Complementing the B-Method with Model-Based Testing*

Ernesto C. B. de Matos

Federal Univeristy of Rio Grande do Norte
ernestocid@ppgsc.ufrn.br

Abstract. In this Ph.D project, we propose a Model-Based Testing approach to complement the B-Method development process. The approach generates unit test cases that verify the conformance between the initial abstract model and the actual implementation of the system, checking if the developed code behaves as specified in the model. The test cases are generated using classic testing techniques such as Input Space Partitioning and Logical Coverage. The tests generated by the approach are complete, executable test cases containing test data, preamble calculation, oracle evaluation and test data concretization. The approach is also supported by a tool that partially automates the test generation process.

Keywords: Model-Based Testing, B-Method

1 The Problem

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

There are many methods and processes to develop such systems. They usually involve many activities and, 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. The V&V process is known to consume a great part of the resources involved during the development of software systems. According to [16], almost 50% of the time and money required to develop a system is spent on V&V.

The task of ensuring that a system is safe, robust and error-free is a difficult one, especially for safety-critical systems that have to comply with several security standards. There are many methods and techniques that can help with software V&V. The most widely known are *Software Testing* techniques. Another practice that is especially common in the development of safety-critical systems is the use of *Formal Methods*. Formal methods and testing are V&V techniques

* This paper is about a Ph.D project that is in its final year (forth year). The project started on February, 2012 and is expected to finish on February, 2016.

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 [17]. For these reasons, there is an effort from both formal and testing communities to integrate these disciplines.

The B-Method [1] 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. The B-Method development process is presented in Figure 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 almost algorithmic representation of the initial model called B0. The B0 representation serves as basis for translation of the model to programming language code, which can be done either manually or by code generation tools.

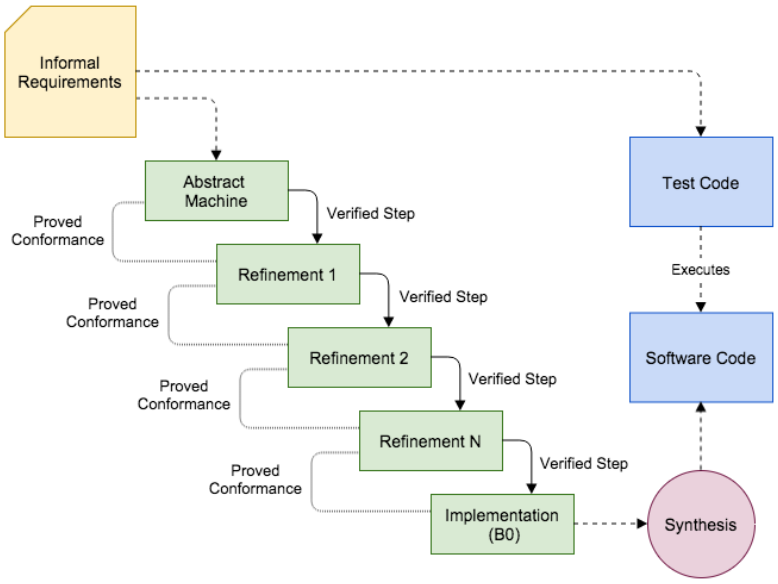

Fig. 1. The B-Method Process.

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. The testing part presented on Figure 1 is not incorporated in the B-Method's process originally, but it is still important and necessary to increase the level of trust in the developed system. In [22] the authors present some limitations of the B-Method 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 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;
- The model can be wrong in respect to its informal requirements. In essence, there is no way to prove that the informal, possibly ill-defined, user needs are correctly addressed by the formal specification;
- Modeling is performed by a human that is liable to fail;
- Refinement of an abstract specification will always require a certain degree of human input, admitting possibilities of human errors;
- The formal system underlying the method may be wrong;
- Ultimately, proofs may be faulty.

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

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

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

In this Ph.D project, we propose a Model-Based Testing approach to complement the B-Method development process. This approach is tool supported

and partially automates the generation of test cases for a software implementation based on B-Method's abstract state machines. The test cases generated by this approach try to verify the conformance between the initial abstract model and the produced source code, checking if the behavior specified in the model is actually present in the software implementation. The tests generated are unit tests that test each operation in the model individually. They are generated using classic testing techniques that include input space partitioning and logical coverage. The test cases generated by the approach are complete, executable test cases containing test data, preamble calculation, oracle evaluation and test data concretization.

2 Related Work

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 (e.g. unit, module and system testing) using different formal input models (e.g. Z, B, Alloy and VDM), and with different levels of automation [2,18,7,19,9,5,12,3,6].

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

- Their tests focus on finding problems in the model rather than on the respective implementation;
- They use *ad hoc* testing strategies instead of relying on well-established criteria from the software testing community;
- They lack on automation and tool support, something essential for the applicability of the proposed approaches.

In our work, we try to solve the problems listed above. Instead of testing the model, our focus is on testing the implementation derived from the model. Our approach generates unit tests that verify the conformance of the implementation when compared to the model. These tests can increase the trust of the developer by asserting that the code developed behaves as specified in the model. To achieve this goal, BETA generates tests using coverage criteria that have been validated through time by the software testing community, such as *Input Space Partitioning* (e.g. *Equivalent Classes* and *Boundary Value Analysis*) and *Logical Coverage* (e.g. *Predicate* and *Clause coverage*). Ultimately, the approach is automated by a tool that makes it easier to be adopted by engineers that have no background in formal methods. The tool automates the process from the design of the test cases to the generation of partial test scripts that, after some small adaptations, can be executed to test the implementation's code.

3 Proposed Solution

The proposal of this Ph.D work is a Model-Based Testing approach that complements the B-Method process, trying to ease some of the problems mentioned in

the previous sections. The approach is called BETA[1] and is partially automated by a tool that has the same name. Figure 2 presents an overview of the BETA approach and each of the steps of its test generation process.

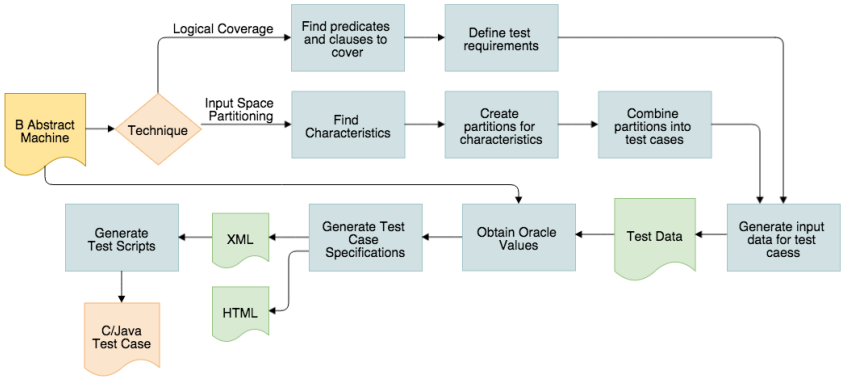

Fig. 2. An overview of the BETA approach.

The approach starts with an abstract B machine, and since it generates tests for each unit of the model individually, the process is repeated for each one of its operations.

Once an operation is chosen, the approach acts accordingly to the testing technique selected. If *Logical Coverage* is the chosen technique, it inspects the model searching for predicates and clauses to cover. Then, it creates test formulas that express situations that exercise the conditions/decisions that should be covered according to one of the supported logical coverage criteria. If *Input Space Partitioning* is the chosen technique, it explores the model to find interesting characteristics about the operation. These characteristics are constraints applied to the operation under test. After the characteristics are enumerated, they are used to create test partitions for the input space of the operation under test. Then, combinatorial criteria are used to select and combine these partitions in test cases. A test case is also expressed by a logical formula that describes the test scenario.

To obtain test input data for each of these test scenarios a constraint solver is used. Once test input data is obtained, the original model may be animated using these inputs to obtain oracle data (expected test case results). Test inputs and expected results are then combined into test case specifications that could be either in XML or HTML format. The test case specifications are used as a guide to code the concrete test cases. A separate module uses the XML specifications to generate partial test scripts that help the developer in the coding process.

[1] BETA project's website: `http://www.beta-tool.info`

All the testing criteria used in the approach are presented in [4]. For Input Space Partitioning, the approach uses *Equivalent Classes* and *Boundary Value Analysis*. To combine the obtained partitions into test cases, the approach currently supports three algorithms: *Each-Choice*, *Pairwise* and *All-Combinations*. For Logical Coverage, the approach supports *Predicate Coverage*, *Clause Coverage*, *Combinatorial Clause Coverage* and *Active Clause Coverage* (this last criterion is equivalent to *Modified Condition/Decision Coverage*).

The tool that automates the approach relies on ProB [10] as a constraint solver to perform some of the steps in the process. ProB is an animator and model checker for the B-Method. It 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. ProB can be integrated into other tools using its command-line interface or its API. BETA uses ProB's constraint solver to obtain test case data and to animate the models to obtain oracle data for the test cases evaluation.

4 Preliminary Work

In the current state of our project, we already developed a tool that automates most of the process presented in Figure 2. The tool has already been tested through several case studies that evaluated its effectiveness and usability. More details about the cases studies are found in Section 6.

The tool implements all the coverage criteria mentioned in the previous sections and is already integrated with ProB to obtain test data and oracle data for the generated test cases.

We also used ProB's constraint-based test case generator as part of our strategy to calculate preambles for our test cases. The preambles are used to put the system in the state where a particular test case needs to be executed. To find the preambles, BETA defines state goals and uses ProB to find paths that lead the system to the desired state. We already have a prototype of this feature, but it is currently not integrated into the official release of the tool because we still plan to test it more rigorously.

We also implemented a module that partially automates the generation of executable test scripts [21]. In the initial versions of the tool, it was only capable of generating test case specifications that guided the engineer in the implementation of test cases. This new module translates these specifications into executable Java and C test scripts. These scripts still need some adaptations before they can be executed, but they already decrease a good amount of the effort necessary to code the concrete test cases.

Another recent contribution to the project is a strategy for test data concretization. It uses the *glue invariant* used during the refinement of the model to find the relation between the abstract data structures used in the higher levels of abstraction and the concrete data structures used by the implementation. The strategy is not implemented yet, but we plan to implement a prototype of this feature soon, so it can be tested and integrated into the official release.

5 Expected Contributions

The main contributions expected by the end of this Ph.D project are the following:

- Develop a tool-supported approach to generate unit tests from B-Method's abstract state machines. These tests should be able to verify the conformance of the implementation – which could be derived from the model either manually or by code generation tools – when compared to the original model;
- The developed approach should implement well-defined and well-established coverage criteria developed and tested through time by the software testing community. The implemented testing criteria should increase the trust of the developers in the final product. Also, it would be interesting to implement criteria that are required by some certification standards like *Modified Condition/Decision Coverage* (MC/DC) [8]. The developed tool should also be flexible enough to allow more testing criteria to be integrated into it in the future;
- The final approach is expected to be a strategy that could be replicated in other environments. With some adaptations, the approach could be used to generate tests from different formal notations other than the B-Method. It should act as a framework to generate tests these other notations;
- As another contribution of this work, we expect to present a strategy for the concretization of test data in the B-Method environment. The strategy will not only be a theory but will also be implemented in the tool to be automatically executed;
- The approach will also support the generation of executable test scripts and will implement different oracle strategies for the evaluation of test cases. The implemented oracle strategies are based on the ones presented in [11]. The tool already implements the generation of partial test scripts, but we plan to automatically generate test scripts that require as few adaptations as possible;
- Another contribution will be the automatic calculation of test case preambles. We are already developing a strategy for the calculation of preambles using constraint solving tools. Once this strategy is implemented, the test engineer will be able to find, automatically, execution paths that lead the system to the state in which the test case has to be executed.

6 Plan for Evaluation and Validation

To validate the approach and the tool, a series of case studies was planned. Some of these case studies are already finished, but some of them still need to be performed.

Once the initial version of the approach was designed [20], we performed a case study to evaluate it [15]. The goal of this first evaluation was to assess the quality of the initial approach and determine the improvements that were

necessary. The process of test case generation and evaluation of the test cases was performed completely by hand. After this first evaluation, the top priority of the project was defined: we needed a tool to automate the test case generation process. Performing the steps to generate the test cases by hand was a process very susceptible to errors and took a considerable amount of time to accomplish.

After the initial prototype of the tool was implemented, it was evaluated in a second case study [13]. The target now was a model that was little more complex than the model used in the first case study. This second case study revealed some problems in the approach and bugs in the tool. It also helped to identify interesting features that could improve the tool, such as the generation of executable test cases and automatic oracle evaluation for the test cases.

Once the improvements were implemented in the initial version of the tool [14], we took some steps towards more complex models. In the third case study, a model of the Lua[2] programming language API was used to assess the second version of the tool [21]. But the goal now was not only to assess the tool but also to perform some analysis of the quality of the generated test cases. The models of the Lua API were much more complex and challenging for the tool. Once the test cases were generated, they were subjected to a code coverage analysis. In this case study, we also tested new features of the tool such as the test script generator.

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

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

7 Current Status

In this paper, we presented a tool-supported approach that aims to complement the B-Method. This approach tries to solve some of the problems that are inherent to its process, such as the lack of formalization in the code generation step. This Ph.D project is expected to finish by February, 2016.

As mentioned on sections 3 and 4, most of the approach is currently supported and automated by a tool. There are two features that have to be finished yet:

[2] Lua project's website: `http://www.lua.org/`

the preamble calculation and the test data concretization. For the preamble calculation, we already have a prototype implemented, but it still has to be tested through more case studies. The test data concretization strategy is currently just a tested theory. We still have to implement the strategy so it can be automated and integrated into the tool.

Besides implementing these two missing features, we are still planning to perform more case studies, this time focusing on evaluating the scalability of the approach and the tool. Currently, we are also experimenting with mutation testing to evaluate the effectiveness of the generated test cases to find bugs.

The final time line planned for this project is the following:

- *from May/15 to June/15*: we are planning to focus on writing an initial version of the thesis that has to be submitted for a first evaluation. We are also scheduling this time to write papers about the current project achievements;
- *from July/15 to August/15*: we will focus on the implementation of the preamble calculation and test data concretization features;
- *from September/15 to October/15*: we will switch our focus to the case studies for a final evaluation of the tool;
- *from November/15 to January/16*: this time is reserved for final adjustments in the approach and tool and also the writing of the final thesis.

References

1. J. R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.
2. 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.
3. 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.
4. P. Ammann and J. Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, NY, first edition, 2010.
5. S. Burton and H. York. Automated Testing from Z Specifications. Technical report, York, 2000. Report: University of York.
6. 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.
7. A. Gupta and R. Bhatia. Testing functional requirements using B model specifications. *SIGSOFT Softw. Eng. Notes*, 35(2):1–7, 2010.
8. K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. A Practical Tutorial on Modified Condition/Decision Coverage. Technical report, 2001.
9. 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.

10. 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.
11. 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.
12. D. Marinov and S. Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. *International Conference on Automated Software Engineering*, 0:22, 2001.
13. 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.
14. E. C. B. Matos and A. M. Moreira. Beta: a tool for test case generation based on B specifications. *CBSoft Tools*, 2013.
15. 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.
16. G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, New Jersey, third edition, 2011.
17. J. Rushby. Verified software: Theories, tools, experiments. chapter Automated Test Generation and Verified Software, pages 161–172. Springer-Verlag, Berlin, Heidelberg, 2008.
18. 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.
19. 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.
20. F. M. Souza. Geração de casos de teste a partir de especificações b. Master's thesis, Natal, 2009.
21. J. B. Souza Neto and A. M. Moreira. Um estudo sobre geração de testes com BETA: Avaliação e aperfeiçoamento. volume 2, pages 50–55, 2014.
22. 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.