The solution consists of monitoring the internal decisions made and the values attributed to variables during the execution of a non-deterministic operation.

ProTest currently lives inside ProB, and it is a little different from what was presented in previous papers. The tool uses model-checking and constraint-solving based techniques to find sequences of operation calls that are used as test cases. It does not generate executable test scripts and, consequently, it is not capable of comparing the results of the concrete test cases against the animations of the original model.

As a side note, even though the test generation approach proposed by [Satpathy et al., 2005] is different in many aspects when compared to our approach, we still use the current version available in ProB to calculate preambles for our test cases generated using BETA.

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 same manner as *predicate coverage*). 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 works 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, an *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.

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

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

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 automata. 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.

According to the authors, the use of logical coverage criteria for this type of models makes more sense than using automaton coverage criteria in many cases. They state that generating automata from this kind of specifications, in some cases, may result in a single transition that is not useful for test case generation, even though it might contain formulas that are not trivial to cover. Logical coverage criteria can generate more meaningful and in-depth tests from logical formulas. We agree with this point of view and also support a set of logical coverage criteria in our approach.

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. BETA and FASTEST have some similarities. Both generate unit tests using input space partitioning techniques and have similar steps in their approach. While FASTEST uses only logical coverage criteria to generate its test cases, BETA additionally supports input space partitioning criteria. Another difference is that FASTEST only generate positive test cases while BETA generates positive and negative test cases.

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, such as [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 does not use precise criteria to generate test cases. As seen in Table 4.1, there are papers like [Marinov and Khurshid, 2001, Cheon and Leavens, 2002, Gupta and Bhatia, 2010] that use *ad hoc* criteria for test case generation. In our work, we tried to improve this aspect by adopting and implementing common knowledge coverage criteria, which have been evaluated and shown to be effective through time by the software testing community. Our test generation approach implements input space and logical coverage criteria according to the definitions presented in [Ammann and Offutt, 2010].

Another common problem in the work we found in the current literature is the lack of tool support. Many papers focused only on theoretical issues and provided only partial tool support [Amla and Ammann, 1992, Dick and Faivre, 1993, Marinov and Khurshid, 2001, Cheon and Leavens, 2002, Satpathy et al., 2005, Gupta and Bhatia, 2010, Dinca et al., 2012]. Also, in almost every case, the tools developed were not made available to the public, with the exception of [Marinov and Khurshid, 2001, Satpathy et al., 2005, Gupta and Bhatia, 2010, Dinca et al., 2012, Cristiá et al., 2014]. In our work, we dedicated a lot of effort to tool support. The approach is almost entirely automated by a tool, providing mechanisms to generate test data and test case scripts, to define oracles, to calculate preambles and to concretize test data. BETA is available for free to download, and its source code is open to the public.

When it comes to more specific problems such as test data concretization, we found no paper, in the scope of our bibliographic research, that addressed this problem in a detailed 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. In this thesis, we present a strategy that is capable of translating data that was generated from abstract models into concrete data that is more suitable to implement concrete test cases. This strategy relies on formulas that describe the relationship between abstract and concrete test data and uses a constraint solver to obtain actual data from these formulas.

Regarding oracle strategies for test case evaluation, most of the strategies [Marinov and Khurshid, 2001, Ambert et al., 2002, Satpathy et al., 2005] used simple oracle solutions, not providing many options and configurations for the test engineer. BETA is capable of

defining oracle data automatically by animating the original specification and checking the expected results for a particular test case according to the behaviour specified in the model. The approach also offers different oracle strategies that can be combined to perform weaker or stronger oracle verifications.

Ultimately, few papers [Marinov and Khurshid, 2001, Cheon and Leavens, 2002, Satpathy et al., 2005] dealt with the problem of generation of executable test scripts as well. In our work, we developed a model that generates partially executable test scripts in Java and C. These scripts only require a few adaptations before they can be executed and remove a lot of the effort necessary to implement concrete test cases.

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 BETA approach and its process to derive test cases from abstract B machines (Section 5.1). It also includes a presentation of the tool that automates the approach and some details on its implementation (Section 5.2). Ultimately, we conclude with a brief discussion on the evolution of BETA, presenting how the approach and the tool have evolved from [Matos, 2012] to this thesis (Section 5.3).

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* (for now, only abstract machines can be used to generate test cases), 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[1] 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, such as preconditions and invariants. After the characteristics are enumerated, they are used to create test partitions for the input space of the operation under test. Then, combination

[1] We use the term *clause* differently from what is used in the predicate logic standards, as discussed in Section 3.5. What we call *clause* is, in fact, an *atomic formula* or *positive literal*.

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 is animated using these inputs to obtain oracle data (expected test case results). A preamble is also automatically calculated for each one of the test cases. Test inputs, expected results and preambles are then combined into test case specifications that could be either in HTML or XML format. The test case specifications are used as a guide to code the concrete test cases. Before coding the test cases, sometimes it might be necessary to concretize the test data presented in the test case specifications. This happens because the data strucutures used in the abstract specification may differ from the ones used in the actual implementation of the system. This concretization process can be performed either manually or automatically using the tool. Once the test data is concretized, the test case specifications can be translated into executable test scripts. This step can also be performed manually or automatically.

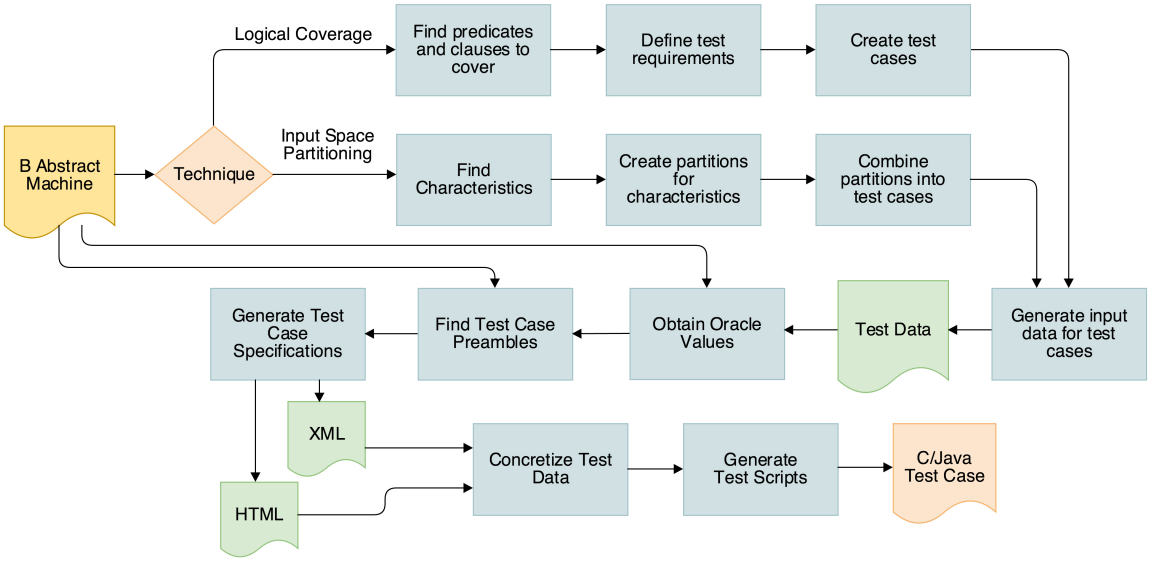

Figure 5.1: An overview of the BETA approach.

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 assumes that the model used as input was previously verified on a proper modeling tool and does not contain any syntactic or semantic faults.

BETA currently supports all the notation used by abstract B machines with the exception of *structs* constructs[2]. It does not support test case generation from *refinements* and *implementations*.

An example of an abstract B machine is presented in Listing 5.1. The *Classroom* machine, which we first introduced in Chapter 2, will be used throughout this chapter to demonstrate how the BETA approach works in practice. This machine manages the students enrolled in a course and their grades. It has four operations:

- *add_student*: enrolls a student in the course;
- *add_grade*: registers a grade for the student in the course;
- *present_on_lab_classes*: registers if a student was present or not in the practical lab class;
- *student_pass_or_fail*: determines if a student passed, failed or is required to take a final exam in the course. If a student had a grade higher than three and was present in the lab class, then he or she passes the course. If a student had a grade three and was present in the lab class, then he or she is required to take a final exam. If the student had a grade bellow three or missed the lab class, then he or she fails the course.

Listing 5.1: The Classroom machine

```
1  MACHINE Classroom
2
3  SETS
4   all_students = {st1,st2,st3,st4,st5,st6,st7,st8,st9,st10};
5   result = {pass, final_exam, fail}
6
7  VARIABLES
8   grades,
9   has_taken_lab_classes,
10  students
11
12 INVARIANT
13  students <: all_students &
14  grades : students +-> 0..5 &
15  has_taken_lab_classes : students +-> BOOL
16
```

[2]Structs are not part of the original B Method notation. Some tools allow the use of structs as an extra feature to help the modelling process but it is not supported by the original method.

```
17 INITIALISATION
18  grades := {} ||
19  students := {} ||
20  has_taken_lab_classes := {}
21
22 OPERATIONS
23   add_student(student) =
24     PRE
25       student : all_students
26     THEN
27       students := students \/ {student}
28     END;
29
30   add_grade(student, grade) =
31     PRE
32       student : students &
33       grade : 0..5
34     THEN
35       grades(student) := grade
36     END;
37
38   present_on_lab_classes(student, present) =
39     PRE
40       student : students &
41       present : BOOL
42     THEN
43       has_taken_lab_classes(student) := present
44     END;
45
46   rr <-- student_pass_or_fail(student) =
47     PRE
48       student : students &
49       student : dom(grades) &
50       student : dom(has_taken_lab_classes)
51     THEN
52       IF grades(student) > 3 &
53          has_taken_lab_classes(student) = TRUE
54       THEN rr := pass
55       ELSIF grades(student) > 2 &
56             has_taken_lab_classes(student) = TRUE
57       THEN rr := final_exam
58       ELSE rr := fail
59       END
60     END
61 END
```

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.

Finding predicates and clauses to cover

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, but since the focus of the approach is on operations, it only searches in substitutions that can be used inside an operation. Currently, BETA searches for predicates to cover in the following places:

– *Precondition* substitutions;

– *If* substitutions;

– *Case* substitutions;

– *Select* substitutions;

– *Any* substitutions.

Table 5.1 presents examples of how predicates and clauses are extracted from these substitutions. The search for predicates and clauses is also extended for operations with nested substitutions. If we have an *if* inside an *elsif* substitution, for example, the predicates for the outter *if-elsif* and for the inner *if* substitution are extracted independently.

Table 5.1: How predicates and clauses are extracted from substitutions.

Substitutions	Example	Predicates	Clauses
Precondition	PRE $C1 \wedge (C2 \vee C3)$	{ $C1 \wedge (C2 \vee C3)$ }	{ C1, C2, C3 }
If-elsif	IF $C1 \wedge C2$ ELSIF $C3$ ELSE	{ $C1 \wedge C2$, C3 }	{ C1, C2, C3 }
Case	CASE x EITHER a OR b ELSE	{ $x = a$, $x = b$ }	{ $x = a$, $x = b$ }
Select	SELECT $C1 \wedge C2$ WHEN $C3$ ELSE	{ $C1 \wedge C2$, C3 }	{ C1, C2, C3 }
Any	ANY x WHERE $C1 \wedge C2$	{ $C1 \wedge C2$ }	{ C1, C2 }

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.

Since the focus of the approach is on operations, predicates present on *Constraints*, *Properties*, *Invariant*, and *Assertions* clauses are not covered. Also, covering clauses in places such as the invariant would result in test requirements that would break the invariant in some cases (e.g. a test requirement that asks an invariant clause to be *false* would break the invariant). This would result in test cases that require the software to be in an inconsistent state before its execution. Even tough it would be interesting for security testing, it is not one of the current objectives of the approach. Predicates used in *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[3] (see Section 3.5).

Defining test requirements

BETA currently supports four logical coverage criteria (more detailed information about the test criteria can be found on Section 3.5.1):

- *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 and each major clause c_i which belongs to the clauses of p, choose minor clauses c_j 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 many variations of ACC (see Section 3.5 for more information). BETA currently uses *General Active Clause Coverage*.

We illustrate how each one of these criteria works for the *Classroom* example in the next section.

[3] This is the definition we decided to use based on the coverage criteria explanations presented in [Ammann and Offutt, 2010]. For a discussion about these terms, see Section 3.5.

Comments on typing predicates In the B notation, types for parameters and state variables are expressed using typing clauses. In a B model, these clauses are merged together with other clauses that apply restrictions to the parameters and variables. There is no clear separation between typing and non-typing predicates, everything belongs to the same set of predicates. Because of this, we have to separate typing and non-typing clauses in different subsets. We do this separation because typing clauses are treated as special clauses. We never modify a typing clause because it would normally result in infeasible test case scenarios or compilation errors (if a strongly typed programming language is being used). For *predicate coverage* we extract the typing clauses from a predicate p and negate the rest when we do a $\neg p$ test, resulting in a test formula like:

$$tc_1 \wedge tc_2 \wedge ... \wedge tc_n \wedge \neg(ntc_1 \wedge ntc_2 \wedge ... \wedge ntc_m) \tag{5.1}$$

Where tc represents typing clauses and ntc represents non typing clauses. For *clause coverage* and *combinatorial coverage*, the approach does not generate tests formulas that require a typing clause c to be $\neg c$. Ultimately, for *active clause coverage*, typing clauses are never considered as major clauses. By doing this, we avoid some cases where typing clauses would need to be negated when generating test formulas for active clause coverage.

Creating test cases

Logical formulas are used to define the test cases. We call these formulas *test formulas*. Some of the algorithms used to create the test formulas for logical coverage are presented in Appendix D.

For PC, CC, and CoC criteria the process is straightforward. For PC, each predicate p results in two test requirements, one for p and another for $\neg p$. The approach then creates test formulas (or test cases) to cover these requirements. If the predicate being covered is a precondition, it creates special formulas for it, one where the precondition is *true* and another where the precondition is *false*. If the predicate being covered is not a precondition, it creates two test formulas: $I \wedge PRE \wedge P$ and $I \wedge PRE \wedge \neg P$ (where I is the invariant, PRE is the precondition and P is the predicate being covered). If the operation has no predicates, the approach creates a single test formula representing the invariant.

For CC, each clause c results in two test requirements, one for c and another for $\neg c$. The approach then creates formulas to cover these requirements. If the operation under test has a precondition, it creates special test formulas for it: one where all clauses of the precondition are *true*, and then formulas negating each clause of the precondition individually. For the remainder of the clauses extracted from the operation, it creates two test formulas in the following format: $I \wedge PRE \wedge C$ and $I \wedge PRE \wedge \neg C$ (where I is the invariant, PRE is the precondition and C is the clause being covered). If the operation has no predicates, the approach creates a single test formula representing the invariant.

For CoC, the set of test requirements is equal to all the combinations of truth values for a predicate. 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 $\neg a \wedge \neg b$. All formulas are connected through conjunction to the invariant of the machine and to the operation's precondition (when necessary). If the operation has no predicates, the approach creates a single test formula representing the invariant.

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 $false$. 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, which relies on the XOR ($\oplus$) operator:

$$p_c = p_{c=true} \oplus p_{c=false} \tag{5.2}$$

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. This formula is concatenated with the invariant and the truth value that we want for the major clause to define the test case formula. It is important to notice that, using this approach, we can not assume predicate coverage is satisfied since there might be cases where p will evaluate to the same truth value for different values of c. This is a known issue with *General Active Clause Coverage*.

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

$$p_{x>0} = p_{x>0=true} \oplus p_{x>0=false} \tag{5.3}$$

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

$$p_{x>0} = true \oplus y < 5 \tag{5.5}$$

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

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 formula is very helpful when dealing with more complex predicates and when it comes to automation of the process.

The following two formulas describe the test case scenarios where $x > 0$ is treated as a major clause and determines the outcome of p. The two test cases represent situations where $p = true$ and $p = false$, satisfying the criterion:

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

$$\neg(x > 0) \wedge \neg(y < 5) \tag{5.8}$$

From this point forward, the process continues as for the other coverage criteria, and the formulas are solved on a constraint solver to obtain input data for the test cases.

To illustrate this process with an example, let us use the operation *student_pass_or_fail* from our *Classroom* machine. If we tried to generate tests for this operation using logical coverage, BETA would extract the following predicates from the model:

$$student \in dom(grades) \wedge student \in dom(htlc) \wedge student \in students \tag{5.9}$$

$$grades(student) > 3 \wedge htlc(student) = TRUE \tag{5.10}$$

$$grades(student) > 2 \wedge htlc(student) = TRUE \tag{5.11}$$

Where the predicate in the formula 5.9 is the precondition and the ones in the formulas 5.10 and 5.11 come from the conditional substitution in the body of the operation (the name of the variable *has_taken_lab_classes* was shortened to *htlc*).

If predicate coverage was employed to generate the test cases, we would end up with the following test formulas, each one of them corresponding to a test case:

$$I \wedge student \in dom(grades) \wedge student \in dom(htlc) \wedge student \in students \tag{5.12}$$

$$I \wedge \neg(student \in dom(grades) \wedge student \in dom(htlc) \wedge student \in students) \tag{5.13}$$

$$I \wedge PRE \wedge grades(student) > 3 \wedge htlc(student) = TRUE \tag{5.14}$$

$$I \wedge PRE \wedge \neg(grades(student) > 3 \wedge htlc(student) = TRUE) \tag{5.15}$$

$$I \wedge PRE \wedge grades(student) > 2 \wedge htlc(student) = TRUE \tag{5.16}$$

$$I \wedge PRE \wedge \neg(grades(student) > 2 \wedge htlc(student) = TRUE) \tag{5.17}$$

Where I represents the invariant of the machine and PRE represents the precondition of the operation that is being tested. We have to add I and PRE to the formula (even though we said before that the approach currently does not cover the invariant) because both the invariant and the precondition contain additional restrictions applied to the operation under test.

For clause coverage, BETA would generate the following test formulas:

$$I \wedge student \in dom(grades) \wedge student \in dom(htlc) \wedge student \in students \tag{5.18}$$

$$I \wedge \neg(student \in dom(htlc)) \wedge student \in dom(grades) \wedge student \in students \tag{5.19}$$

$$I \wedge \neg(student \in dom(grades)) \wedge student \in dom(htlc) \wedge student \in students \tag{5.20}$$

$$I \wedge \neg(student \in students) \wedge student \in dom(grades) \wedge student \in dom(htlc) \tag{5.21}$$

$$I \wedge PRE \wedge htlc(student) = TRUE \tag{5.22}$$

$$I \wedge PRE \wedge \neg(htlc(student) = TRUE) \tag{5.23}$$

$$I \wedge PRE \wedge grades(student) > 2 \tag{5.24}$$

$$I \wedge PRE \wedge \neg(grades(student) > 2) \tag{5.25}$$

$$I \wedge PRE \wedge grades(student) > 3 \tag{5.26}$$

$$I \wedge PRE \wedge \neg(grades(student) > 3) \tag{5.27}$$

For combinatorial clause coverage, BETA would generate the following test formulas:

$$I \wedge student \in dom(grades) \wedge student \in dom(htlc) \wedge student \in students \tag{5.28}$$

$$I \wedge \neg(student \in dom(grades)) \wedge \neg(student \in dom(htlc)) \wedge \neg(student \in students) \tag{5.29}$$

$$I \wedge \neg(student \in dom(grades)) \wedge \neg(student \in dom(htlc)) \wedge student \in students \tag{5.30}$$

$$I \wedge \neg(student \in dom(grades)) \wedge \neg(student \in students) \wedge student \in dom(htlc) \tag{5.31}$$

$$I \wedge \neg(student \in dom(grades)) \wedge student \in dom(htlc) \wedge student \in students \tag{5.32}$$

$$I \wedge \neg(student \in students) \wedge student \in dom(grades) \wedge student \in dom(htlc) \tag{5.33}$$

$$I \wedge \neg(student \in dom(htlc)) \wedge student \in dom(grades) \wedge student \in students \tag{5.34}$$

$$I \wedge \neg(student \in dom(htlc)) \wedge \neg(student \in students) \wedge student \in dom(grades) \tag{5.35}$$

$$I \wedge PRE \wedge grades(student) > 2 \wedge htlc(student) = TRUE \tag{5.36}$$

$$I \wedge PRE \wedge grades(student) > 2 \wedge \neg(htlc(student) = TRUE) \tag{5.37}$$

$$I \wedge PRE \wedge htlc(student) = TRUE \wedge \neg(grades(student) > 2) \tag{5.38}$$

$$I \wedge PRE \wedge \neg(grades(student) > 2) \wedge \neg(htlc(student) = TRUE) \tag{5.39}$$

$$I \wedge PRE \wedge grades(student) > 3 \wedge htlc(student) = TRUE \tag{5.40}$$

$$I \wedge PRE \wedge grades(student) > 3 \wedge \neg(htlc(student) = TRUE) \tag{5.41}$$

$$I \wedge PRE \wedge htlc(student) = TRUE \wedge \neg(grades(student) > 3) \tag{5.42}$$

$$I \wedge PRE \wedge \neg(grades(student) > 3) \wedge \neg(htlc(student) = TRUE) \tag{5.43}$$

For active clause coverage, BETA would generate the test formulas presented bellow. Notice that the formulas use a different format than the one we used to explain how we obtain test formulas for ACC previously in this section, which relied on the XOR ($\oplus$) operator. This modification was necessary because the B notation does not support the XOR operator.

So, instead, we used the equivalence $a \iff \neg b$, which still provides what we need.

$$I \wedge ((TRUE \wedge student \in dom(grades) \wedge student \in dom(htlc)) \iff \tag{5.44}$$
$$(\neg(FALSE \wedge student \in dom(grades) \wedge student \in dom(htlc)))) \wedge student \in students \tag{5.45}$$

$$I \wedge ((TRUE \wedge student \in dom(grades) \wedge student \in dom(htlc)) \iff \tag{5.46}$$
$$(\neg(FALSE \wedge student \in dom(grades) \wedge student \in dom(htlc)))) \wedge \neg(student \in students) \tag{5.47}$$

$$I \wedge ((student \in students \wedge student \in dom(grades) \wedge TRUE) \iff \tag{5.48}$$
$$(\neg(student \in students \wedge student \in dom(grades) \wedge FALSE))) \wedge student \in dom(htlc) \tag{5.49}$$

$$I \wedge ((student \in students \wedge student \in dom(grades) \wedge TRUE) \iff \tag{5.50}$$
$$(\neg(student \in students \wedge student \in dom(grades) \wedge FALSE))) \wedge \neg(student \in dom(htlc)) \tag{5.51}$$

$$I \wedge ((student \in students \wedge TRUE \wedge student \in dom(htlc)) \iff \tag{5.52}$$
$$(\neg(student \in students \wedge FALSE \wedge student \in dom(htlc)))) \wedge student \in dom(grades) \tag{5.53}$$

$$I \wedge ((student \in students \wedge TRUE \wedge student \in dom(htlc)) \iff \tag{5.54}$$
$$(\neg(student \in students \wedge FALSE \wedge student \in dom(htlc)))) \wedge \neg(student \in dom(grades)) \tag{5.55}$$

$$I \wedge PRE \wedge ((grades(student) > 2 \wedge TRUE) \iff \tag{5.56}$$
$$(\neg(grades(student) > 2 \wedge FALSE))) \wedge htlc(student) = TRUE \tag{5.57}$$

$$I \wedge PRE \wedge ((grades(student) > 2 \wedge TRUE) \iff \tag{5.58}$$
$$(\neg(grades(student) > 2 \wedge FALSE))) \wedge \neg(htlc(student) = TRUE) \tag{5.59}$$

$$I \wedge PRE \wedge ((grades(student) > 3 \wedge TRUE) \iff \tag{5.60}$$
$$(\neg(grades(student) > 3 \wedge FALSE))) \wedge htlc(student) = TRUE \tag{5.61}$$

$$I \wedge PRE \wedge ((grades(student) > 3 \wedge TRUE) \iff \tag{5.62}$$
$$(\neg(grades(student) > 3 \wedge FALSE))) \wedge \neg(htlc(student) = TRUE) \tag{5.63}$$

$$I \wedge PRE \wedge ((TRUE \wedge htlc(student) = TRUE) \Leftrightarrow \tag{5.64}$$
$$(\neg(FALSE \wedge htlc(student) = TRUE))) \wedge grades(student) > 2 \tag{5.65}$$

$$I \wedge PRE \wedge ((TRUE \wedge htlc(student) = TRUE) \Leftrightarrow \tag{5.66}$$
$$(\neg(FALSE \wedge htlc(student) = TRUE))) \wedge \neg(grades(student) > 2) \tag{5.67}$$

$$I \wedge PRE \wedge ((TRUE \wedge htlc(student) = TRUE) \Leftrightarrow \tag{5.68}$$
$$(\neg(FALSE \wedge htlc(student) = TRUE))) \wedge grades(student) > 3 \tag{5.69}$$

$$I \wedge PRE \wedge ((TRUE \wedge htlc(student) = TRUE) \Leftrightarrow \tag{5.70}$$
$$(\neg(FALSE \wedge htlc(student) = TRUE))) \wedge \neg(grades(student) > 3) \tag{5.71}$$

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 combination criteria to define test requirements.

Finding 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, 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 algorithm used by the approach to find these input parameters is presented in Appendix C. The parameters of the operation can be easily found in its signature. 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 other variables. 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.

Together with the definition of the input parameters, the approach also defines the characteristics for the operation under test. It is done by finding restrictions applied to the input parameters. In the B-Method, these 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 in:

– *Invariant* clauses;

– *Properties* clauses;

– *Precondition* clauses;

– *If* substitutions;

– *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. The algorithm used by the approach to find these characteristics is presented in Appendix C.

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.

To illustrate this process, let us go back to the *Classroom* example. If we consider the *student_pass_or_fail_example* operation, first we would obtain the set of input parameters IP:

$$\text{IP} = \{\text{students, has\_taken\_lab\_classes, grades, student}\}$$

For this set of input parameters, we obtain the following set of characteristics CHs:

$$\begin{gathered}\text{CHs} = \{\ students \subset all\_students,\ (1)\\ grades \in (students \nrightarrow 0..5),\ (2)\\ has\_taken\_lab\_classes \in (students \nrightarrow BOOL),\ (3)\end{gathered}$$

$$grades(student) > 2 \wedge has\_taken\_lab\_classes(student) = TRUE, \text{ (4)}$$
$$grades(student) > 3 \wedge has\_taken\_lab\_classes(student) = TRUE, \text{ (5)}$$
$$student \in dom(grades), \text{ (6)}$$
$$student \in dom(has\_taken\_lab\_classes), \text{ (7)}$$
$$student \in students \text{ (8) } \}$$

Characteristics (1), (2), and (3) were extracted from the invariant; (4) and (5) from the conditional statement inside the operation; and (6), (7), and (8) from its precondition.

Creating partitions for characteristics

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 an 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 of the same partition or characteristic at the same time. The blocks are also *complete*, meaning that the union of all the blocks of a characteristic covers its entire domain. Each characteristic can be partitioned into 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 in Appendix A.

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 six blocks, depending on the chosen partition strategy. If equivalence classes are used to partition the characteristic, the partition is composed of three blocks: one block for values inside the interval (e.g. 12), one block for values preceding the interval (e.g. 9) and one block for values following the interval (e.g. 26). If boundary value analysis is used, the proposal is to cover it with six blocks: one block containing the value right below the right limit (24), one block representing the right limit (25), one block right above the right limit (26), one block right below the left limit (9), one block representing the left limit (10), and one block right above the left limit (11).

– cases in which the negation of the formula corresponds to situations that normally do not generate interesting tests. The approach considers two situations to be “not

interesting" when partitioning a characteristic in blocks. The first one is 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. The second one is when the origin of a characteristic is the invariant or the properties of the machine. It does 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. In these cases the characteristics correspond to a trivial block partition.

For our running example, we would obtain the blocks presented in Table 5.2, for each characteristic.

Table 5.2: Blocks created for the student_pass_or_fail example

Characteristic	Block 1	Block 2
$ch_1 = students \subset all\_students$	ch_1	$\neg ch_1$
$ch_2 = grades \in (students \nrightarrow 0..5)$	ch_2	$\neg ch_2$
$ch_3 = has\_taken\_lab\_classes \in (students \nrightarrow BOOL)$	ch_3	$\neg ch_3$
$ch_4 = grades(student) > 2 \wedge has\_taken\_lab\_classes(student) = TRUE$	ch_4	$\neg ch_4$
$ch_5 = grades(student) > 3 \wedge has\_taken\_lab\_classes(student) = TRUE$	ch_5	$\neg ch_5$
$ch_6 = student \in dom(grades)$	ch_6	$\neg ch_6$
$ch_7 = student \in dom(has\_taken\_lab\_classes)$	ch_7	$\neg ch_7$
$ch_8 = student \in students$	ch_8	$\neg ch_8$

This example clearly illustrates the most common scenarios where the approach only generates two blocks for each characteristic: one positive block and one negative block. Since none of the characteristics represent an interval, the same blocks presented in Table 5.2 would be generated for both equivalence classes and boundary value analysis.

On the other hand, the *add_grade* operation contains one characteristic that falls into one of the exceptions we mentioned. The $grade \in 0..5$ characteristic specifies an interval that can be partitioned in more than two blocks. Table 5.3 presents the blocks created for this characteristic using equivalence classes and Table 5.4 presents the values representing the blocks created using boundary value analysis.

Table 5.3: Blocks created for the $grade \in 0..5$ characteristic with Equivalence Classes

Block 1	Block 2	Block 3
$grade \in MININT..-1$	$grade \in 0..5$	$grade \in 6..MAXINT$

Table 5.4: Values representing the blocks created for the $grade \in 0..5$ characteristic with Boundary Values

Block 1	Block 2	Block 3	Block 4	Block 5	Block 6
$grade = -1$	$grade = 0$	$grade = 1$	$grade = 4$	$grade = 5$	$grade = 6$

Combining 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 combination criteria. BETA currently supports three combination 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 [Lei and Tai, 1998];

– *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 algorithms for the three coverage criteria are presented in details in Appendix C.

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.

As an example of the formulas generated in this step, we present the test formulas generated for the *student_pass_or_fail* operation using equivalence classes, and the each-choice and pairwise combination criteria (we will not present the formulas for the all-combinations criterion because it would take too much space since it generates 32 test formulas or test cases)[4]:

[4]It is important to notice that only the first formulas for each criterion represent a feasible test case. The remainder of the formulas contain logical inconsistencies that make them infeasible test case scenarios.

Each-choice formulas (2 test cases):

$$grades(student) > 2 \wedge grades(student) > 3 \wedge htlc(student) = TRUE \wedge student \in dom(grades) \wedge \quad (5.72)$$
$$student \in dom(htlc) \wedge student \in students \quad (5.73)$$

$$not(grades(student) > 2 \wedge htlc(student) = TRUE) \wedge not(grades(student) > 3 \wedge \quad (5.74)$$
$$htlc(student) = TRUE) \wedge not(student \in dom(grades)) \wedge not(student \in dom(htlc)) \wedge \quad (5.75)$$
$$not(student \in students) \quad (5.76)$$

Pairwise formulas (6 test cases):

$$grades(student) > 2 \wedge htlc(student) = TRUE \wedge \neg(grades(student) > 3 \wedge \quad (5.77)$$
$$htlc(student) = TRUE) \wedge student \in dom(grades) \wedge student \in dom(htlc) \wedge student \in students \quad (5.78)$$

$$grades(student) > 2 \wedge grades(student) > 3 \wedge htlc(student) = TRUE \wedge \quad (5.79)$$
$$\neg(student \in dom(grades)) \wedge student \in dom(htlc) \wedge student \in students \quad (5.80)$$

$$grades(student) > 2 \wedge grades(student) > 3 \wedge htlc(student) = TRUE \wedge \quad (5.81)$$
$$\neg(student \in dom(htlc)) \wedge \neg(student \in students) \wedge student \in dom(grades) \quad (5.82)$$

$$grades(student) > 3 \wedge htlc(student) = TRUE \wedge \neg(grades(student) > 2 \wedge \quad (5.83)$$
$$htlc(student) = TRUE) \wedge \neg(student \in dom(htlc)) \wedge student \in dom(grades) \wedge \quad (5.84)$$
$$student \in students \quad (5.85)$$

$$grades(student) > 3 \wedge htlc(student) = TRUE \wedge \neg(grades(student) > 2 \wedge \quad (5.86)$$
$$htlc(student) = TRUE) \wedge \neg(student \in students) \wedge student \in dom(grades) \wedge \quad (5.87)$$
$$student \in dom(htlc) \quad (5.88)$$

$$\neg(grades(student) > 2 \wedge htlc(student) = TRUE) \wedge \neg(grades(student) > 3 \wedge \quad (5.89)$$
$$htlc(student) = TRUE) \wedge \neg(student \in dom(grades)) \wedge \neg(student \in dom(htlc)) \wedge \quad (5.90)$$
$$\neg(student \in students) \quad (5.91)$$

5.1.4 Generating input data for test cases

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

The BETA approach 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. The BETA tool 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;

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 specification. 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 represents an infeasible test case scenario. An infeasible test scenario is a result of some logical inconsistency in the test formula.

Considering the formulas generated for our pairwise example in the previous section, the first formula represents a feasible test case while the remainder of them contain logical inconsistencies that make them infeasible. The constraint solver will only be able to generate test data for the first formula. Table 5.5 presents the generated data.

Table 5.5: Generated test data for student_pass_or_fail using IPS

Input Parameter	Generated Data
$students$	$\{st1\}$
$has\_taken\_lab\_classes$	$\{(st1 \mid -> TRUE)\}$
$grades$	$\{(st1 \mid -> 3)\}$
$student$	$st1$

Regarding the time and performance to generate test data from these test formulas, in our experiments, when the formulas are solvable, the constraint solver usually takes seconds to solve them, even for our larger models. The number of test formulas generated for a particular coverage criterion has a more meaningful impact on the time necessary to generate the test case specifications. For example, when using the all-combinations criterion, the approach may produce dozens of formulas that, when solved in sequence, may require a few minutes to terminate.

Test data randomization

Another element added to the BETA approach and tool was the test data randomization feature. Sometimes, when solving formulas, constraint solvers can provide data that is too trivial and straightforward to its variables. For example, when generating data for an array of integers, it might always produce arrays where all elements are equal to zero, which is not interesting for testing purposes. It is much better to generate data in a more random way.

With that in mind, BETA now supports test data randomization. This is possible thanks to a randomization feature inside ProB that uses Kodkod[5] to solve the constraints. Using this feature, we could generate more interesting test case scenarios.

The randomization feature is currently an optional configuration in the tool settings. The user can choose to use it or not, although we encourage the use of randomization since it produces better test suites, achieving better coverage results as discussed in our experiments in Chapter 6.

5.1.5 Obtaining 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. It is important to point out that this feature only works for machines that do not use non-deterministic substitutions. If there is non-determinism in the specification, this process has to be done manually, relying on the engineer's own criteria.

The oracle verification can be done using one or a combination of the following oracle strategies, which determine what kind of verifications are done by the test oracle:

- *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 that is generated by the test script generation module;
- *State variables checking*: executes the test and verifies if the values for the state variables are the ones expected according to the specification;
- *Return variables checking*: executes the test and verifies if the values returned by the operation are the ones expected according to the specification.

[5]Kodkod's website: http://alloy.mit.edu/kodkod/

The four strategies can be used separately, to make weaker verifications, or combined, to make stronger verifications. These strategies were implemented in the test case generator developed in [Souza Neto and Moreira, 2014] and are inspired by the ones presented in [Li and Offutt, 2014].

The last two strategies can be perfomed either automatically or manually. Figure 5.2 presents an overview of the automated process. The BETA tool interacts with ProB's API to animate the original model, calling the operation under test and passing as parameters the inputs for the test case. After the animation, BETA checks what is the expected result for the test case. The process 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 (a sequence of transitions) to a state where a given predicate P holds true. So, first of all, it is necessary to define the predicate P that specifies the intended state. Since a test case already defines the values of the state variables, the definition of this predicate is straightforward. The tool will write P 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 model of the operation under test passing test values as parameters, if the particular operation has any parameters;

3. After the model of the operation under test is executed, the machine will transition to a new state. The tool then queries information about the new state – such as the updated values for the state variables – to define what are the expected values for the test case oracle.

Unfortunately, there are still some limitations in the ProB API side that affect our implementation for the automatic oracle evaluation. ProB allows the use of deferred set elements in its GUI version, but the API has no methods to use such elements. This is an expected limitation since, according to the B Method theory, elements of deferred sets are unknown. 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).

If modification of the model is not viable, there is also the possibility to perform the oracle definition process manually. To do this, the test engineer has to open the original model using ProB's interface, and animate the operation under test using the test inputs. After that he has to manually check what is the expected system behavior for the test case executed.

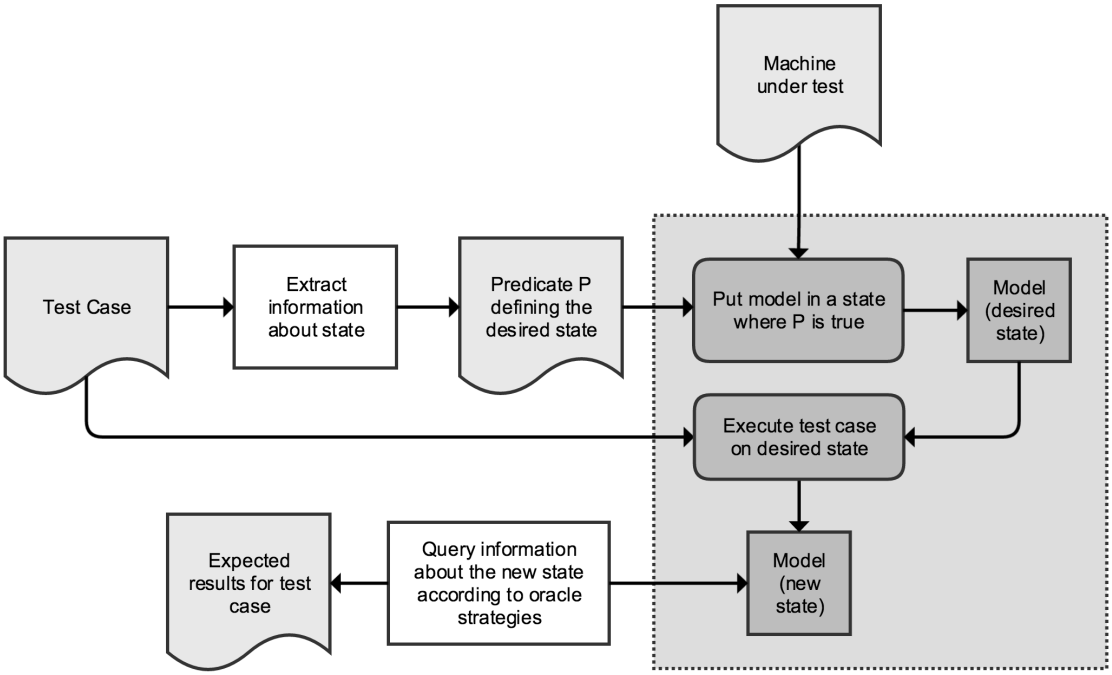

Figure 5.2: Oracle strategy overview. The sheets of paper represent artifacts used during the process, white rectangles represent processes performed by BETA, rectangles with rounded corners represent the steps performed by ProB (all activities performed by ProB are grouped inside the dotted rectangle), and darker rectangles represent transitions in the model.

That is the general process applied to positive test cases. On the other hand, negative tests will most likely violate the operation's preconditions. When a precondition is broken, according to the B Method's philosophy, it is impossible to foresee how the system will behave. In this case, the test engineer has to define – using his own knowledge – the criteria used by the oracle to evaluate the test case results.

For our running example, the operation *student_pass_or_fail* does not modify any state variable, so their values should be maintained according to the oracle. However, the operation has a return variable rr, and according to the specification, it should return the value "*final_exam*" for the test data presented in Table 5.5.

5.1.6 Finding test case preambles

Test cases are not simply a set of input values and expected results to test a piece of software. As explained in Chapter 3, they are artifacts with multiple parts that require some effort to design. In this section we explain how the BETA approach defines the *prefix values* of a test case, or the so called *test case preambles*.

Preambles are sometimes challenging to create. Once the state necessary to execute a test case is identified, it is necessary to put the software in this specific state (from now on we will refer to this state as *required state*). Usually, this could be done in one of two ways. The

first and more simple way is to use *set functions* to modify the state of the software. Using these functions, we can attribute values to the state variables of the software directly. The second way to do this is to use functions already available in the implementation to put the software in the required state. Sometimes that is the only way to do this since set functions are not always available. The problem with this second solution is to find the sequence of functions and values for their parameters that will lead the software to the required state to execute the test case.

Initially, when we defined preambles for BETA test cases, we used the first solution and required the use of set functions to modify the state of the software as we wanted. Unfortunately, after a few case studies, we confirmed the fact that we could not always rely on the availability of such functions. So, we ended up with the following research question: *"How can we find sequences of function calls and its respective parameter values to compose preambles for BETA test cases?"*.

The preamble question has been addressed by previous work, such as [Dick and Faivre, 1993], [Legeard et al., 2002], [Satpathy et al., 2005], and [Dinca et al., 2012]. They focused on system or module level of testing and used testing criteria based on operation coverage, which produces sequences of operation calls as test cases. These test cases begin with an initialization and finish when a test requirement is met. In these cases, a test case already has preamble when it is created. In our case, we generate unit test cases, which test functions individually. We define the input data for these tests first and only after this we find a suitable preamble for the test cases.

To answer the aforementioned question, we developed a strategy that uses ProB's test case generation features to find the sequences of functions and parameter values that we need to compose our test case preambles.

An overview of ProB's test case generation features

In our work, we used some features from ProB's test case generators[6] to define preambles for BETA test cases. Using ProB, one can generate test cases from different notations, such as B, Event-B, TLA+ and Z. In the context of ProB, a test case is a sequence of operations (or events, in the case of Event-B models), along with parameter values for each operation call, concrete values for the constants, and initial values for the model. Each test case begins with an initialization and has one or more operation calls.

The coverage criteria currently supported by ProB are restricted to operation coverage. For example, using one of ProB's test case generators, a test engineer can generate test suites that ensure that every operation in the model is exercised by at least one test case.

ProB supports two different technologies (or algorithms) for test case generation: *model-checker based* and *constraint-based* test case generation.

[6] ProB's Wiki source: http://stups.hhu.de/ProB/w/Test_Case_Generation

The *model-checker based algorithm* (MC) uses ProB's model-checker to generate the state space of the formal model that is being tested. The state space is a graph where each node is a reachable state in the model and the edges that connect the nodes are operation calls that make transitions between the states. ProB's model-checker can compute the complete state space for the model if it is finite and small. If the model has an infinite state space, the state space generation process terminates when the machine runs out of memory. For both cases – finite and infinite state spaces – the model-checker will also stop when an error is found in the model.

By default, the model-checker uses a mix of depth-first/breadth-first search to build the state space. This search can be customized using the settings in the model-checker. The settings allow the user to define if the search should use depth-first or breadth-first only, if it should be randomized, or if it should use a heuristic defined by a function provided by the user. It is important to notice that the extension of the search is also limited by the model-checker settings since they limit the size of deferred and parameter sets, the number of computed initialisations and enabled operations by state. If the settings for this parameters are set too low, the state space generated may be too limited. On the other hand, if the parameters are set too high, the process may not terminate.

When it comes to test case generation, the state space generation stops when the coverage criterion has been satisfied. The generated state space contains all possible initializations, along with all possible values for the constants and values for the operations parameters. The state space generated by this process represents a set of test cases.

The *constraint-based algorithm* (CBC) uses ProB's constraint solver to generate feasible sequences of operations, searching in a breadth-first fashion and also stops when the coverage criterion has been satisfied. It will not examine every possible valuation for the constants and initial values of the machine, nor every possible value for the parameters of the operation. Hence, the CBC algorithm does not construct the full state space of the formal model, but rather constructs a tree of feasible execution paths. This algorithm is more advisable to use when the model has a large number of possible values for variables or parameters since it might be more effective.

In our work, we decided to use ProB's CBC test case generator as a mechanism to calculate preambles for BETA test cases. The CBC test case generator is more suitable for this task because it allows the user to define what operation should be the last in each test sequence – which is useful to define our *state goals* presented later in this chapter – and it also provides parameter values for each operation call in the test sequence. In the next section, we explain how we use it to create our test case preambles.

Preamble generation strategy: using CBC to generate preambles

BETA is capable of generating test cases for all the operations of the *Classroom* machine. To illustrate the preamble calculation process, let us use as an example the test case for the operation *student_pass_or_fail* presented in Figure 5.3.

State	
Variable	Value
students	*{st1}*
has_taken_lab_classes	*{(st1|->TRUE)}*
grades	*{(st1|->4)}*
Input Values	
Parameter	Value
student	*st1*

Figure 5.3: Test case for *student_pass_or_fail* from *Classroom* machine.

This test case defines values for the input parameters of the operation and the state variables of the machine. Supposing that there are no set functions available to modify the state variables directly and that it is not possible to implement them, then we have to use the functions already available in the implementation to put the system in the required state for the test case. To do this, we use ProB's CBC test case generator as a mechanism to find a sequence of operation calls that will work as our test case preamble.

The CBC test case generator can be executed either via GUI (presented in Figure 5.4) or CLI interfaces; or using ProB's Java API. The test case generator is quite straightforward to use and only requires a few parameters before it can be executed. The parameters are the following:

- *Operations that should be covered:* the list of operations that must be covered by the test cases;
- *Maximum search depth:* the maximum depth explored by the constraint solver while building the state space for the model;
- *Predicate to identify target states:* the predicate that must be satisfied by the state after the execution of the last operation in the sequence;
- *Targeted events must be final only:* the selected operations must be covered as final operations only in the test case sequence.

The CBC test case generator was integrated into BETA using the ProB's Java API. An overview of the preamble generation process is presented in Figure 5.5.

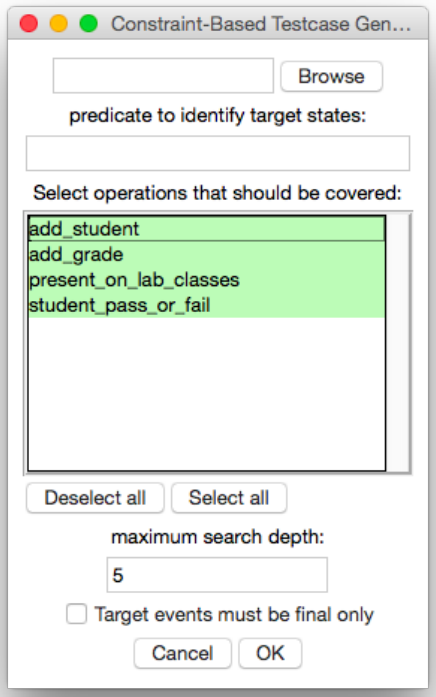

Figure 5.4: The CBC test case generator.

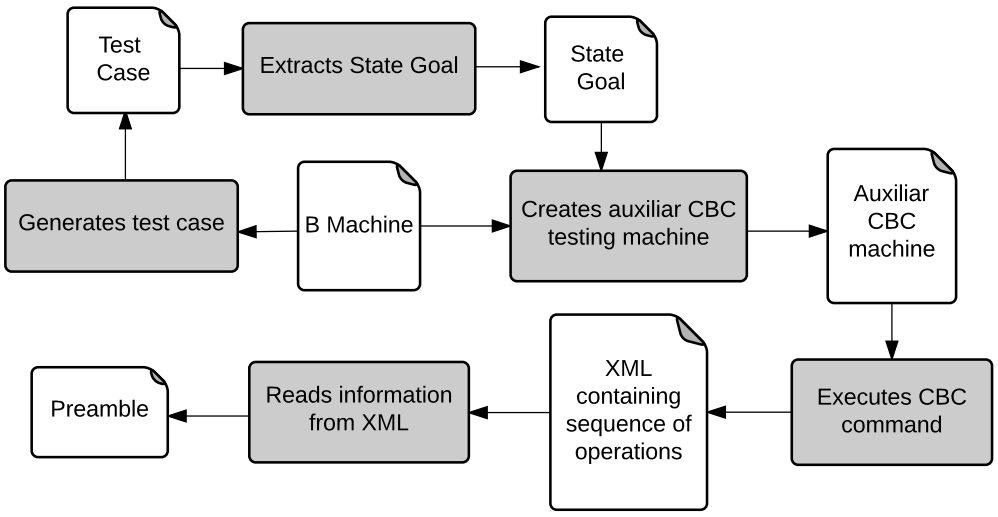

Figure 5.5: The preamble calculation process using ProB's CBC test case generator. Gray boxes represent BETA tasks and white sheets represent input/output artifacts.

The goal of this process is to elaborate a sequence of operation calls that begins with an initialization and terminates in a required state. To do this, we extract the information on the state from the test case, and we define a *state goal* for our preamble. A state goal is simply a predicate that represents the required state. For the *student_pass_or_fail* example presented in Figure 5.3, the state goal is the following:

$$students = \{st1\} \wedge has\_taken\_lab\_classes = \{(st1 \mapsto TRUE)\} \wedge grades = \{(st1 \mapsto 4)\}$$

Once the state goal is defined, BETA creates an auxiliary B machine that is used by the CBC test case generator to create the preamble. This auxiliary machine includes the original machine that contains the operation under test and promotes all of its operations, except the operation under test. If the machine imports any other machine (using one of B's modularization features), the auxiliary machine also includes these machines and promotes their operations. By doing this, we allow the CBC test case generator to use all the operations from the original machine and from the machines imported in our test case preamble, but it will not be able to use the operation under test in the sequence of operations. The machine also contains an auxiliary operation that is used to guide the CBC algorithm to reach the state goal. This operation has as its precondition the state goal and does not have any behavior. By setting this operation as a final operation for our test sequence, the CBC test case generator will provide a sequence that begins with an initialization and contains one or more operation calls that will lead to the required state (a state where the state goal holds true).

To create a preamble for the *student_pass_or_fail* example presented in Figure 5.3, BETA generates the auxiliary machine presented in Listing 5.2.

Listing 5.2: Example of auxiliary machine to calculate preambles

```
1  MACHINE ClassroomCBCTest
2
3  INCLUDES Classroom
4
5  PROMOTES add_grade, add_student, present_on_lab_classes
6
7  OPERATIONS
8  student_pass_or_fail_test1 =
9  PRE
10   students = {st1} &
11   has_taken_lab_classes = {(st1 |-> TRUE)} &
12   grades = {(st1 |-> 4)}
13 THEN skip
14 END
15 END
```

After the auxiliary B machine is generated, we run the CBC test case generator on it using the ProB API which is integrated into BETA. The CBC's algorithm then finds 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 the auxiliary operation is enabled, then it means that the system has reached the state goal.

When the CBC test case generator finishes its execution, it outputs an XML file that contains the sequence of operation calls which represents the preamble for our test case. Listing 5.3 presents the XML generated for the *student_pass_or_fail* example using the auxiliary machine presented in Listing 5.2 (this preamble will be presented in a more readable way in the next section).

Listing 5.3: Example of XML generated by the CBC test case generator from a auxiliary machine

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <extended_test_suite>
3  <test_case>
4    <initialisation>
5      <value type="variable" name="grades">{}</value>
6      <value type="variable" name="has_taken_lab_classes">{}</value>
7      <value type="variable" name="students">{}</value>
8    </initialisation>
9    <step name="add_student">
10     <value name="student">st1</value>
11     <modified name="students">{st1}</modified>
12   </step>
13   <step name="add_grade">
14     <value name="student">st1</value>
15     <value name="grade">4</value>
16     <modified name="grades">{(st1|->4)}</modified>
17   </step>
18   <step name="present_on_lab_classes">
19     <value name="student">st1</value>
20     <value name="present">TRUE</value>
21     <modified name="has_taken_lab_classes">
22       {(st1|->TRUE)}
23     </modified>
24   </step>
25   <step name="student_pass_or_fail_test1" />
26 </test_case>
27 </extended_test_suite>
```

Ultimately, BETA reads the XML file generated, extracts the preamble information and formats it so it can be added to its own test case specifications. An example of BETA test case specification for the *student_pass_or_fail* example contaning a preamble is presented

in Figure 5.6.

Preamble:

1. initialisation({students={}, has_taken_lab_classes={}, grades={}})
2. add_student({student=st1})
3. add_grade({student=st1, grade=4})
4. present_on_lab_classes({present=TRUE, student=st1})

Input data:

State	
Variable	**Value**
students	*{st1}*
has_taken_lab_classes	*{(st1|->TRUE)}*
grades	*{(st1|->4)}*
Input Values	
Parameter	**Value**
student	*st1*

Figure 5.6: Example of BETA test case with preamble.

Implementation Limitations It is important to notice that there are some limitations in our implementation of the strategy proposed in this section. The first limitation is related to the complexity of the models it can support. If the models are too complex – resulting in a large state space – the CBC test case generator may not find a sequence of operations that reaches the state goal. Also, in some of our experiments, we noticed that when the *maximum search depth* is set too high, the algorithm may not terminate. This is a limitation of the constraint solver we use, and we believe the same problem would occur for other constraint solvers since it is a common challenge for this type of tools. Another limitation of the implementation is that it currently does not support models with deferred sets for the same reasons already explained in Section 5.1.5. Currently, as a solution to this problem, we require the user to replace the deferred sets in the model with enumerated sets. In the future, we are planning to implement this fix in the tool, so it could automatically replace, internally, the deferred sets with enumerated sets and avoid these modifications in the original model.

5.1.7 Generating test case specifications

Once the test cases are generated with all of their parts, 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. These tests are created using a particular combination of testing strategy and coverage criteria. Each test case in the specification is composed of three parts:

1. *Preamble*: a preamble which contains a sequence of operation calls and its respective parameters that will put the system in the state intended to execute the test case;
2. *Input data*: this part contains test values for the input parameters of the operation under test (for both operation parameters and state variables);
3. *Expected results*: this part represents the test case oracle and contains the values expected after the execution of the test case.

This information is displayed in an HTML page (like the one presented in Figure 5.6) that contains the test case specification. The tool also supports the generation of test specifications in XML format. The XML specifications contain the same information present in 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. An example of XML test case specification is presented in Appendix B.

Once a specification is generated, the engineer can proceed and implement the executable test cases in a programming language and test framework of his choice. The HTML test case specification contains all the information necessary to code the tests. The engineer can also use an XML specification to generate partially executable test scripts using a separate module in the tool (more information about this module in Section 5.1.9).

5.1.8 Concretizing test data

The test values generated by BETA are abstract values based on data structures from abstract models. In some cases, before we can implement concrete test cases, it is necessary to translate these values into concrete test data. In this section, we present our strategy for test data concretization in the BETA approach.

Every test case generated by BETA is represented internally by a test case formula. This formula specifies the conditions for the test case. It defines values for each input variable – both parameters and state variables – which represent part of the input space of the operation under test. A test case formula has the following format:

$$\exists av_1, av_2, ..., av_n.(c_1 \wedge c_2 \wedge ... \wedge c_i) \tag{5.92}$$

where $av_1, av_2, ..., av_n$ is a list of abstract variables and parameters that compose the input space of the operation under test, and $c_1 \wedge c_2 \wedge ... \wedge c_i$ are restrictions to the values of these quantified variables.

Using a constraint solver, one can evaluate this formula to obtain test data for the test case. The constraint solver will provide values for the quantified variables in the formula; these values are our test case values.

As we mentioned before, these values are abstract values. These values will only sometimes match the data structures used by the implementation of the model. During the refinement process, the data structures used for each variable will most likely change, so it is necessary to translate the abstract values into concrete values that can be used to test the implementation of the model. It is important to notice that test data concretization is an optional step. If the engineer only needs abstract test data, it is possible to skip this step. The BETA approach tries to be as flexible as possible.

If the B Method's refinement process is completely followed, we can rely on the gluing invariant as a mechanism to find the relationship between abstract and concrete variables. If the refinement process is not followed, this relationship has to be mapped by hand.

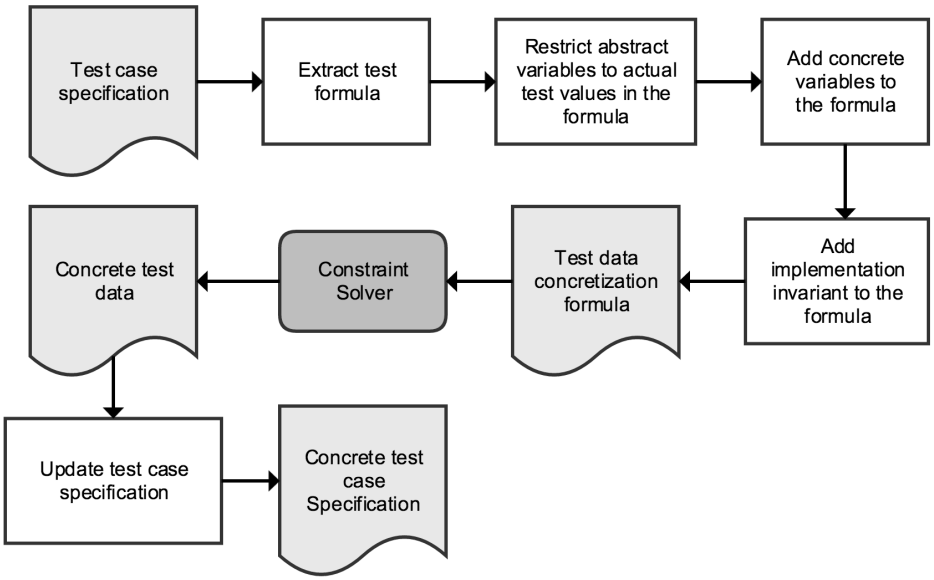

Figure 5.7: The process to create the test data concretization formula. Rectangles represent steps performed by BETA, sheets represent artifacts used during the process, and rectangles with rounded edges represent steps performed by external tools.

To translate the abstract values to concrete values we make some modifications in the test case formula to create a test data concretization formula. This new formula can provide values for both abstract and concrete variables (an overview of the process to create this formula is presented in Figure 5.7):

$$\exists av_1, av_2, ..., av_n, cv_1, cv_2, ..., cv_m.(c_1 \wedge c_2 \wedge ... \wedge c_i \wedge I_{imp}) \tag{5.93}$$

In the above formula, $av_1, av_2, ..., av_n$ is the list of abstract variables and parameters (from the abstract machine) that compose the input space of the operation under test;

$cv_1, cv_2, ..., cv_m$ is the list of concrete variables of its respective implementation module; $c_1 \wedge c_2 \wedge ... \wedge c_i$ are restrictions to the values of the abstract variables and parameters; and I_{imp} is the invariant of the implementation module.

For implementation purposes, we have to add another information to the concretization formula. Since the process requires two separate steps: one to generate the abstract data and another to generate the concrete data; we have to clarify in the concretization formula the values for the abstract variables that we are trying to concretize. We do this by adding the clauses $av_1 = x \wedge av_2 = y \wedge ...av_n = z$ to the quantification predicate. This step is necessary because otherwise we would generate a set of concrete values that would satisfy the test case restrictions but could be different than the abstract values generated initially. So the updated formula would look like the following:

$$\exists av_1, av_2, ..., av_n, cv_1, cv_2, ..., cv_m.(av_1 = x \wedge av_2 = y \wedge ...av_n = z \wedge c_1 \wedge c_2 \wedge ... \wedge c_i \wedge I_{imp}) \tag{5.94}$$

For models with multiple refinement steps, the formula would be a little different. It would be necessary to add to the list of quantified variables all the variables from each refinement step. Also, we would have to add the invariant from each one of these refinements to the quantified predicate using conjunctions. By doing this, we would be able to find the transitive relationship between each refined variable. Currently, the tool only supports one step refinements, but we plan to improve it to support refinements with multiple steps in the future.

When evaluating this formula on a constraint solver, we obtain the values for both abstract variables and the concrete variables of the implementation. Suppose we have in the model an abstract variable av_1 that is defined in the implementation as cv_1. The implementation will have a gluing invariant that maps av_1 to cv_1. Thanks to this gluing invariant, the constraint solver will find a value for cv_1 that matches the value of av_1 as defined by the gluing invariant. The implementation invariant I_{imp} will also provide typing clauses that are important to define the types of the concrete variables or other restrictions.

Let us use the *Player* example (Listings 5.4 and 5.5) to show how the process works in practice. The example was extracted from [Schneider, 2001]. It specifies a machine which manages a soccer team. The machine has a single variable which stores the current team, and a deferred set PLAYER represents the larger set of players. The machine has two operations: one to substitute a player in the current with another player out of the team, and a second that checks if a particular player is currently on the team. In the implementation level (Listing 5.5), the abstract set of players is refined to an array of integers, a typical example of data structure refinement in the B Method.

Listing 5.4: The Player machine

```
1  MACHINE Player
```

```
2
3  SETS
4    PLAYER
5
6  PROPERTIES
7    card(PLAYER) > 11
8
9  VARIABLES
10   team
11
12 INVARIANT
13   team <: PLAYER & card(team) = 11
14
15 INITIALISATION
16   ANY
17     tt
18   WHERE
19     tt <: PLAYER & card(tt) = 11
20   THEN
21     team := tt
22   END
23
24 OPERATIONS
25   substitute(pp, rr) =
26   PRE
27     pp : PLAYER & pp : team & rr : PLAYER & rr /: team
28   THEN
29     team := (team \/ {rr}) - {pp}
30   END;
31
32   aa <-- in_team(pp) =
33   PRE
34     pp : PLAYER
35   THEN
36     IF pp : team THEN
37       aa := TRUE
38     ELSE
39       aa := FALSE
40     END
41   END
42 END
```

Listing 5.5: The Player_i implementation which refines Player

```
1  IMPLEMENTATION Player_i
2
```

```
3  REFINES
4    Player
5
6  VALUES
7    PLAYER = 1..22
8
9  CONCRETE_VARIABLES
10   team_array
11
12 INVARIANT
13   team_array : 0..10 >-> 1..22 &
14   ran(team_array) = team
15
16 INITIALISATION
17   team_array(0) := 1;
18   team_array(1) := 2;
19   team_array(2) := 3;
20   team_array(3) := 4;
21   team_array(4) := 5;
22   team_array(5) := 6;
23   team_array(6) := 7;
24   team_array(7) := 8;
25   team_array(8) := 9;
26   team_array(9) := 10;
27   team_array(10) := 11
28
29 OPERATIONS
30   substitute ( pp , rr ) =
31   BEGIN
32     VAR ii, pl IN
33       ii := 0;
34       pl := 0;
35       WHILE ii < 11 DO
36         pl := team_array(ii);
37         IF pl = pp THEN
38           team_array(ii) := rr;
39           ii := 11
40         ELSE
41           ii := ii + 1
42         END
43       INVARIANT
44         ii : 0..11
45       VARIANT
46         11 - ii
47       END
48     END
49   END;
```

```
50
51    aa <-- in_team( pp ) =
52    BEGIN
53      VAR ii, pl IN
54        ii := 0;
55        pl := 0;
56        aa := FALSE;
57        WHILE ii < 11 DO
58          pl := team_array(ii);
59          IF pl = pp THEN
60            aa := TRUE;
61            ii := 11
62          ELSE
63            ii := ii + 1
64          END
65        INVARIANT
66          ii : 0..11
67        VARIANT
68          11 - ii
69        END
70      END
71    END
72  END
```

Suppose BETA generates a test case that is represented by the test formula bellow:

$$\exists(team, pp, rr).(team \subset PLAYER \wedge card(team) = 11 \tag{5.95}$$

$$\wedge\, pp \in PLAYER \wedge pp \in team \wedge rr \notin team \wedge rr \in PLAYER) \tag{5.96}$$

Following the strategy we just proposed, the first thing we have to do is to add the concrete variables from the implementation into the list of quantified variables of the formula. The *Player_i* implementation has a single concrete variable called *team_array*. Once we add this variable to the concretization formula, it will look like this:

$$\exists(team, pp, rr, team\_array).(team \subset PLAYER \wedge card(team) = 11 \tag{5.97}$$

$$\wedge\, pp \in PLAYER \wedge pp \in team \wedge rr \notin team \wedge rr \in PLAYER) \tag{5.98}$$

The next step is to add the implementation's invariant (I_{imp}) and the test case values for

the abstract variables to the formula's predicate:

$$\exists(team, pp, rr, team\_array).(pp = PLAYER1 \wedge rr = PLAYER12 \tag{5.99}$$

$$\wedge\, team = \{PLAYER1, PLAYER2, PLAYER3, ..., PLAYER10, PLAYER11\} \tag{5.100}$$

$$\wedge\, team \subset PLAYER \wedge card(team) = 11 \tag{5.101}$$

$$\wedge\, pp \in PLAYER \wedge pp \in team \wedge rr \notin team \wedge rr \in PLAYER \tag{5.102}$$

$$\wedge\, team\_array \in 0..10 \mapsto 1..22 \wedge ran(team\_array) = team) \tag{5.103}$$

The concretization formula is now complete and can be evaluated using a constraint solver.

Going back to the test case formula (see 5.95 and 5.96) of our example, when we evaluated it using ProB, we obtained the values presented in Table 5.6 (notice that ProB can provide different solutions for the same test case formula).

Table 5.6: Values obtained after the evaluation of the test case formula.

Variable	Value
pp	PLAYER1
rr	PLAYER12
team	{PLAYER1,PLAYER2,PLAYER3,PLAYER4,PLAYER5,PLAYER6, PLAYER7,PLAYER8,PLAYER9,PLAYER10,PLAYER11}

The values for the variable *team* are generated according to the type defined in the abstract model, which in this case is a subset of a deferred set (see lines 3, 4 and 10 from Listing 5.4). Deferred sets are regularly used on abstract B machines since they help to postpone the decision about how the types and data structures for some variables should be implemented. In this case, ProB generates a set of abstract, enumerated values that represent the elements of the deferred set.

These abstract values are not available and cannot be used in the software implementation. Using our test data concretization formula created for this example, we can translate the abstract test data into concrete test data. After evaluating the test data concretization formula, ProB would provide the solution presented in Table 5.7.

The abstract set was translated into an array that is more suitable to test the implementation. Using this information, the test engineer can implement and execute a concrete test case, with no need to make adaptations in the abstract data generated for the test case specifications.

Strategy Limitations Currently, our implementation of the strategy only works for one step refinements, for models that consist of an abstract machine and an implementation. As we showed in the presentation of the strategy, it can work for models with multiple

Table 5.7: Values obtained after the evaluation of the test data concretization formula.

Variable	Value
pp	PLAYER1
rr	PLAYER12
team	{PLAYER1,PLAYER2,PLAYER3,PLAYER4,PLAYER5,PLAYER6, PLAYER7,PLAYER8,PLAYER9,PLAYER10,PLAYER11}
team_array	{(0 ↦ 1),(1 ↦ 2),(2 ↦ 3),(3 ↦ 4), (4 ↦ 5),(5 ↦ 6), (6 ↦ 7),(7 ↦ 8), (8 ↦ 9),(9 ↦ 10),(10 ↦ 11)}

refinements, but due to time restrictions we decided only to support one step refinements in our implementation for now. The implementation of the strategy can be further developed to support multiple refinements, but we will leave this problem for future work.

5.1.9 Generating 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. This module was developed by [Souza Neto, 2015]. Since it is an external module we will not go into further details about its implementation, and will only explain how it works.

It receives the XML file as input and then extracts from it 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 languages, using the JUnit[7] and CuTest[8] frameworks, respectively.

The generated test script still has to be adapted before it can be executed. Adaptations are required when names of variables and methods change during the refinement and development process.

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

Currently, the test script generator only supports two languages and test frameworks, but we are implementing it in a way that more programming languages and test frameworks can be supported by the tool in the future. More information about this module can be found in BETA's website.

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

5.2 The BETA Tool

In this section we present a brief overview of the BETA tool. The BETA tool automates the whole test generation process presented in Figure 5.1. It is capable of generating test case specifications and executable test scripts from abstract B machines using either input space partitioning criteria or logical coverage criteria. The generated test cases are complete test cases containing test case data, oracle values and preambles. Figure 5.8 shows the interface of the tool.

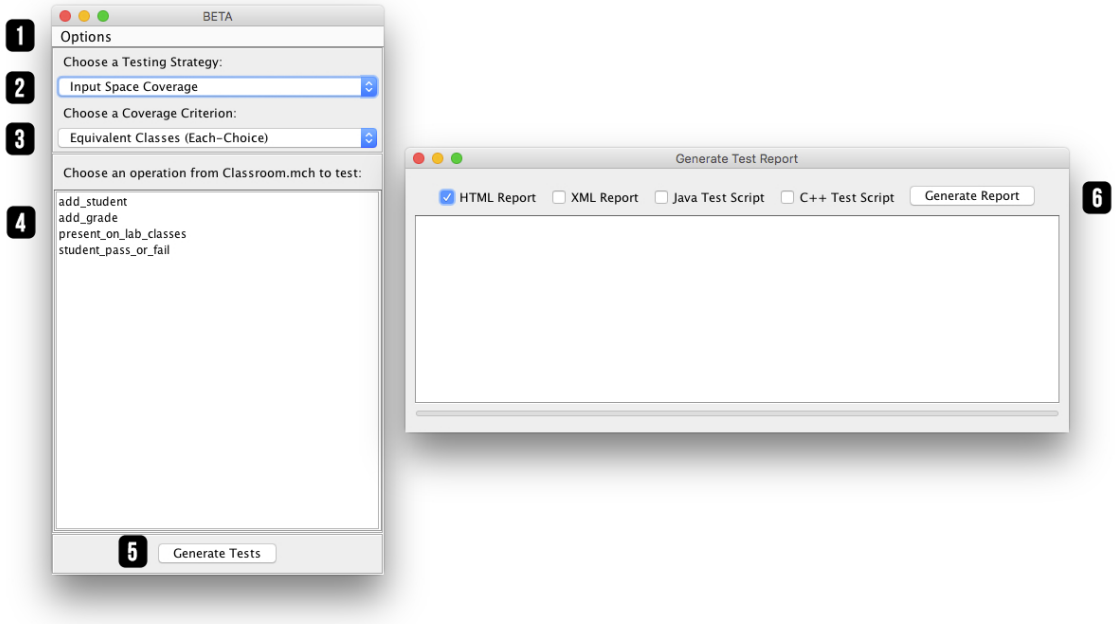

Figure 5.8: An overview of the BETA User Interface.

The window on the left is the main interface that greets the user when he opens the tool. This window has an Options menu (1) that has three items:

1. *Load Machine*: which loads a machine on the interface so the user can generate tests for one of its operations. In the figure example, the Classroom machine is currently loaded;

2. *Concretize Test Data*: which opens a window that allows the user to load a XML test case specification generated by BETA so it can concretize the test data on the specification;

3. *Settings*: which opens the settings window where the user can modify BETA configuration parameters.

In the main interface the user can also choose the *Testing Strategy* (2) that he intends to use (*Input Space Partitioning* or *Logical Coverage*). Once he chooses a testing strategy

the tool will present the coverage criteria for this strategy in the *Choose Coverage Criterion* menu (3). On (4) the user will find the list of operations of the machine currently loaded on BETA. Once he chooses an operation to test, he can click on the *Generate Tests* button (5) and the tool will present him with the test generation window (6). In this window, he can choose the format in which he wants to generate the test cases (HTML and XML test cases specifications, and Java and C test scripts).

5.2.1 Implementation details and architecture

The BETA tool is implemented in Java[9]. We chose to implement the tool in Java due to the language's portability, performance, easy environment to install, and because it would be easier to integrate the tool with some components that it uses. Java's portability was a great benefit to have, making it easier to have BETA running in the three major platforms (Windows, Linux and OS X). An overview of BETA's current architecture is presented in Figure 5.9.

The tool's code is organized in six packages that have different responsabilities. It also relies on three external components to peform some tasks.

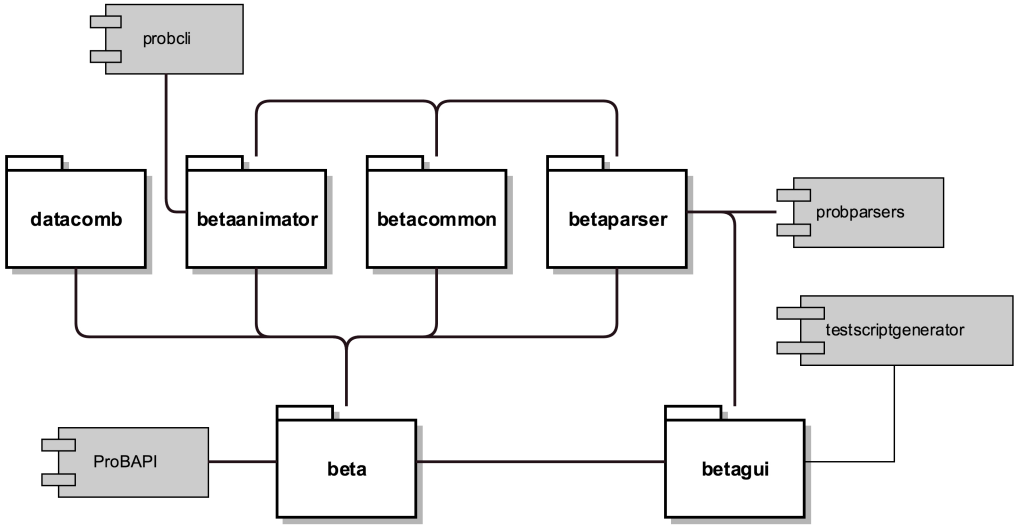

Figure 5.9: An overview of the BETA tool architecture. White folder represent BETA packages and gray blocks represent external components.

The packages are organized as follows:

– *beta*: contains all core classes of the tool, such as partition strategies, coverage criteria, preamble calculation, oracle evaluation, test data concretization and test report generation classes;

[9]Java's webpage: http://www.java.com/

– *datacomb*: contains all combination criteria used by the input space partitioning strategy;

– *betaanimator*: contains all the classes responsible for the animation of the auxiliar B machines used to generate test data;

– *betacommon*: contains all utility classes used by different packages;

– *betaparser*: contains our parser that extends the ProB parser for the B notation, including classes and methods that make it easier to extract information from B machines;

– *betagui*: contains the classes that implement BETA's user interface.

These packages interact with external components from ProB and a test script generator module that was developed separately:

– *probcli*: the tool uses the ProB command line interface to perform tasks like machine animations and preamble calculations;

– *probparsers*: our parser extends the ProB parser for the B notation, adding some extra functionalities to it;

– *ProBAPI*: the tool uses methods from the ProB API to perform animations (as an alternative to the *probcli* interface) and to obtain oracle data;

– *testscriptgenerator*: the tool uses a separate module that was implemented by [Souza Neto, 2015] to generate the executable test scripts in Java and C.

More information about the tool can be found on its website: www.beta-tool.info. There, you can find instructions on how to install the tool, its user guide, some materials used in our case studies, and B models to experiment with. BETA is free to use, open source and it works on Windows, Linux and OS X. The repository with the tool's source code can be found on: https://github.com/ernestocid/beta1.

5.3 Comments on the evolution of BETA

The BETA approach and tool have been under development since the author's masters course. This section has the objective of clarifying 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 improvements were the following:

– One of the first things we worked on during this doctorate was on improving the strategies used to create partitions using the input space partitioning criteria. We

reviewed the B Method's grammar and defined, for each type of characteristic that can be extracted from a model, how it would be partitioned using both equivalence classes and boundary value analysis. As a result, we produced a guide that is available on BETA's website and on Appendix A that presents our partition strategies[10];

– To complement the input space partitioning criteria already supported, we decided to implement logical coverage criteria in the approach. 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 and would also help to study and compare the effectiveness of different criteria;

– As part of this doctorate, the author spent a year doing a Ph.D. internship at Heinrich-Heine University in Düsseldorf, Germany. During this internship, we worked with the STUPS group[11] on integrations between BETA and ProB. BETA now uses ProB's Java API to interact with its kernel. Thanks to this integration, BETA can now use features from ProB that it could not use before. This integration also helped to improve the architecture of the tool making it easier to maintain its code base;

– Before this doctorate, the process to obtain oracle data for the test cases had to be done manually. The process required the user to animate the original model and simulate the test case execution using an animation tool (such as ProB). Now, using features from ProB's API, this process is automated;

– We also implemented different types of oracle evaluation strategies [Souza Neto and Moreira, 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*;

– 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. Unfortunately, in some cases "set" methods are not available and can not be implemented. So, we proposed and implemented the strategy presented in Section 5.1.6 which generates preambles using methods already available in the implementation;

– We also tackled the test data concretization problem during this doctorate. Initially, BETA was only capable of generating abstract test data for its test case specifications. So, we developed a solution to this problem that uses a B Method feature called *gluing*

[10] Input Space Partitioning with BETA: http://beta-tool.info/files/BETA_IPS_Map_03_08_2015.pdf
[11] STUPS group webpage: https://www3.hhu.de/stups/wiki/index.php/STUPS_Group

invariant to translate the abstract data intro concrete data that can be used to code the concrete test cases. This solution is explained with further details in Section 5.1.8;

– Another improvement was made in the last step of the BETA approach: the implementation of the concrete test cases. 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].

Chapter 6

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. Additionally, they helped to evaluate the applicability of BETA on problems with different characteristics. The results of each case study were used as a reference for the improvement of the approach and its supporting tool.

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, five case studies were performed: the *General Door Controller (GDC)*, the *FreeRTOS*, the *Lua API*, and the *c4b and b2llvm* case studies. We also revisited some old experiments in a fifth case study.

This chapter is organized as follows:

- **Section 6.1** presents a review of the case studies performed during the author's master's [Matos, 2012]. These case studies were our first experiments with the BETA approach, and they focused mainly on evaluating the initial versions of the approach and tool (BETA 0.3). Regarding the test cases generated, they concentrated on quantitative aspects, measuring the number of test cases produced by each coverage criterion supported by the initial approach. These case studies are presented here only for historical reasons; no in-depth details are discussed in this chapter. The remaining sections present the case studies performed during this doctorate;

- **Section 6.2** presents our third case study. In this case study, we used BETA to generate tests for the Lua programming language API based on a B model proposed by [Moreira and Ierusalimschy, 2013]. This case study was the first to evaluate BETA's test generation process as a whole, from the design to implementation and execution of test cases. For the first time, in this case study, we also tried to evaluate the quality of the test cases using code coverage metrics. The Lua API case study was performed with BETA 1.1;

- **Section 6.3** presents our fourth case study which had the objective to evaluate the

ability of BETA generated test cases to identify possible faults on code generators. In this case study, we selected a set of B machines and generated test cases using BETA to test the code generated by C4B and b2llvm, two code generation tools for the B Method. This case study was performed with BETA 1.2;

– **Section ??** presents our final case study. In this case study, we revisited the experiments performed for the code generators case study and evaluated how recent enhancements in the approach and the tool improved the quality of the test cases generated by BETA. The experiments were organized in two new rounds and the results were compared with the round of experiments performed in the original case study (Section 6.3). The experiments included comparisons between two versions of the tool: 1.2 and 2.0 (the current version). In these last experiments, we also used mutation testing to evaluate the quality of the test cases for the first time.

To clarify the differences between versions of the tool used in the case studies, Table 6.1 presents the list of features introduced in each version.

		Versions			
		0.3	1.1	1.2	2.0
Features	IPS Coverage	✓	✓	✓	✓
	Logical Coverage			✓	✓
	Plain Text Test Case Specifications	✓			
	HTML Test Case Specifications		✓	✓	✓
	XML Test Case Specifications		✓	✓	✓
	Executable Test Scripts		✓	✓	✓
	Automatic Oracle				✓
	Oracle Strategies				✓
	Automatic Preamble				✓
	Test Data Randomization				✓
	Test Data Concretization				✓
	ProB API Integration				✓

Table 6.1: BETA versions and corresponding features.

6.1 Previous Case Studies

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 was 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 [Souza, 2009]. 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 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].

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.

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

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]. It 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 predicates on conditional statements as characteristics 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 combination 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 combination 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.

6.2 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. 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

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

23 abstract machines.

The objective of this case study was to evaluate BETA with more complex specifications, which could 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. The case study was performed using BETA 1.1, and consequently only used test cases generated using input space coverage criteria. Also, this was the first case study to evaluate the test script generation feature. Additionally, it provided a way to refine the Lua API model and find discrepancies between the model and its implementation.

6.2.1 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 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, which, in total, encompassed 12 modules.

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 could not 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 combination 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 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 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: *statement* and *branch* coverage.

Tables 6.2 and 6.3 present an overview of the coverage obtained by the tests generated for each API function that was tested. They present respectively the numbers for statement and branch coverage. Table 6.2 shows the number of statements for each function and the number of statements covered by the tests generated for each combination criterion. Table 6.3 shows the number of branches for each function and the number of branches covered by the tests generated for each combination criterion. All test cases were generated using equivalence classes since the characteristics of the operations under test did not contain any

[8] http://libcheck.github.io/check/

intervals.

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. For example, the branch coverage for the *lua_arith* operation was very low (66.6% for the best criterion) because it uses a *switch-case* statement and the test cases generated using input space partitioning could not cover all the branch possibilities. 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.

Table 6.2: Code coverage for the functions in the Lua API

Function	Lines	All Combinations	Each Choice	Pairwise
lua_checkstack	10	10 (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%)

6.2.2 Final Remarks and Conclusions

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

Table 6.3: 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%)

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.

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 do not expect 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. By the time this case study was performed, BETA assumed that it was possible to modify state variables directly (using a "set" function, for example). This case study confirmed 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 implemented in BETA's most current version and it uses ProB's constraint-based testing capabilities to find out what 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 was 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). This feature is also implemented in BETA's current version.

– *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 tests generated by BETA – using equivalent classes and boundary value analysis – could not reach. The support to logical coverage criteria helped with this problem and increased the code coverage for situtations that had multiple execution paths (more details in Section **??**).

Ultimately, the case study also helped to improve the model of the API. During the implementation of the test cases, we found scenarios where the model did not represent the actual implementation of the API correctly. These discrepancies were reported so the model could be improved.

6.3 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 6.1 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 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 will not 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 test 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.

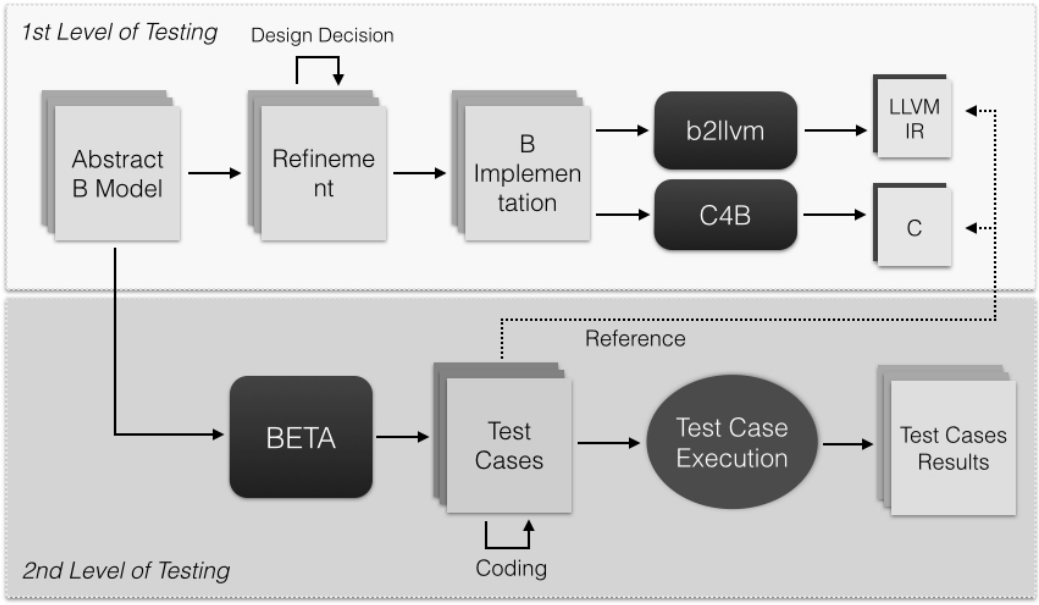

Figure 6.1: Testing Strategy for C4B and b2llvm

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 (the case study was performed using BETA 1.2, which did not support test data concretization at the time). 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, which was then executed. After the execution, the test 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 consistent with its respective abstract machine (this can happen if the refinement proofs are not complete).

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.

6.3.1 Results

The same set of B models was used to test both code generation tools. Table 6.4 presents some information on these models, the tests, and the obtained results for each coverage

criterion, 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 6.4: Overview of the model-based tests generated by BETA.

(a) **B Modules**				(b) **EC/BV**			(c) **ACC**			(d) **CoC**		
N.	Machine	Lines	Ops	TCs	b2llvm	C4B	TCs	b2llvm	C4B	TCs	b2llvm	C4B
1	Counter	51	4	10	10	10	8	8	8	6	6	6
2	Swap	18	3	3	3	3	6	6	6	1	1	1
3	Calculator	48	6	10	10	10	26	26	26	6	6	6
4	Wd	27	3	5	5	5	5	5	5	4	4	4
5	Prime	10	1	4	4	4	5	5	5	3	3	3
6	Division	12	1	2	2	2	3	3	3	1	1	1
7	Team	36	2	3	3	3	7	7	7	4	4	4
8	BubbleSort	35	1	1	1	1	1	1	1	1	1	1
9	TicTacToe	67	3	13	13	13	12	12	12	8	8	8
10	Fifo	22	2	2	0*	2	4	0*	4	2	0*	2
11	Calendar	40	1	2	0*	2	25	0*	25	25	0*	25
12	ATM	28	3	3	0*	3	7	0*	7	2	0*	2
13	Timetracer	47	6	7	0*	4	11	0*	6	9	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. In the end, the models covered all the production rules of the B0 notation.

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.

6.3.2 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 do not 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 judge 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 of import 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 could evaluate the improvements made

in the last steps of the approach (Research Question 1). It was also important to evaluate the improvements we made in the supported coverage criteria and to show their effectiveness (Research Question 2).

6.4 Final Experiments

In our final experiments, we focused on an empirical study to analyze the BETA approach and tool, focusing on quantitative and qualitative aspects of the test cases generated. We revisited some of the models that had already been used in previous experiments [Moreira et al., 2015], and performed the entire testing process for them, from test case design to execution and evaluation of the results. The results were evaluated using metrics such as statement and branch coverage, and mutation analysis [Andrews et al., 2005].

In these final experiments, we also evaluated recent changes and new features of the BETA approach and tool. For example, for the first time, we performed experiments to assess the logical coverage criteria implemented by the approach. Another aspect that was evaluated in this new round of experiments was how the recently implemented test data randomization feature helps to improve the quality of the test cases. These experiments helped us to measure how the different enhancements made on BETA improved the quality of the test cases generated. It also contributed to compare the effectiveness of the implementation of each coverage criterion supported by the approach, identifying which ones produce better test suites.

Some of the questions we addressed in these final experiments – which are related to the research questions **RQ2** and **RQ3** presented in Chapter 1 – were:

– ***Q1****: How do the BETA implementation of the partitioning strategies and combination criteria differ in quantity (size of the test suites) and quality (code coverage and mutation analysis) of the results and how recent improvements influence these results?*

  We evaluated the results obtained by BETA when using input space partitioning criteria, checking the number of test cases generated, both feasible and infeasible, and also measuring the coverage obtained by these test cases. We also used mutation testing to evaluate the ability of the generated test cases to detect faults in the code.

– ***Q2****: How do the BETA implementation of the logical coverage criteria differ in quantity (size of the test suites) and quality (code coverage and mutation analysis) of the results?*

  For the first time, we performed an investigation on the quality of test cases generated using logical criteria, in the same way we did before for the test cases generated using

input space partitioning.

– ***Q3****: How do the BETA implementation of the input space partitioning and logical coverage criteria differ from each other? Which criteria provide better coverage and are more capable of detecting faults?*

  Additionally, we also wanted to make comparisons between the results obtained by the two techniques, checking the differences in the number of test cases generated, code coverage and mutation score (mutation testing is often used as a reference to evaluate other criteria due to the high quality of its results in spite of its higher costs [Ammann and Offutt, 2010, Delamaro et al., 2007]).

To answer these questions, we submitted BETA to new experiments to evaluate the input space partitioning techniques (and its recent improvements) and the logical coverage criteria. Additionally, we experimented with the test data randomization feature, comparing results obtained when using test data randomization and when not using it. The experiments were organized in three rounds: *original*, *improved* and *randomization*. They were organized as follows:

– *Original experiment*: on this first round, we consider the original experiments presented in Section 6.3. They were performed with BETA 1.2, which lacked some of the current features. Its results were the basis for a series of improvement requirements which led to a new version of the BETA approach and tool. The results of this round of experiments were published in [Matos et al., 2015];

– *Improved experiment*: this second round was performed with the main goals of evaluating the influence of the improvements of BETA 2.0 partitioning criteria on its results, the quantity and quality of BETA logical coverage criteria implementation and to provide material for a comparison between the two family of criteria;

– *Randomization experiment*: BETA 2.0 supports test data randomization as an option when generating test case data. This option was activated on this third round to explicitly analyze the influence of randomization on the quality of the generated test cases with respect to the improved experiment where it was not used. To evaluate this particular feature, each test case generation was carried out 5 (five) times and the average code and mutation coverage obtained by the test cases was used.

To perform the statement and branch coverage analysis, we used the *GCOV*[9] and *LCOV*[10] tools. GCOV is a test coverage analysis software for the *C* programming language, and LCOV

[9]GCOV's website: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[10]LCOV's website: http://ltp.sourceforge.net/coverage/lcov.php

is a graphical front-end for GCOV that generates rich coverage reports in *HTML*. The version of GCOV used in our experiments was the one integrated into *Xcode 7.2*[11], an integrated development environment (IDE) for software development on *OS X*, and the version of LCOV used was *1.2*.

Mutation analysis was performed to evaluate the capability of the test cases generated using BETA to detect faults. Mutation analysis works with fault simulation by injecting syntactic changes in the program under test [Ammann and Offutt, 2010]. The program with a syntactic change is called *mutant*. A mutant is said to be *killed* when a test can differentiate it from the original artifact. If a mutant can not be distinguished from the original artifact, it is called *equivalent*. The ratio between the number of killed mutants and the number of non-equivalent mutants is called *mutation score* and it is used to measure the quality of a test set. Since mutation analysis works with fault simulation, their results provide reliable information about the test case effectiveness [Andrews et al., 2005]. The tool we used to generate mutants in our experiments is called *Milu* [Jia and Harman, 2008] (version 3.2[12]). The equivalent mutants were identified manually, and the execution and analysis were automated by scripts.

6.4.1 Results

The abstract machines of the 13 modules from Table 6.4 were submitted to the BETA tool, and it was capable of generating test cases for the three experiments. For the *original* experiment, BETA generated test cases using all partitioning strategies and all combination criteria, but it was not able to generate the test cases using the BVS strategy for two modules (*Division* and *Prime*). This issue has been fixed in the version 2.0 of the tool. For the *improved* and *randomization* experiments, BETA 2.0 was able to generate test cases for all 13 B modules using all input space partitioning and logical coverage strategies. In this case study, some B modules use numerical ranges, leading to different results when BVS partitioning strategy was used instead of ECS.

Considering the total of test cases generated for all 13 modules, Figure 6.2 presents a bar chart with the amount of test cases generated by BETA in the *original* experiment, and Figure 6.3 presents a bar chart with the amount of test cases generated by BETA in the *improved* and *randomization* experiments. Since the test data randomization feature does not have any influence on the number of test scenarios generated, the same amount of test cases was generated by BETA in the *improved* and *randomization* experiments. In both charts, it is possible to see the amount of infeasible test cases, and positive and negative feasible test cases generated by BETA for every testing strategy.

In the *original* experiment, the bar chart (Figure 6.2) shows that the BVS strategy gen-

[11]Xcode's website: https://developer.apple.com/xcode/
[12]Milu's website: http://www0.cs.ucl.ac.uk/staff/y.jia/Milu/

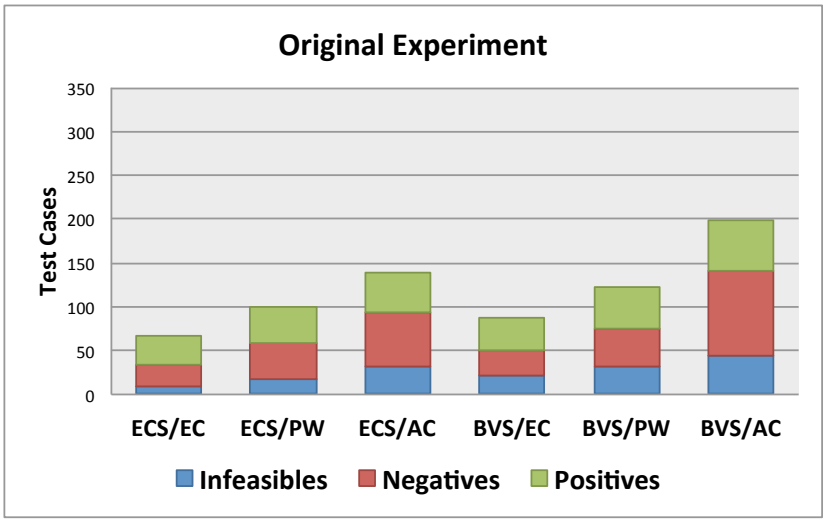

Figure 6.2: Amount of the test cases generated by BETA in the *original* experiment. The bar charts show the amount of infeasible test cases and negative and positive feasible test cases generated by BETA.

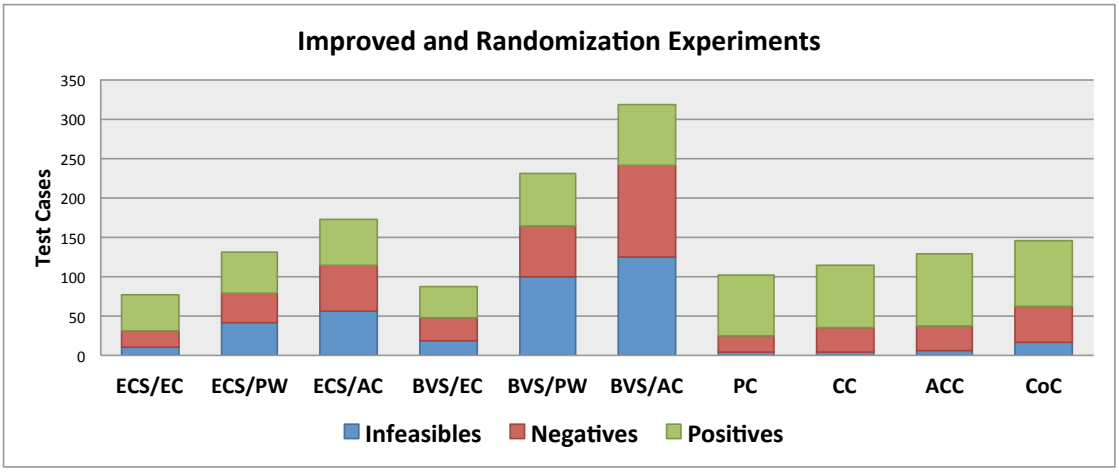

Figure 6.3: Amount of the test cases generated by BETA in the *improved* and *randomization* experiments. The bar charts show the amount of infeasible test cases and negative and positive feasible test cases generated by BETA.

erates the largest amount of test cases. It also shows that the combination criterion AC generated the largest amount of test cases, followed by PW and EC. In the *improved* and *randomization* experiments, the bar chart (Figure 6.3) shows that the same pattern of the *original* experiment was followed for the amount of test cases generated, but the number of test cases generated by BETA 2.0 was substantially higher than the *original* experiment. It is a result of improvements made in the tool. The bar chart (Figure 6.3) also shows that with logical coverage criteria BETA generates, in general, fewer test cases than with input space partitioning. Moreover, it is possible to see that, as expected, CoC generates more tests than any other logical criterion, followed by ACC, CC, and PC.

To evaluate the tests generated by BETA, we used statement coverage, branch coverage,

and mutation analysis as metrics. In this evaluation, only the test results for the C code correctly generated by C4B were considered (the *Timetracer* module was not considered because C4B did not generate correct code for it). The results of statement and branch coverage in the three experiments are presented in Figures 6.4 and 6.5. The figures show bar charts with the averages of the statement and branch coverage, respectively, obtained with the input space partitioning criteria and, for the *improved* and *randomization* experiments, also the logical coverage criteria. With these bar charts, we can see that the results obtained with input space partitioning criteria improved, in general, in BETA 2.0. It is also possible to see in the charts that the results obtained with logical coverage criteria are close to or better than those obtained with input space partitioning.

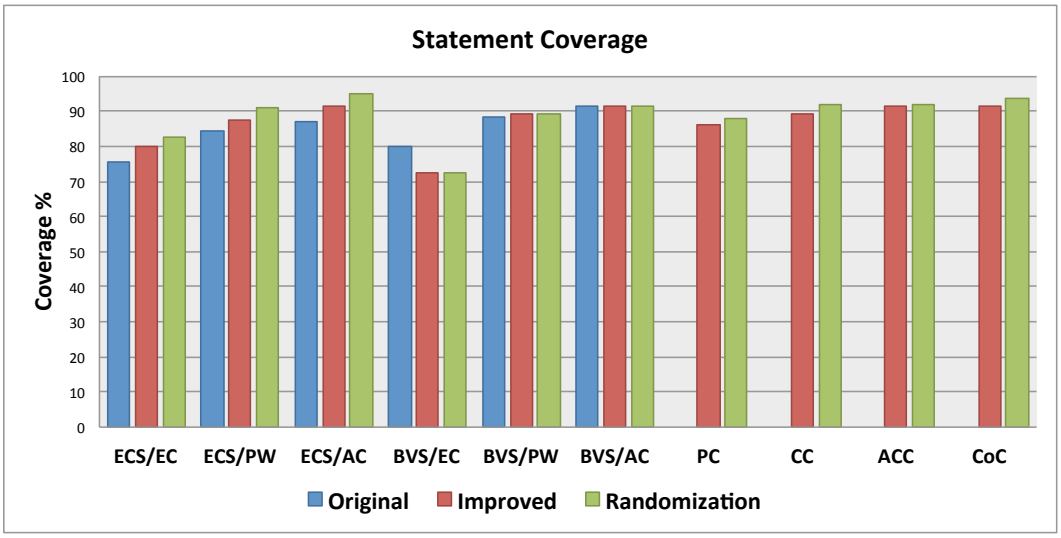

Figure 6.4: Bar charts showing the average of the statement coverage results obtained with the tests generated by BETA in each each experiment.

Mutation analysis was performed to evaluate the capabilities of the test cases to detect faults. Table 6.5 presents the mutation analysis results in the three experiments. The table shows information about the modules: the number of operations (Op.), the number of non-equivalent mutants (Mut.), and the mutation score achieved by the tests generated with input space partitioning and logical coverage criteria in each experiment. The last row of Table 6.5 presents an average of the mutation scores obtained in the *original*, *improved* and *randomization* experiments. For the *original* experiment, the average does not include the modules *Division* and *Prime* because the BETA tool did not generate tests with the strategy BVS for these two modules.

Figure 6.6 presents a bar chart with the mutation score obtained by the tests generated by BETA with each input space partitioning criterion and logical coverage criterion in each experiment. Looking at this chart, it is possible to see that the changes made in the input space partitioning implementation in BETA 2.0 lead to an improvement in the tests generated with ECS concerning the original experiment using BETA 1.2. The BVS partitioning

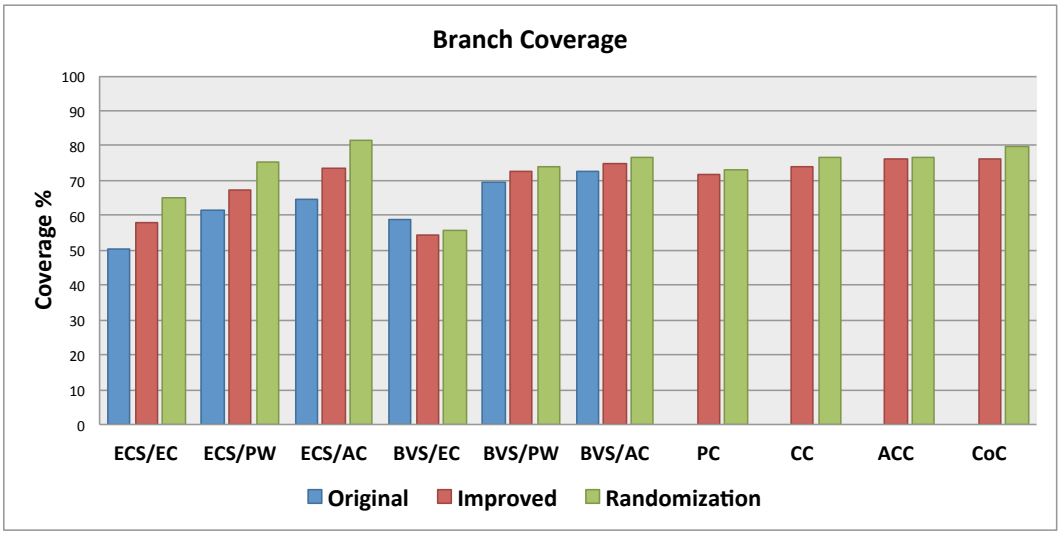

Figure 6.5: Bar charts showing the average of the branch coverage results obtained with the tests generated by BETA in each experiment.

strategy, however, presented slightly worse results in average. This unexpected result is discussed on section 6.4.2. As we saw in the statement and branch coverage analysis, the results obtained with logical coverage criteria in the mutation analysis are close to or better than those obtained with input space partitioning.

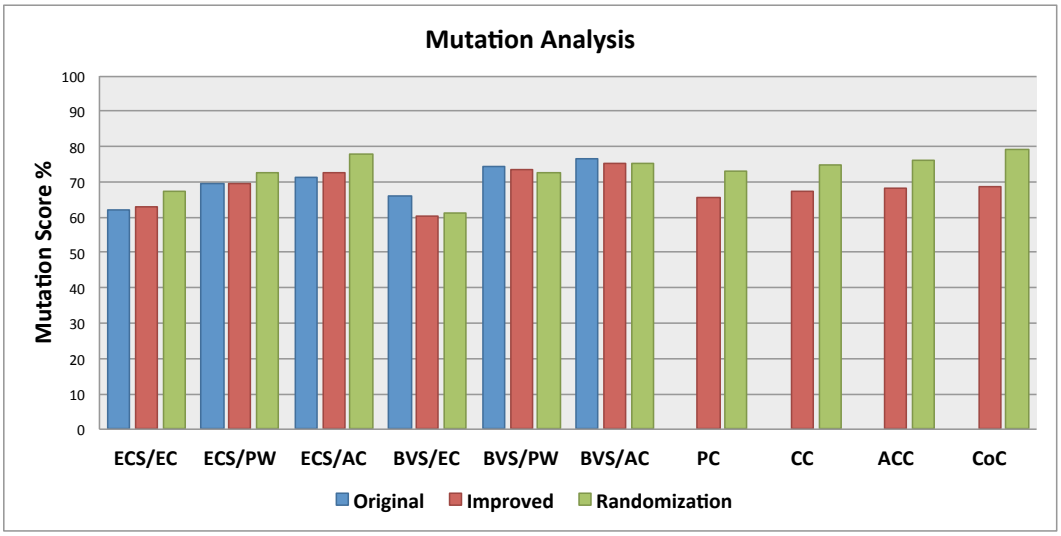

Figure 6.6: Bar charts with the average of the mutation scores obtained with the tests generated by BETA in each each experiment.

6.4.2 Final Remarks and Conclusions

The first question (**Q1**) that we addressed in our last experiments is concerned with the quantitative and qualitative aspects of the tests generated by BETA using input space partitioning. Regarding the number of test cases generated, the results of the three experiments

Table 6.5: Mutation Analysis Results

Modules			Exp.	Percentage of mutants killed - Mutation Score %									
				Equivalent Classes			Boundary Value Analysis			Logical			
Name	Op.	Mut.		EC	PW	AC	EC	PW	AC	PC	CC	ACC	CoC
ATM	3	11	*Orig.*	81.8	81.8	81.8	81.8	81.8	81.8	–	–	–	–
			Imp.	81.8	81.8	81.8	81.8	81.8	81.8	81.8	81.8	81.8	81.8
			Rand.	92.7	90.9	87.3	92.7	90.9	87.3	94.5	90.9	96.4	90.9
Sort	1	123	*Orig.*	89.4	89.4	89.4	89.4	89.4	89.4	–	–	–	–
			Imp.	89.4	89.4	89.4	89.4	89.4	89.4	91.1	91.1	91.1	91.1
			Rand.	90.7	90.7	91.1	90.7	90.7	91.1	91.1	91.1	91.1	91.1
Calculator	6	120	*Orig.*	47.5	47.5	47.5	74.2	74.2	74.2	–	–	–	–
			Imp.	46.5	46.6	46.6	71.6	71.6	71.6	46.6	46.6	46.6	46.6
			Rand.	58.7	50.8	56.8	78	75	76.2	55.3	59.5	56	51.8
Calendar	1	67	*Orig.*	9	19.4	19.4	25.4	35.8	35.8	–	–	–	–
			Imp.	80.6	94	94	0	35.8	35.8	94	94	94	94
			Rand.	80.6	94	94	0	35.8	35.8	94	94	94	94
Counter	4	87	*Orig.*	41.4	85.1	85.1	41.4	94.2	94.2	–	–	–	–
			Imp.	41.4	85.1	85.1	41.4	94.2	94.2	85.1	85.1	85.1	85.1
			Rand.	45.5	86.8	87.1	43.2	95.2	97.5	71.9	80.7	75.4	78.4
Division	1	29	*Orig.*	31	31	31	–	–	–	–	–	–	–
			Imp.	31	31	31	89.6	96.5	96.5	34.5	34.5	34.5	34.5
			Rand.	59.3	73.8	68.3	89.6	96.5	96.5	53.8	54.5	43.4	74.5
Fifo	2	40	*Orig.*	90	90	90	90	90	90	–	–	–	–
			Imp.	90	90	90	90	90	90	90	90	90	90
			Rand.	90	90	90	90	90	90	90	90	90	90
Prime	1	66	*Orig.*	34.8	34.8	53	–	–	–	–	–	–	–
			Imp.	34.8	34.8	53	0	43.9	43.9	62.1	62.1	62.1	62.1
			Rand.	39.1	40.6	75.4	0	43.9	43.9	71.5	71.5	72.4	78.5
Swap	3	8	*Orig.*	100	100	100	100	100	100	–	–	–	–
			Imp.	100	100	100	100	100	100	37.5	37.5	37.5	37.5
			Rand.	90	85	87.5	90	85	87.5	97.5	80	87.5	95
Team	2	89	*Orig.*	68.5	68.5	68.5	68.5	68.5	68.5	–	–	–	–
			Imp.	68.5	68.5	68.5	68.5	68.5	68.5	68.5	68.5	68.5	68.5
			Rand.	68.8	69	69	68.8	69	69	67.9	73.3	82.7	80.2
TicTacToe	3	764	*Orig.*	0	21.9	40.3	0	20.4	40.3	–	–	–	–
			Imp.	0	21.9	40.3	0	20.4	40.3	3.1	23.3	36.8	40
			Rand.	0	20.5	40.5	0	18.1	41.3	3.1	23.4	36.3	40.9
Wd	3	68	*Orig.*	91.2	91.2	91.2	91.2	91.2	91.2	–	–	–	–
			Imp.	91.2	91.2	91.2	91.2	91.2	91.2	91.2	91.2	91.2	91.2
			Rand.	90	88.8	86.5	90	88.8	86.8	86.5	87.6	88.8	86.5
Total/Average	**30**	**1,472**	***Orig.***	**61.9**	**69.5**	**71.3**	**66.2**	**74.5**	**76.5**	**–**	**–**	**–**	**–**
			Imp.	**63**	**69.5**	**72.6**	**60.3**	**73.6**	**75.3**	**65.5**	**67.1**	**68.3**	**68.5**
			Rand.	**67.1**	**72.6**	**77.8**	**61.1**	**72.5**	**75.2**	**73.1**	**74.7**	**76.2**	**79.1**

correspond to what should be theoretically expected. BVS generates more tests than ECS when numerical ranges are used in the B module; the combination criterion AC generates substantially more tests than PW, which in turn produces more tests than EC. This pattern was followed by the number of infeasible tests and feasible positive and negative tests. These results were not different from those obtained in the first case studies performed in [Matos and Moreira, 2012]. Since the AC criterion combines all partitions, it is expected that it generates more infeasible cases.

We used statement and branch coverage, and mutation analysis as metrics to evaluate the quality of the tests generated by BETA using input space partitioning criteria. For the original experiment, the resulting coverage followed the quantity patterns obtained for the tests generated using input space partitioning criteria. In other words, the results achieved with BVS strategy were better than those achieved with ECS strategy, and the results obtained with AC combination criterion were better than those obtained with PW, which in turn, were better than those obtained with EC. Coverage results did not show the same significant

variations between the partitioning strategies and the combination criteria as observed in the quantities, however. Results also showed that, using PW, it is possible to achieve very close results to those obtained using AC, with lower costs, since it generates fewer tests. The results obtained for BETA, although derived from a restricted set of models, are consistent with the common knowledge that advocates that PW provides a good benefit-cost ratio [Ammann and Offutt, 2010].

The obtained results for the original experiment, revealed that the test cases generated using input space partitioning criteria, even in the best cases, could not achieve all statements and decisions of the system under test, which can affect the testing process. Mutation analysis reinforced these coverage results. Even in the best case, the mutation score was not much higher than 75% on average.

These results indicated that it was necessary to improve BETA to generate more effective tests and provided guidelines on how to do it. This feedback led to the implementation of BETA 2.0. To evaluate BETA 2.0, we performed the *improved* and the *randomization* experiments.

Regarding input space partitioning in the *improved* and *randomization* experiments, all but one configuration provided better coverage results using BETA 2.0. The results showed that the improvements and corrections made in the input space partitioning implementation, leading to a better treatment of some B constructs, led to an increase on size and quality of the test suites in the improved and randomization experiments. The improved experiment attained better coverage than the original experiment (best results: increase of 7% on average branch coverage for ECS partitioning). Additionally, the randomization experiment, in general, obtained better coverage than the improved experiment with the same number of tests (best results: increase of 8% on average branch coverage for ECS partitioning), showing that the randomization feature improves testing results.

The exception was the test suite generated using BVS partitioning strategy and EC combination criterion. The average results obtained using the ECS partitioning strategy were better than those obtained using BVS, an unexpected outcome. In fact, BETA generates, as expected, more test case configurations for BVS than for ECS each time there are intervals in the definition of the operation under test. However, the combination implementation of BETA may still lead to a high number of infeasible configurations. Because BVS generates some very specific requirements corresponding to the borders of the intervals, chances are most of the new configurations end up being infeasible, leading to this unexpected behavior. This issue is agravated by the fact that infeasible (empty) blocks corresponding to numbers greater than MAXINT or smaller than MININT are being generated. The solution to this problem and a general improvement on coverage results passes through a better treatment of infeasible combinations of otherwise satisfiable requirements (and, of course, non generation of infeasible blocks).

The abstraction gap between the abstract model and its implementation, may also hinder

coverage results, as it happened with the *TicTacToe* model. Its implementation considered explicitly each winning situation for each TicTacToe player, leading to a much more complex control flow than the corresponding abstract operation specification. This is a common issue in black box testing, however. A possible solution would be to generate test cases from implementation modules, which are more similar to the actual code.

The quantitative and qualitative aspects of the BETA's logical coverage implementation are the target of the second question (**Q2**). Considering the number of test cases generated by BETA, the results for the logical coverage criteria correspond to what should be theoretically expected. The CoC criterion produced a (slightly) higher number of test cases, followed by ACC, CC and PC. This pattern was not strictly followed by the number of positive test cases, since the ACC criterion generated slightly more positive tests than CoC. When using the ACC criteria, the approach can sometimes produce redundant positive test cases due to some formulas that, although different, specify the same test scenarios (or in other words, we may end up with a set of extra formulas that cover the same test case scenario).

The logical coverage criteria test cases were capable of identifying the same faults that were identified by the input space partitioning criteria. This result showed that the test cases generated using logical coverage were effective in those verification and validation processes. The tests were also evaluated using statement and branch coverage analysis, and mutation analysis. As was observed in the number of test cases, the CoC criterion obtained better results, followed by ACC, CC and PC. However, there were no significant variations among them, with very close results when compared with each other, this happened because the models used in the experiment were simple, in general. Except for the *TicTacToe* model, all models have predicates with only one or two clauses. On the limit, when all predicates contain a single clause, all of the logical coverage criteria collapse into the same criterion: *Predicate Coverage* [Ammann and Offutt, 2010]. The absence of more complex predicates is a tendency in programming style as observed in [Durelli et al., 2015], but whether this trend is present in the case of B formal models is a matter which needs further analysis. Again, the randomization feature presented positive influence on coverage results, of around 8% on average mutation coverage for the different logical criteria.

The third question (**Q3**) analyzes the differences, quantitative and qualitative, between the BETA implementation of the input space partitioning criteria and logical coverage criteria. Regarding the number of test cases generated, using input space partitioning, BETA may produce a considerably larger number of test cases than with logical coverage criteria. The biggest difference is the number of infeasible test cases and negative feasible test cases. Input space partitioning promotes combinations between positive blocks (which respect the precondition) and negative blocks (which does not respect the preconditions), that favors a higher number of infeasible and negative feasible test cases.

Considering only positive test cases, using logical coverage, BETA generated a number of test cases that was close to or greater than the ones generated using input space parti-

tioning. In the current case studies, as mentioned before, only the positive test cases were implemented and executed. Consequently, statement coverage, branch coverage, and mutation analysis results of the tests generated using logical coverage criteria were close to or slightly better than the results of the tests generated using input space partitioning criteria. There were, however, no significant differences considering the best results.

The greatest advantage in the use of logical coverage is the efficiency of the test cases generated. As we mentioned, the test suites generated using logical coverage and input space partitioning achieved similar results. However, the number of test cases generated using logical coverage was significantly lower (Figure 6.3), implying that is test suites are more efficient than the test suites generated using input space partitioning criteria.

Another significant finding from our last experiments is that the use of randomization in the generation of test data increased all the coverage results and mutation scores. This conclusion shows the positive influence of test data randomization in the generation of test suites.

Chapter 7

Conclusions and Future Work

In this thesis, we continued with the work presented on [Matos, 2012], extending and improving the BETA approach and tool. We added several features that enhanced the approach, and also reviewed some old features, fixing known issues and updating them to make BETA more effective and efficient. Another objective of this thesis was to validate BETA through more complex case studies and also experiment with the test cases it generates to assess their quality, checking for coverage levels and capability to identify faults.

BETA has been under development for six years now. The tool was validated through case studies with reasonable levels of complexity, which mimicked real project environments. The case studies also assessed the usability of the approach and tool, and they have shown that BETA can easily be adopted in B Method development environments. The tool is no more just an academic prototype; it is a mature tool that is ready to be used in industry-level projects.

The tool supports all common steps of model-based testing, providing automation for the entire test generation process. Also, in the code generators case study, we demonstrated how the approach could be a valuable addition to the B Method verification process. It can add an extra layer of verification, which checks if the system code implements what was specified in the original abstract model.

In [Marinescu et al., 2015], the authors present a recent review of model-based testing tools for requirement-based specification languages. Looking at this review and our own research of the state of the art, we can establish some strong points for BETA when compared with other model-based testing tools for the B Method, and even for other notations. There are not many projects in the field that use well-defined test selection criteria as we did with BETA. Also, they do not cover all aspects of the test generation process as BETA does. For example, there are not many tools that deal with test data concretization or that supports diverse coverage criteria as our tool.

Another strong point for BETA is its availability. There are not many tools with the same purpose that are publicly available; not only to be used but also to be extended. BETA is open source and can be easily adapted or improved by other researchers.

Taking into consideration the research questions presented in Chapter 1, we achieved the following results:

Research Question 1: *How can we improve the test generation process, mainly the last steps of the approach?*

Our case studies pointed out that most of the improvements required to improve BETA should be made in the last steps of the test generation process. With that in mind, in this thesis we worked on the following points:

– We implemented a test script generation module that is capable of translating BETA test case specifications into concrete executable test cases [Souza Neto, 2015]. The scripts are written in Java and C, and still require a few adaptations before they can be executed, but they already save a lot of the effort needed to implement the concrete test cases. An example of test script generated by BETA is presented in Appendix B;

– We also proposed and implemented an strategy to generate oracle data, and implemented strategies to perform oracle verifications. BETA is now capable of animating the original model with test data to identify what is the expected behavior for a particular test case according to the initial specification. Furthermore, the oracle verification process now supports different verification strategies that can be used during the execution of the test cases;

– Another feature proposed and implemented during this thesis was the automatic calculation of preambles for BETA test cases. Thanks to the integration with ProB, BETA is now capable of identifying sequences of operations from the model that can put the system in the state required to execute a particular test case;

– Ultimately, we defined and implemented a test data concretization strategy that is capable of translating the abstract test data from the test case specifications into concrete test data that can be used by the executable test scripts.

Thanks to all these recently implemented features, BETA now automates all the steps of the model-based test generation process we proposed. Performing these steps by hand was very time-consuming and liable to human errors. With the current level of automation, BETA saves a reasonable amount of the effort required to design and code the test cases.

Research Question 2: *How the testing criteria supported by the approach can be improved?*

BETA's first version only supported input space partitioning as a coverage criterion. In this thesis, we reviewed the old strategies used to generate tests using input space coverage

and, after some improvements, we reduced the number of infeasible test cases generated by the technique and also improved the quality of the partitions it generates. An overview of the updated strategies is presented in Appendix A.

Besides, we implemented a new set of logical coverage criteria in the approach. BETA now supports *Predicate Coverage*, *Clause Coverage*, *Combinatorial Clause Coverage* and *Active Clause Coverage* as criteria to generate test cases. This work showed how flexible the approach can be to add new coverage criteria. The support to logical coverage is also important because many certification standards for safety-critical systems require this type of coverage.

Our experiments presented in Chapter 6, showed the efficiency of the coverage criteria that BETA already supports. Overall, they achieved good results regarding code coverage and mutation analysis. Still, we are planning to work on improvements for this set of coverage criteria and implement new ones in the future.

Research Question 3: *How can we measure the quality of the test cases generated by BETA?*

In this thesis, we also presented new case studies used to evaluate the BETA approach and tool. We generated test cases based on a specification of the Lua API and also used BETA to evaluate two code generation tools for the B Method (more details in Chapter 6). Through these case studies, we were able to evaluate BETA's test generation process as a whole. They also helped us to identify what were the points of the approach that needed improvement. Besides, they showed that BETA is mature enough to be used in more complex projects.

Ultimately, in our last experiments, we focused on assessing the quality of the test cases generated by the approach. We decided to use statement and branch coverage together with mutation testing to do this evaluation. Our experiments showed that, in the best cases (considering the criterion that produced the best results), BETA is capable of achieving:

- an average statement coverage of 95% (maximum of 100% and minimum of 80%), using input space partitioning, the All-Combinations criterion, and the test data randomization feature;
- an average branch coverage of 86.1% (maximum of 100% and minimum of 37.5%), using logical coverage, the combinatorial coverage criterion, and the test data randomization feature;
- an average mutation score of 79.1% (maximum of 95% and minimum of 34.5%), using logical coverage, the combinatorial coverage criterion, and the test data randomization feature.

These results showed that there are still some aspects that need to be improved so the approach can achieve better outcomes. We already identified some points for improvements, and we discuss them in Section 7.1. More data on the experiments and case studies results can be found on Chapter 6.

So far, the contributions of our research resulted in the following 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.

– In [Matos et al., 2015] we presented an empirical evaluation of the BETA approach. We presented the results obtained after the Lua API and the code generators case studies, and also discussed about our first experiments using mutation testing. This paper won a *best paper* award.

Besides these four publications, there is also the master's work of [Souza Neto, 2015], which is part of the BETA project as a whole. His work was responsible for conducting several experiments using BETA, providing valuable feedback to improve the approach, and also for the development of the test script generation module. We were also invited to submit an extended version of the [Matos et al., 2015] paper to the *Journal of the Brazilian Computer Society*. The paper was already submitted and is currently under revision.

BETA is currently available for download on its website http://www.beta-tool.info. In the website, the user can find instructions on the installation and usage of the tool and some additional material such as information about the case studies and B machines to experiment with. The tool is free to use and open source.

7.1 Future Work

This doctorate has come to an end, but there are still some opportunities for future work in the BETA project. Bellow, we present some ideas that should be implemented in the future:

- *Reduce the number of infeasible test cases:* the current experiments showed that, during the test case generation process, we lose test cases due to infeasibility. Some of the test formulas contain contradictions that make it impossible to generate test data that satisfy them. In some cases, it would be possible to turn these infeasible test formulas into feasible test formulas with smarter combination algorithms. An important point of improvement would then be to enhance our combination algorithms to avoid the combination of contradictory clauses, resulting in fewer infeasible test cases;

- *Perform more in-depth experiments using new features:* during our case studies we identified several features that should be implemented to improve the BETA test generation process. During this thesis, we defined and implemented all of the identified features, but we believe it is still necessary to perform more in-depth experiments using them, gathering more data on how they improved the BETA approach;

- *Generate test cases from B implementation modules:* since B implementations represent more closely the tested code, the ability to generate test cases from them would reduce the abstraction gap and make the process of refinement of the test cases easier. This improvement could result in less time required to adapt and code the concrete test cases. As a side effect, it would also make us lose the advantages of generating the test cases from the original abstract specifications;

- *New improvements to the partition strategies:* some improvements in the way partitions are created during the test generation process have already been made. We believe we can still improve the partitioning process using knowledge acquired during the latest experiments, and using techniques proposed in related work, such as [Cristiá et al., 2014], which enumerates a number of partitioning strategies for different constructs used in formal models;

- *Test data concretization for refinements with multiple steps:* currently, the test data concretization feature only works for refinements with a single step. We can still improve this feature to support refinements with multiple levels;

- *Add support to other coverage criteria:* this thesis showed that BETA is flexible enough to implement new coverage criteria. Adding more techniques to the set of supported coverage criteria would make the approach even more diverse and appealing to test engineers;

– *Add support to other formal notations criteria:* an interesting future work would be to adapt the approach to support different types of formal notations. A strong candidate to be the next formal notation supported would be Event-B. The current architecture does not support language plugins, but we believe that, with moderate effort, it is possible to improve the tool implementation to incorporate new notations easily. It is just a matter of implementing an intermediate notation, which source languages could be translated to, and have a constraint solver that can solve formulas in this intermediate notation. ProB already applies a similar idea to support different notations.

Appendices

Appendix A

In this appendix we present how input space partitioning is performed for the B models. The tables present the types of *characteristics* that can be found in a B specification and how they are partitioned into blocks, using both *equivalence classes* and *boundary value analysis*. These tables are just a summary of how the process works. More detailed information can be found on a report published in the tool's website [Matos and Moreira, 2015].

Table 1: Equivalence Classes Input Space Partition overview for non-typing characteristics.

Characteristic	Example	Block 1	Block 2	Block 3
Strictly greater than	$Expression > Expression$	$x > y$	$x < y$	$x = y$
Greater than or equal	$Expression \geq Expression$	$x \geq y$	$x < y$	
Strictly less than	$Expression < Expression$	$x < y$	$x > y$	$x = y$
Less than or equal	$Expression \leq Expression$	$x \leq y$	$x > y$	
Equals	$Expression = Expression$	$x = y$	$x \neq y$	
Unequal	$Expression \neq Expression$	$x \neq y$	$x = y$	
Negation	$not(Predicate)$	$not(P)$	P	
Implication	$Predicate \Rightarrow Predicate$	$P_1 \Rightarrow P_2$	$not(P_1 \Rightarrow P_2)$	
Equivalence	$Predicate \Leftrightarrow Predicate$	$P_1 \Leftrightarrow P_2$	$not(P_1 \Leftrightarrow P_2)$	
Universal Quantifier	$\forall list\_ident.(Predicate \Rightarrow Predicate)$	$\forall v_1, v_2.(P_1 \Rightarrow P_2)$	$not(\forall v_1, v_2.(P_1 \Rightarrow P_2))$	
Existential Quantifier	$\exists list\_ident.(Predicate)$	$\exists v_1, v_2.(P)$	$\exists v_1, v_2.(not(P))$	
Belongs	$Expression \in Expression$	$exp_1 \in exp_2$	$exp_1 \notin exp_2$	
Does not belong	$Expression \notin Expression$	$exp_1 \notin exp_2$	$exp_1 \in exp_2$	
Includes	$Expression \subset Expression$	$exp_1 \subset exp_2$	$exp_1 \not\subset exp_2$	
Does not include	$Expression \not\subset Expression$	$exp_1 \not\subset exp_2$	$exp_1 \subset exp_2$	
Strictly includes	$Expression \subseteq Expression$	$exp_1 \subseteq exp_2$	$exp_1 \not\subseteq exp_2$	
Does not strictly include	$Expression \not\subseteq Expression$	$exp_1 \not\subseteq exp_2$	$exp_1 \subseteq exp_2$	

Table 2: Equivalence Classes Input Space Partition overview for typing characteristics.

Characteristic	Example	Block 1	Block 2	Block 3
Is a natural number	$id \in NAT$	$x \in NAT$	$x \notin NAT$	
Is a natural number different than zero	$id \in NAT1$	$x \in NAT1$	$x \notin NAT1$	
Is an integer	$id \in INT$	$x \in INT$		
Is a boolean	$id \in BOOL$	$x \in BOOL$		
Is a number in the rage	$id \in Expression..Expression$	$x \in Exp_1..Exp_2$	$x \in MININT..Exp_1 - 1$	$x \in Exp_2 + 1..MAXINT$
Is a member of abstract set	$id \in ID$	$x \in SET$		
Is a member of	$id \in id$	$x \in y$	$x \notin y$	
Is a total function	$id \in Simple\_set --> Simple\_set$	$x \in SET --> SET$		
Is a partial function	$id \in Simple\_set +-> Simple\_set$	$x \in SET +-> SET$		
Is an injective function	$id \in Simple\_set >+> Simple\_set$	$x \in SET >+> SET$		
Is a total surjective function	$id \in Simple\_set -->> Simple\_set$	$x \in SET -->> SET$		
Is a partial surjective function	$id \in Simple\_set +->> Simple\_set$	$x \in SET +->> SET$		
Is a bijective function	$id \in Simple\_set >->> Simple\_set$	$x \in SET >->> SET$		
Is a member of the enumerated set	$id \in \{Simple\_term+\}$	$x \in \{t_1, t_2, ..., t_n\}$	$x \notin \{t_1, t_2, ..., t_n\}$	
Is a string	$id \in STRING$	$x \in STRING$		
Is a subset of an abstract set	$id \subset ID$	$id \subset ID$		
Is a subset of the set of booleans	$id \subset BOOL$	$id \subset BOOL$		

Table 3: Equivalence Classes Input Space Partition overview for typing characteristics.

Characteristic	Example	Block 1	Block 2
Is a subset of the natural numbers	$id \subset NAT$	$id \subset NAT$	$id \not\subset NAT$
Is a subset of the positive natural numbers	$id \subset NAT1$	$id \subset NAT1$	$id \not\subset NAT1$
Is a subset of the integer set	$id \subset INT$	$id \subset INT$	
Is a subset of a range	$id \subset Expression..Expression$	$id \subset Exp_1..Exp_2$	$id \not\subset Exp_1..Exp_2$
Is a subset of	$id \subset id$	$id \subset id$	$id \not\subset id$
Is a subset strict of an abstract set	$id \subseteq ID$	$id \subseteq ID$	$id \not\subseteq ID$
Is a subset strict of the set of booleans	$id \subseteq BOOL$	$id \subseteq BOOL$	
Is a subset strict of the natural numbers	$id \subseteq NAT$	$id \subseteq NAT$	$id \not\subseteq NAT$
Is a subset strict of the positive natural numbers	$id \subseteq NAT1$	$id \subseteq NAT1$	$id \not\subseteq NAT1$
Is a subset strict of the integer set	$id \subseteq INT$	$id \subseteq INT$	
Is a subset strict of a range	$id \subseteq Expression..Expression$	$id \subseteq Exp_1..Exp_2$	$id \not\subseteq Exp_1..Exp_2$
Isasubsetstrictof	$id \subseteq id$	$id \subseteq id$	$id \not\subseteq id$
Is equal to abstract set	$id = ID$	$id = ID$	$id \neq ID$
Is equal to a term	$id = Term$	$id = T$	$id \neq T$
Is equal to array	$id = \{X \mapsto A, ..., Z \mapsto C\}$ or $id = SET * \{Term\}$	$id = \{X \mapsto A, ..., Z \mapsto C\}$ or $id = S * \{T\}$	$id \neq \{X \mapsto A, ..., Z \mapsto C\}$ or $id \neq S * \{T\}$
Is equal to range	$id = Expression..Expression$	$id = Exp_1..Exp_2$	$id \neq Exp_1..Exp_2$
Is equal to the set of natural numbers	$id = NAT$	$id = NAT$	$id \neq NAT$
Is equal to the set of positive natural numbers	$id = NAT1$	$id = NAT1$	$id \neq NAT1$
Is equal to the set of integers	$id = INT$	$id = INT$	$id \neq INT$

Table 4: Boundary Value Analysis Input Space Partition overview for non-typing characteristics.

Characteristic	Block 1	Block 2	Block 3
Strictly greater than	$x > y$	$x < y$	$x = y$
Greater than or equal	$x \geq y$	$x < y$	
Strictly less than	$x < y$	$x > y$	$x = y$
Less than or equal	$x \leq y$	$x > y$	
Equals	$x = y$	$x \neq y$	
Unequal	$x \neq y$	$x = y$	
Negation	$not(P)$	P	
Implication	$P_1 \Rightarrow P_2$	$not(P_1 \Rightarrow P_2)$	
Equivalence	$P_1 \Leftrightarrow P_2$	$not(P_1 \Leftrightarrow P_2)$	
Universal Quantifier	$\forall v_1, v_2.(P_1 \Rightarrow P_2)$	$not(\forall v_1, v_2.(P_1 \Rightarrow P_2))$	
Existential Quantifier	$\exists v_1, v_2.(P)$	$\exists v_1, v_2.(not(P))$	
Belongs	$exp_1 \in exp_2$	$exp_1 \notin exp_2$	
Does not belong	$exp_1 \notin exp_2$	$exp_1 \in exp_2$	
Includes	$exp_1 \subset exp_2$	$exp_1 \not\subset exp_2$	
Does not include	$exp_1 \not\subset exp_2$	$exp_1 \subset exp_2$	
Strictly includes	$exp_1 \subseteq exp_2$	$exp_1 \not\subseteq exp_2$	
Does not strictly include	$exp_1 \not\subseteq exp_2$	$exp_1 \subseteq exp_2$	

Table 5: Boundary Value Analysis Input Space Partition overview for typing characteristics.

Characteristic	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6
Is a natural number	$x = -1$	$x = 0$	$x = 1$	$x = MAXINT - 1$	$x = MAXINT$	$x = MAXINT + 1$
Is a positive natural number	$x = 0$	$x = 1$	$x = 2$	$x = MAXINT - 1$	$x = MAXINT$	$x = MAXINT + 1$
Is an integer	$x \in INT$					
Is a boolean	$x \in BOOL$					
Is a number in the rage	$x = Exp_1 - 1$	$x = Exp_1$	$x = Exp_1 + 1$	$x = Exp_2 - 1$	$x = Exp_2$	$x = Exp_2 + 1$
Is a member of abstract set	$x \in SET$					
Is a member of	$x \in y$	$x \notin y$				
Is a total function	$x \in S --> S$					
Is a partial function	$x \in S +-> S$					
Is an injective function	$x \in S >+> S$					
Is a total surjective function	$x \in S -->> S$					
Is a partial surjective function	$x \in S +->> S$					
Is a bijective function	$x \in S >->> S$					
Is a member of the enumerated set	$x \in \{t_1, t_2, ..., t_n\}$	$x \notin \{t_1, t_2, ..., t_n\}$				
Is a string	$x \in STRING$					
Is a subset of an abstract set	$id \subset ID$					
Is a subset of the set of booleans	$id \subset BOOL$					

Table 6: Boundary Value Analysis Input Space Partition overview for typing characteristics.

Characteristic	Block 1	Block 2
Is a subset of the natural numbers	$id \subset NAT$	$id \not\subset NAT$
Is a subset of the positive natural numbers	$id \subset NAT1$	$id \not\subset NAT1$
Is a subset of the integer set	$id \subset INT$	
Is a subset of a range	$id \subset Exp_1..Exp_2$	$id \not\subset Exp_1..Exp_2$
Is a subset of	$id \subset id$	$id \not\subset id$
Is a subset strict of an abstract set	$id \subseteq ID$	$id \not\subseteq ID$
Is a subset strict of the set of booleans	$id \subseteq BOOL$	
Is a subset strict of the natural numbers	$id \subseteq NAT$	$id \not\subseteq NAT$
Is a subset strict of the positive natural numbers	$id \subseteq NAT1$	$id \not\subseteq NAT1$
Is a subset strict of the integer set	$id \subseteq INT$	
Is a subset strict of a range	$id \subseteq Exp_1..Exp_2$	$id \not\subseteq Exp_1..Exp_2$
Isasubsetstrictof	$id \subseteq id$	$id \not\subseteq id$
Is equal to abstract set	$id = ID$	$id \neq ID$
Is equal to a term	$id = T$	$id \neq T$
Is equal to array	$id = \{X \mapsto A, ..., Z \mapsto C\}$ or $id = S * \{T\}$	$id \neq \{X \mapsto A, ..., Z \mapsto C\}$ or $id \neq S * \{T\}$
Is equal to range	$id = Exp_1..Exp_2$	$id \neq Exp_1..Exp_2$
Is equal to the set of natural numbers	$id = NAT$	$id \neq NAT$
Is equal to the set of positive natural numbers	$id = NAT1$	$id \neq NAT1$
Is equal to the set of integers	$id = INT$	$id \neq INT$

Appendix B

In this appendix we present an example of code generated by BETA's test script generation module (the example was extracted from [Souza Neto, 2015]) and an example of XML test case specification also generated by the tool.

Example of a BETA generated test script

Listing 1: Example of test script generated by BETA

```
1  public class TicTacToeBlueMoveTest {
2   private TicTacToe tictactoe;
3
4   @Before
5   public void setUp() throws Exception {
6    tictactoe = new TicTacToe();
7   }
8
9   @After
10  public void checkInvariant () throws Exception
11   // Predicate 'bposn <: 1..9' can't be automatically translated
12   // Predicate 'rposn <: 1..9' can't be automatically translated
13   // Predicate 'bposn /\ rposn = {}'' can't be automatically
14   // translated
15   if(!(tictactoe.getTurn() != null /* turn : Player */)){
16    fail("The invariant 'turn : Player' was unsatisfied");
17   }
18  }
19
20  @Test
21  public void testCase1 {
22   int [] rposn = {};
23   tictactoe.setRposn(rposn);
24
25   int [] bposn = {};
26   tictactoe.setBposn(bposn);
```

```
27
28    Player turn = blue;
29    tictactoe.setTurn(turn);
30
31    intpp=1;
32
33    tictactoe.BlueMove(pp);
34
35    int[] bposnExpected = {1};
36    assertEquals(tictactoe.getBposn(), bposnExpected);
37
38    int[] rposnExpected = {};
39    assertEquals(tictactoe.getRposn(), rposnExpected);
40
41    Player turnExpected = red;
42    assertEquals(tictactoe.getTurn(), turnExpected);
43   }
44  }
```

Example of a XML test case specification

Listing 2: Example of XML test case specification

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <test-suite>
3    <machine-name>Classroom</machine-name>
4    <machine-invariant>
5      <invariant-clause>grades : (students +-&gt; 0..5)</invariant-
         clause>
6      <invariant-clause>has_taken_lab_classes : (students +-&gt;
         BOOL)</invariant-clause>
7      <invariant-clause>students &lt;: all_students</invariant-
         clause>
8    </machine-invariant>
9    <operation-under-test>student_pass_or_fail</operation-under-test
       >
10   <testing-strategy>Logical Coverage</testing-strategy>
11   <coverage-criteria>Combinatorial Clause Coverage</coverage-
       criteria>
12   <oracle-strategy>
13     <state-variables>true</state-variables>
14     <return-variables>true</return-variables>
15     <invariant-ok>true</invariant-ok>
16   </oracle-strategy>
17   <test-cases>
```

```
18      ...
19      <test-case>
20        <id>5</id>
21        <existential-formula>#student, students,
             has_taken_lab_classes, grades.(students &lt;:
             all_students &amp; grades : (students +-&gt; 0..5) &amp;
             student : dom(has_taken_lab_classes) &amp;
             has_taken_lab_classes(student) = TRUE &amp; student : dom
             (grades) &amp; grades(student) &gt; 3 &amp;
             has_taken_lab_classes : (students +-&gt; BOOL) &amp;
             student : students)</existential-formula>
22        <formula>students &lt;: all_students &amp; grades : (
             students +-&gt; 0..5) &amp; student : dom(
             has_taken_lab_classes) &amp; has_taken_lab_classes(
             student) = TRUE &amp; student : dom(grades) &amp; grades(
             student) &gt; 3 &amp; has_taken_lab_classes : (students
             +-&gt; BOOL) &amp; student : students</formula>
23        <test-inputs-as-formula>students = {st1} &amp; student = st1
              &amp; has_taken_lab_classes = {(st1|-&gt;TRUE)} &amp;
             grades = {(st1|-&gt;4)}</test-inputs-as-formula>
24        <preamble/>
25        <isNegative>false</isNegative>
26        <state-variables>
27          <variable>
28            <identifier>students</identifier>
29            <values>
30              <value>st1</value>
31            </values>
32          </variable>
33          <variable>
34            <identifier>has_taken_lab_classes</identifier>
35            <values>
36              <value>(st1|-&gt;TRUE)</value>
37            </values>
38          </variable>
39          <variable>
40            <identifier>grades</identifier>
41            <values>
42              <value>(st1|-&gt;4)</value>
43            </values>
44          </variable>
45        </state-variables>
46        <operation-parameters>
47          <parameter>
48            <identifier>student</identifier>
49            <values>
50              <value>st1</value>
```

```
51            </values>
52          </parameter>
53        </operation-parameters>
54    </test-case>
55    ...
56  </test-cases>
57  </test-suite>
```

Appendix C

In this appendix we present the algoritmns used to generate combinations of blocks for the input space partitioning criteria. The algorithms are written in Java-like pseudocode.

Extracting input parameters

– **Input:**

operation : Operation

The operation under test.

– **Output:**

inputSpace : Set<String>

A set of variables and parameters that are represented by strings.

– **Algorithm:**

The algorithm is straightforward, it input space parameters from the list of operation parameters (line 4), from *if* substitutions (line 6), and from *any* substitutions (line 7). Additionally, it searches for additional variables (line 5) in the machine under test (lines 15-17), and related variables mentioned in the machine's invariant, including invariants from other modules (lines 19-36).

Listing 3: Pseudocode for the algorithm to find input parameters for the operation under test

```
1  getOperationInputSpace(operation) {
2    inputSpace = new Set<String>
3
4    inputSpace.addAll(operation.getParameters())
5    inputSpace.addAll(getRelatedVariables(operation))
6    inputSpace.addAll(getIfCommandVariables(operation))
7    inputSpace.addAll(getAnyCommandVariables(operation))
8
9    return inputSpace
10 }
11
```

```
12 getRelatedVariables(operation) {
13   relatedVars = new Set<String>
14
15   if (operation.getMachine().getVariables() != null) {
16     relatedVars.addAll(getMachine().getVariables().getAll())
17   }
18
19   if (operation.getPrecondition() != null) {
20     // Add all variables mentioned in the precondition
21     relatedVars.addAll(operation.getPrecondition().getVariables())
22
23     // Search for related variables on the invariant (including
24     // invariants from other modules)
25     relatedVars.addAll(getRelatedVarsOnInvariants(relatedVars))
26
27     // Search for related variables on machine definitions
28     relatedVars.addAll(getRelatedVarsOnDefinitions(relatedVars))
29
30     // Remove set names that might be added
31     removeSets(relatedVars)
32
33     // Remove garbage that might be added during
34     // the process: constants, return variables, etc.
35     removeGarbageVariables(relatedVars)
36   }
37
38   return relatedVars
39 }
```

Extracting characteristics

– **Input:**

operation : Operation

The operation under test.

– **Output:**

characs : Set<Characteristic>

A set of *Characteristic* objects.

– **Algorithm:**

The algorithm extracts general characteristics from the invariant and precondition (line 5), and from conditional substitutions (line 8). The search for general charac-

teristics includes invariants from imported modules (lines 28-30) and characteristics related to variables mentioned in the body of the operation (lines 35-38). For characteristics from conditional substitutions (lines 43-58), the algorithm also considerers parallel and nested conditionals, independently (lines 49-55).

Listing 4: Pseudocode for the algorithm to find characteristics for the operation under test

```
1  getOperationCharacteristics(operation) {
2    characs = new Set<Characteristic>
3
4    // Gets characteristics from precondition and invariant
5    characs.addAll(getGeneralCharacteristicsClauses(operation))
6
7    // Gets characteristics from conditional statements
8    characs.addAll(getConditionalCharacteristics(operation))
9
10   // If there is any definition used in a predicate,
11   // they are replaced with their values
12   characs = expandDefinitions(characs)
13
14   return characs
15 }
16
17 getGeneralCharacteristicsClauses(operation) {
18   characs = new Set<Characteristic>
19
20   // Searches for characteristics in the precondition
21   for (Characteristic c : getCharacteristicsFromPrecondition()) {
22     if (!setContains(c, characs))
23       characs.add(c)
24   }
25
26   // Searches for characteristics in the invariant
27   // (including invariants from other modules)
28   for (Characteristic c : getCharacteristicsFromInvariant()) {
29     if (!setContains(c, characs))
30       characs.add(c)
31   }
32
33   // Searches for characteristics that mention
34   // operation body variables
35   for (Characteristic c : getCharacteristicsForOpBodyVars()) {
36     if (!setContains(c, characs))
37       characs.add(c)
38   }
39
40   return characs
```

```
41  }
42
43  getConditionalCharacteristics(operation) {
44    conditions = new Set<Characteristic>
45    opBody = operation.getOperationBody()
46
47    // If operation body has a parallel substitution,
48    // considers all substitutions
49    if (opBody instanceof ParallelSubstitution) {
50      for (Substitution s : opBody.getSubstitutions()) {
51        conditions.addAll(getConditionsFromConditionals(s))
52      }
53    } else {
54      conditions.addAll(getConditionsFromConditionals(opBody))
55    }
56
57    return conditions
58  }
```

Each-choice algorithm

– **Input:**

$blockLists : List < List < Block >>$

A list of block lists. Each block list contains blocks created for one characteristic.

– **Output:**

$combinations : Set < List < Block >>$

A set of block lists. Each block list contains a combination of blocks from different characteristics.

– **Algorithm:**

The algorithm can go two ways. If the list of blocks has blocks for just one characteristic (line 3), it will generate one combination for each block of this characteristic (lines 9-18). If there are more than one characteristic (line 5), the first blocks for each characteristic will be combined together, then, the second blocks for each characteristic will be combined, and so on (lines 28-39). If the number of blocks for the characteristics are not equal, once a characteristic reaches its last block, this last block will be used in the following combinations (line 35).

Listing 5: Pseudocode for the Each-choice combination algorithm

```
generateCombinations(blockLists) {
  if (blockLists.size < 2)
    return createCombinationsForSingleCharacteristic(blockLists)
  else
    return createCombinationsForManyCharacteristics(blockLists)
}

createCombinationsForSingleCharacteristic(blockLists) {
  combinations = new Set<List<Block>>
  blocksForSingleCharacteristic = blockLists.get(0)

  for (Block b : blocksForSingleCharacteristic) {
    combination = new List<Block>
    combination.add(b)
    combinations.add(combination)
  }

  return combinations
}

createCombinationsForManyCharacteristics(blockLists) {
  combinations = new Set<List<Block>>

  // Obtain number of blocks for the
  // characteristic with more blocks
  maxOfBlocks = calculateMaxNumberOfBlocks(blockLists)

  for (blockIndex = 0; blockIndex < maxOfBlocks; blockIndex++) {
    combination = new List<Block>

    for (List<Block> characteristicBlocks : blockLists) {
      if (blockIndex < characteristicBlocks.size)
        combination.add(characteristicBlocks.get(blockIndex))
      else
        combination.add(getLastBlock(characteristicBlocks))
      }

      combinations.add(combination)
  }

  return combinations
}
```

Pairwise algorithm

– **Input:**

$blockLists : List < List < Block >>$

A list of block lists. Each block list contains blocks created for one characteristic.

– **Output:**

$combinations : Set < List < Block >>$

A set of block lists. Each block list contains a combination of blocks from different characteristics.

– **Algorithm:**

The algorithm can go two ways. If the list of blocks has blocks for just one characteristic (line 4), it generates one combination for each block of this characteristic. If there is more than one characteristic, the algorithm initializes doing the combination of all pairs for the first two characteristics (line 6 and lines 19-29). If there are only two characteristics (line 9), it returns the initial combinations (lines 10-11), if there are more than two, it proceeds to create pairs with blocks from the remaining characteristics (line 13). The algorithm then proceeds (lines 38-53) using the remaining characteristics (line 41) to grow the combinations horizontally (lines 42-44) and vertically (lines 46-48). During the horizontal growth (lines 59-84), the algorithm combines the initial pairs with blocks from other characteristics until all characteristics are covered. In the process, the algorithm also keeps track of uncovered pairs. These uncovered pairs are passed to the vertical growth function (lines 88-114) which creates combinations until all uncovered pairs are covered. Our algorithm is an implementation of the one proposed by [Lei and Tai, 1998]. More details can be found in the paper.

Listing 6: Pseudocode for the Pairwise combination algorithm

```
1  generateCombinations(blockLists) {
2    if (blockLists.size < 2) {
3      // calling the same method used for the Each-choice algorithm
4      return createCombinationsForSingleCharacteristic(blockLists)
5    } else {
6      initial = initialize(blockLists.get(0), blockLists.get(1))
7      combinations = new Set<List<Block>>
8
9      if (blockLists.size <= 2) {
10       combinations.addAll(initial)
11       return combinations
```

```
12      } else {
13        combinations.addAll(inParamOrderGen(initial, blockLists))
14        return combinations
15      }
16    }
17  }
18
19  // creates all pairs of blocks for the
20  // first two characteristics
21  initialize(firstCharactBlocks, secondCharactBlocks) {
22    initialCombinations = new List<List<Block>>
23
24    for (Block firstCharactBlock : firstCharactBlocks) {
25      for (Block secondCharactBlock : secondCharactBlocks) {
26        combination = new List<Block>
27        combination.add(firstCharactBlock)
28        combination.add(secondCharactBlock)
29        initialCombinations.add(combination)
30      }
31    }
32
33    return initialCombinations
34  }
35
36  // Combines the initial combinations with blocks
37  // from the remaining characteristics
38  inParamOrderGen(initialPairs, blockLists) {
39    combinations = null;
40
41    for (paramIdx = 2; paramIdx < blockLists.size; paramIdx++) {
42      uncoveredPairs = horizontalGrowth(initialPairs,
43                                        blockLists.get(paramIdx),
44                                        blockLists.subList(0, paramIdx))
45      combinations = new List<List<Block>>
46      combinations.addAll(verticalGrowth(initialPairs,
47                                         uncoveredPairs,
48                                         paramIdx))
49      initialPairs = combinations
50    }
51
52    return combinations;
53  }
54
55  // Growns the combinations horizontally, adding blocks from
56  // a different characteristic every time it is called.
57  // It also keeps track of pairs that have not being covered yet
58  // and returns them
```

```
59  horizontalGrowth(initialPairs, blocksForGrowth, prevGrownBlocks) {
60    uncoveredPairs = new PairCollection<Block>(blocksForGrowth,
61                                               prevGrownValues,
62                                               prevGrownValues.size)
63
64    if (initialPairs.size <= blocksForGrowth.size) {
65      return growCombinationsAndRemoveCoveredPairs(initialPairs,
66                                                   blocksForGrowth,
67                                                   uncoveredPairs)
68    } else {
69      uncoveredCombs = growCombinationsAndRemoveCoveredPairs(
70                                                     initialPairs,
71                                                     blocksForGrowth,
72                                                     uncoveredPairs)
73
74      for(i = blocksForGrowth.size; i < initialPairs.size; i++) {
75        newComb = uncoveredPairs.getCombsThatCoversMostPairs(
76                                               initialPairs.get(i),
77                                               blocksForGrowth)
78        initialPairs.set(i, newComb)
79        uncoveredPairs.removePairsCoveredBy(newComb)
80      }
81
82      return uncoveredCombs;
83    }
84  }
85
86  // Creates new combinations for pairs that have not been
87  // covered yet
88  verticalGrowth(combinations, uncoveredPairs, paramIdx) {
89    tempCombinations = new List<List<Block>>
90
91    for (TestPair<Block> testPair : uncoveredPairs.getPairs()) {
92      comb = createCombinationForVerticalGrowth(paramIdx)
93      comb.set(testPair.getValueAIndex(), testPair.getValueA())
94      comb.set(testPair.getValueBIndex(), testPair.getValueB())
95      tempCombinations.add(comb)
96    }
97
98    for (TestPair<Block> testPair : uncoveredPairs.getPairs()) {
99      for (List<Block> comb : tempCombinations) {
100       growingParamEqualsCombination = comb.get(testPair.
            getValueAIndex()) == testPair.getValueA()
101       otherValueIsEmpty = comb.get(testPair.getValueBIndex()) ==
            null
102
103       if (growingParamEqualsCombination && otherValueIsEmpty) {
```

```
104             comb.set(testPair.getValueBIndex(), testPair.getValueB())
105           }
106         }
107       }
108
109       newCombinations = new Set<List<Block>>
110       newCombinations.addAll(combinations)
111       newCombinations.addAll(tempCombinations)
112
113       return newCombinations
114   }
```

All-combinations algorithm

– **Input:**

$blockLists : List < List < Block >>$

A list of block lists. Each block list contains blocks created for one characteristic.

– **Output:**

$combinations : Set < List < Block >>$

A set of block lists. Each block list contains a combination of blocks from different characteristics.

– **Algorithm:**

The algorithm can go two ways. If the list of blocks has blocks for just one characteristic (line 4), it will generate one combination for each block of the characteristic. If there is more than one characteristic, the algorithm will generate all possible combinations between the blocks for each characteristic (line 6). The combination algorithm relies on a buffer to represent the combinations (line 16). Nested *for* loops are used to manipulate this buffer and create the combinations (lines 22-35). The generation process stops if it reaches a maximum number of combinations that can be set as tool configuration parameter and obtained via the method *getMaxCombinations()* (line 22).

Listing 7: Pseudocode for the All-combinations combination algorithm

```
1  generateCombinations(blockLists) {
2    if (blockLists.size < 2) {
3      // calling the same method used for the Each-choice algorithm
4      return createCombinationsForSingleCharacteristic(blockLists)
```

```
5     } else {
6       return createCombinationsForManyCharacteristics(blockLists)
7     }
8   }
9
10  createCombinationsForManyCharacteristics(blockLists) {
11    combinations = new HashSet<List<Block>>
12    totalCombinations = calculateNumberOfCombinations(blockLists)
13
14    // Stores the index for the blocks in each combination.
15    // Each element of the array represents a block index.
16    combBuffer[] = startBuffer(getParametersInputValues())
17
18    // Add first combination
19    combinations.add(getCombination(combBuffer, blockLists))
20
21    // Create and add remaining combinations.
22    for(combIndex = 0; combIndex < totalCombinations - 1 &&
          combIndex < getMaxCombinations() - 1; combIndex++) {
23      for(bufferIndex = combBuffer.length - 1; bufferIndex >= 0;
            bufferIndex--) {
24        int blockListSize = blockLists.get(bufferIndex).size
25
26        if(combBuffer[bufferIndex] < blockListSize - 1) {
27          combBuffer[bufferIndex]++
28          break
29        } else {
30          combBuffer[bufferIndex] = 0
31        }
32      }
33
34      combinations.add(getCombination(combBuffer, blockLists))
35    }
36
37    return combinations
38  }
39
40  getCombination(combBuffer, blockLists) {
41    combination = new List<Block>
42
43    for (i = 0; i < combBuffer.length; i++)
44      combination.add(blockLists.get(i).get(combBuffer[i]))
45
46    return combination;
47  }
```

Appendix D

In this appendix we present some algorithms for logical coverage. The algorithms are written in Java-like pseudocode.

Predicate Coverage

– **Input:**

The algorithm has no direct input, but it can access the operation under test using the *getOpUnderTest()* method and the extracted predicates using the *getPredicates()* method.

– **Output:**

$testFormulas : Set < String >$

A set of test formulas representing test cases.

– **Algorithm:**

If the operation has no precondition and no predicates in its body (line 7) but its machine still has an invariant (line 11), the method creates a single formula for the invariant and adds it to the set of test formulas (line 12). If there are predicates to cover (line 16), the method creates the remaining formulas. For every predicate extracted, it checks if it is the operation's precondition (line 17), and if it is, it creates special formulas for it (line 18): one formula where the precondition is *true* and another where the precondition is *false*. If the predicate is not the precondition (line 19), it creates formulas (line 20) in the format: $I \wedge PC \wedge P$ and $I \wedge PC \wedge \neg P$, where I is the invariant, PC the precondition, and P the current predicate. Ultimately, if there are any definitions used in the formulas, it replaces them with their values (line 24).

Listing 8: Pseudocode for the Predicate Coverage algorithm

```
1  getTestFormulas() {
2    testFormulas = new Set<String>
3    precondition = getOpUnderTest().getPrecondition()
4
```

```
5     // if operation has no precondition and no
6     // predicates in the body
7     if (!opHasPrecondition() && getPredicates().isEmpty()) {
8       invariant = getOpUnderTest().getMachine().getInvariant()
9
10      // but machine has invariant
11      if (invariant != null) {
12        testFormulas.add(invariant.getPredicate())
13      }
14    }
15
16    for (Predicate predicate : getPredicates()) {
17      if(opHasPrecondition() && compare(predicate, precondition)) {
18        testFormulas.addAll(createPreconditionFormulas(predicate))
19      } else {
20        testFormulas.addAll(createPredicateFormulas(predicate))
21      }
22    }
23
24    expandedTestFormulas = expandDefinitions(testFormulas)
25
26    return expandedTestFormulas
27  }
```

Clause Coverage

– **Input:**

The algorithm has no direct input, but it can access the operation under test using the *getOpUnderTest()* method and the extracted clauses using the *getClauses()* method.

– **Output:**

$testFormulas : Set < String >$

A set of test formulas representing test cases.

– **Algorithm:**

If the operation has no precondition and no predicates in its body (line 6) but its machine still has an invariant (line 10), the method creates a single formula for the invariant and adds it to the set of test formulas (line 11). If the operation has a precondition (line 15), it creates special formulas for it (line 16): one formula where all clauses are *true* and then formulas negating each precondition clause individually. If there are still clauses to cover (line 19), the method creates the remaining formulas

(lines 20-24). For every clause extracted, it creates two formulas in the following format: $I \wedge PC \wedge C$ and $I \wedge PC \wedge \neg C$, where I is the invariant, PC the precondition, and C the current clause (preconditions and invariants are only added in the formula when available). Ultimately, if there are any definitions used in the formulas, it replaces them with their values (line 27).

Listing 9: Pseudocode for the Clause Coverage algorithm

```
1  getTestFormulas() {
2    testFormulas = new Set<String>
3    precondition = getOpUnderTest().getPrecondition()
4
5    // if operation has no precondition and no clauses in its body
6    if (!opHasPrecondition() && getClauses().isEmpty()) {
7      invariant = getOpUnderTest().getMachine().getInvariant()
8
9      // but machine has invariant
10     if(invariant != null) {
11       testFormulas.add(invariant.getPredicate())
12     }
13   }
14
15   if (opHasPrecondition()) {
16     testFormulas.addAll(createTestFormulasForPrecondition(
           precondition))
17   }
18
19   for (MyPredicate clause : getClauses()) {
20     if (opHasPrecondition()) {
21       testFormulas.addAll(
             createFormulasForOtherClausesWithPrecondition(
             precondition, clause))
22     } else {
23       testFormulas.addAll(
             createFormulasForOtherClausesWithoutPrecondition(clause))
24     }
25   }
26
27   expandedTestFormulas = expandDefinitions(testFormulas)
28
29   return expandedTestFormulas
30 }
```

Bibliography

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.

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

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.

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.

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.

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

J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th Intl. Conference on Software Engineering*, 2005.

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

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.

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.

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

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.

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

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

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.

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

M. E. Delamaro, J. C. Maldonado, and M. Jino. *Introdução ao teste de software*. Editora Campus – RJ, Rio de Janeiro, Brazil, 2007.

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.

I. Dinca, F. Ipate, L. Mierla, and A. Stefanescu. Learn and Test for Event-B - 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.

V. H. S. Durelli, J. Offut, N. Li, M. E. Delamaro, J. Guo, Z. Shi, and X. Ai. What to expect of predicates: An empirical analysis of predicates in real world programs. *Journal of Systems and Software*, 2015.

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

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.

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

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

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.

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.

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

Y. Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Practice and Research Techniques. TAIC PART'08. Testing: Academic Industrial Conference*, 2008.

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

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.

Y. Lei and K. C. Tai. In-parameter-order: a test generation strategy for pairwise testing. In *Proceedings of the High-Assurance Systems Engineering Symposium*, pages 254–261, 1998.

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.

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.

R. Marinescu, C. Seceleanu, H. Le Guen, and P. Pettersson. A research overview of tool-supported model-based testing of requirements-based designs. Advances in Computers. Elsevier, 2015.

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

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

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.

E. C. B. Matos and A. M. Moreira. BETA: How we create partitions based on B machine characteristics. http://beta-tool.info/files/BETA_IPS_Map_03_08_2015.pdf, 2015. version 2.0.

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, A. M. Moreira, and J. B. Souza Neto. An empirical study of test generation with BETA. In *Proceedings of the 9th Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, 2015.

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

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.

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

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.

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.

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

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.

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.

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.

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

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.

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. 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.

S. Schneider. *B Method, An Introduction*. Palgrave, Basingstoke, 1st edition, 2001.

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.

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

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

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

M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 1st edition, 2007.

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.

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.