An empirical study of test generation with BETA*

Anamaria M. Moreira[1] **, Ernesto C. B. de Matos**[2] **, João B. Souza Neto**[2]

[1] Departamento de Ciência da Computação – DCC/UFRJ

[2]Departamento de Informática e Matemática Aplicada – DIMAp/UFRN

anamaria@dcc.ufrj.br, {ernestocid,jbsneto}@ppgsc.ufrn.br

Abstract. *Software Testing and Formal Methods are two techniques that focus on software quality and that can complement each other. Many researchers in both fields have tried to combine them in different ways. BETA is one of the efforts in this line of work. It is a tool-supported approach to generate test cases based on B-Method specifications. In this paper, we present an empirical study that was made to evaluate BETA. We performed two case studies with different objectives and that used different techniques to measure, not only the quantity, but the quality of the test cases generated by the approach. With the results of these case studies, we were able to identify some flaws and propose improvements to enhance both the approach and the tool.*

1. Introduction

The industry needs reliable and robust software. This fact increases the demand for methods and techniques that focus on software quality. *Formal Methods* and *Software Testing* are two techniques that have this purpose. Many researchers have tried to combine these two techniques to take advantage of their most interesting aspects. This combination can bring many benefits, such as the reduction of development costs through the application of verification techniques in the initial development phases, when faults are cheaper to be fixed, and the automatic generation of tests from formal specifications [6]. Generating test cases from formal specifications is very convenient. Since these specifications usually state the software requirements in a complete and unambiguous way, they can be a good source to generate test cases. This approach for test case generation can be particularly useful in scenarios where formal methods are not strictly followed, and the code is not formally derived from the model [4]. These tests can help to verify the conformance between the abstract model and the actual implementation.

In [11, 12], the authors presented a tool-supported approach called BETA (*B Based Testing Approach*). BETA generates unit tests from formal specifications written using B-Method [1] notation. Using input space partitioning techniques [2], BETA is capable of generating test cases that verify the conformance between the code implemented and the model that originated it. Initially, BETA was evaluated through two case studies. These case studies showed the feasibility of the approach and tool and were important for an initial evaluation. However, it was not a complete evaluation because the final stages of the testing process – the execution of the tests and the evaluation of its results – were not

*This work is partly supported by CAPES and CNPq grants 2057/14-0 (PDSE), 237049/2013-9, 573964/2008-4, (National Institute of Science and Technology for Software Engineering—INES, www.ines.org.br).

performed, being limited just to the test case design. Because of this, it was necessary to submit BETA to new case studies with the intention of performing a complete evaluation, performing all stages of the testing process and evaluating the quality of the test cases.

Other model-based testing tools share similarities with BETA. [9] presents BZ-TT (*BZ-Testing-Tools*), a tool-supported approach for boundary value test generation from B and Z specifications. BZ-TT was evaluated through comparison with a well-established test set created by specialists. The authors conclude that BZ-TT generated a good set of tests, covering most of the previously defined test set, but they have not provided information on the practical use of the tool. [14] presents ProTest, an automatic test environment for B specifications. It does not present an assessment of ProTest, only discussing differences with respect to other approaches. [15] presents TTF (*Test Template Framework*), which generates unit tests from Z specifications. TTF also uses input space partitioning techniques to generate test cases. TTF was evaluated in several case studies, but they mainly focused on the evaluation of abstract test cases and comparisons with other approaches, not providing information on the quality of the tests generated.

In this context, this work aims to perform an empirical study of BETA to analyze the approach and tool in new situations and in a complete manner, focusing not only on quantitative aspects but also on the quality of the test cases. To do this, BETA was evaluated through two new case studies with different complexity and objectives. For both case studies, the entire testing process was performed, from test case design to the test results evaluation. The results were evaluated quantitatively and qualitatively, using metrics such as statement and branch coverage, and mutation analysis [3].

The remainder of this paper is organized as follows: Section 2 introduces the B-Method and presents the BETA approach and tool; Section 3 presents the methodology of the study; Section 4 presents the two case studies we performed, giving an overview of the process and results for both; Section 5 discusses the final results; ultimately, Section 6 concludes with final discussions and future work.

2. The B-Method and the BETA tool

The B-Method is a formal method that uses concepts of first-order logic, set theory and integer arithmetic to specify abstract state machines that represent a software behavior. The consistency of these specifications can be guaranteed by proving that some automatically generated verification conditions are valid. The method also provides a refinement mechanism in which machines may pass through a series of refinements until they reach an algorithmic implementation, called B0, which can be automatically converted into code. Such refinements are also subject to a posteriori analysis through proofs.

A B machine usually has a set of *variables* that represent the software state and a set of *operations* to modify the state. Restrictions on possible values that variables can assume are formulated in the so-called machine *invariant*. The method also has a *precondition* mechanism for operations of a machine. To ensure that an operation behaves as expected, it is necessary to ensure its precondition is satisfied.

2.1. BETA

BETA is a model-based testing approach to generate unit tests from B-Method abstract state machines. The tool [12] receives an abstract B machine as input and produces test

case specifications. BETA is capable of defining positive and negative test cases for a software implementation. Positive test cases use input data that are valid according to the source specification and negative test cases use input data that are not predicted by the specification. It uses input space partitioning strategies, *Equivalent Classes* (ECS) and *Boundary Value Analysis* (BVS), and combinatorial criteria, *Each Choice* (EC), *Pairwise* (PW), and *All Combinations* (AC), to generate test cases [2]. The main objective of the test cases generated by BETA is to verify the accordance between the code implemented – that could be manually implemented or generated by a code generation tool – and the abstract model that originated it.

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

1. *Identify Input Space:* once the operation to be tested is chosen, the approach identifies the variables that compose the operation's input space and, as such, may influence its behavior. The input space for a B operation is composed of its formal parameters and the state variables that are used by the operation;
2. *Find Operation's Characteristics:* the approach then proceeds exploring the model to find interesting characteristics about the operation. These characteristics are constraints applied to the operation under test. These constraints can be found in invariants, preconditions, and conditional statements, for example;
3. *Partition Input Space and Define Test Case Scenarios:* 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 into test cases. A test case is expressed by a logical formula that describes the test scenario;
4. *Generate Test Case Data:* to obtain input data for these test scenarios a constraint solver is used to solve the logical formulas obtained in the previous step;
5. *Generate Test Case Specifications:* once the test data is obtained, the tool generates test case specifications that guide the developer in the implementation of the concrete test cases;
6. *Obtain Oracle Values:* to obtain oracle data for the test cases the original model has to be animated using the generated test data to verify what is the expected system behaviour for the test case according to the original model. This step is performed manually and can be done using an animation tool for B models;
7. *Concretize Test Data:* there are some situations where the test data generated by the approach does not match the data structures used in the actual implementation of the system. This happens because the test data is generated based on data structures used by the abstract models that use abstract data structures. Because of this, some adaptations in the generated test data might be necessary during the implementation of the concrete test cases.

The BETA tool is free and open-source and can be downloaded on `http://www.beta-tool.info`.

3. Methodology

The work presented in this paper had the objective to perform an empirical study to evaluate BETA, detecting possible limitations and analyzing the quality of the test cases it

generates. To achieve this goal and provide a foundation to improve BETA, we addressed the following questions:

- **Q1** What are the difficulties encountered during the application of the BETA approach in its entirety, and what are the limitations of the BETA tool?
- **Q2** How do the BETA implementation of the partitioning strategies and the combinatorial criteria differ in terms of the amount of generated tests (also taking into consideration feasible and infeasible test case scenarios)?
- **Q3** How do the BETA implementation of the partitioning strategies and the combinatorial criteria differ in terms of system coverage and ability to detect faults?

The first question addresses practical aspects of the use of BETA while the two other questions take into consideration the analysis of the its results. To answer these questions, we submitted BETA to two case studies. In both case studies, the difficulties and limitations found were reported and the results were quantitatively and qualitatively analyzed, using metrics such as statement and branch coverage (baseline coverage criteria), and mutation analysis (often used as reference to evaluate other criteria due to the high quality of its results in spite of its higher costs [2]).

4. Case Studies

4.1. Lua API

In this case study, BETA was used to generate tests for the C API (Application Program Interface) of the Lua programming language [7]. The case study was performed using a partial model of the API specified using the B-Method notation. The model was developed by [13] and was based on the documentation of the API. This model has a total of 23 abstract machines that were divided into three categories: *Lua's types and values definition*, *API's state definition*, and *API's operations definition*. The model has a level of complexity that had not been explored before by BETA, such as compound structures, complex types and values (Lua allows flexible types that do not have a simple representation in B-Method) and a high number of operations (71 API operations).

A. Test Case Generation

To generate the test cases, the machines that defined API operations were submitted to BETA. Initially, each combination of partitioning strategy (ECS and BVS) and combinatorial criteria (EC, PW and AC) was tried, but the tool was not capable of generating test case specifications. After investigation, we found out that the problem was related to the constraint solver used by the tool. The complexity of the Lua API model was just too much for the constraint solver. To solve this problem, we tried to change the constraint solver settings, such as increasing the computation timeout, but it was not enough.

Given this limitation of the constraint solver used by BETA, we decided to use a simplified version of the Lua API model to continue the case study. The simplified version is a subset of the original model, where only a few of the Lua types and their related state and operations were considered. This version has a total of 11 abstract machines, instead of 23 from the original model and specifies 25 API operations, instead of 71 from the original model. For this version, BETA was capable to generate the test cases using all

partitioning strategies and combinatorial criteria. The model for the Lua API does not use numerical ranges in the definitions, so, when this happens, the partitioning strategies ECS and BVS generate the same results. Considering the total of test cases generated with the strategy ECS (or BVS), Figure 1 presents the amount of infeasible test cases, and positive and negative feasible test cases generated by BETA for each combinatorial criteria. The criterion AC generated the largest amount of test cases, followed by PW and EC.

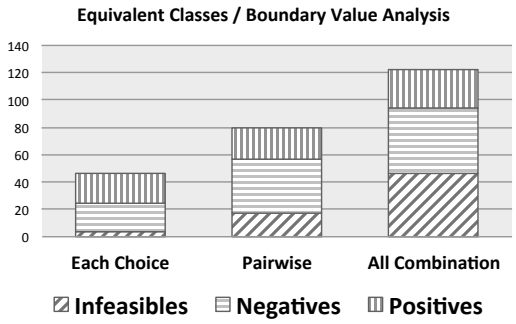

Figure 1. Test cases generated in Lua API case study plot

B. Test Case Implementation

Before implementing the test cases, expected results were obtained using the method proposed by the BETA approach, to be used as test oracles. After the oracles definition, the tests for each one of the 25 operations were implemented. In this step of the study, the negative tests were not considered because it was not our goal to evaluate robustness of the Lua API, besides that, more information is necessary to perform this type of test, such as what is the expected behaviour of the software for a scenario that was not foreseen.

The tests implementation had challenges related to the gap between the Lua API B model and the Lua API standard implementation. The Lua API has a complex implementation. To implement the test cases it was necessary to map the variables of the API's B model into variables of the API's standard implementation. Moreover, it was necessary to create a function to adapt the test data generated by BETA into test data for the actual implementation. After this, all positive tests were implemented and executed. The tests results and its analysis are presented in the next section.

C. Results and Analysis

Since only positive tests were implemented and executed, it was expected that all tests passed. However, some tests for three (3) operations of the API failed. These tests failed because the obtained results with the standard implementation were different from the expected results derived from the B model. Thus, the B model of these operations did not match their standard implementation, that was the main reference for the model. Through reverse engineering, the tests generated by BETA found problems in the Lua API B model.

To analyze the tests generated by BETA in this case study, we used statement and branch coverage metrics. Considering only the main file of the API implementation, and

only the operations tested in this study, the coverage results are summarized in Table 1. The coverage results showed that the tests generated with the criterion AC obtained better results, followed by PW and EC. However, differences in the resulting coverage are much less significant than the differences in the underlying quantity of tests.

	Equivalent Classes / Boundary Value Analysis		
Coverage	**EC**	**PW**	**AC**
Statement	70.6%	76.5%	85.9%
Branch	38.2%	44.1%	58.8%

Table 1. Lua API Case Study Coverage Results

4.2. b2llvm and C4B

In this case study, BETA was used to contribute with the validation of two code generation tools for the B-Method, b2llvm [5] and C4B[1]. Both code generators receive as input a B implementation, in the B0 notation, and produce code (in LLVM, for b2llvm, and C, for C4B) as output. A set of B modules (B abstract machines and their B implementations) was used with both code generators and BETA. The tests generated by BETA were used as oracles to verify the compliance of the code generated by b2llvm and C4B with the original B abstract machine. This case study was presented in [10], mainly focusing on the validation of b2llvm and C4B.

A. Test Case Generation

A set of 13 B modules (*ATM, Sort, Calculator, Calendar, Counter, Division, Fifo, Prime, Swap, Team, TicTacToe, Timetracer* and *Wd*) was selected to be used in the code generators and BETA. The B modules selected use a reasonable range of structures and resources of the B-Method to exercise b2llvm and C4B. Besides, they present different scenarios to exercise BETA. To generate the test cases, the abstract machines of the 13 modules were submitted to the BETA tool. The tool was capable of generating test cases using all partitioning strategies and all combinatorial criteria. But, for two modules, *Division* and *Prime*, the BETA tool was not able to generate the test cases with BVS strategy. This issue was reported to be fixed in the next release of the tool. As some B modules use numerical ranges in their specifications, the results obtained with ECS and BVS partitioning strategies were different.

Considering the total of test cases generated for all 13 modules, Figure 2 presents a plot graph of the amount of infeasible test cases, and positive and negative feasible test cases generated by BETA using ECS (first plot graph) and BVS (second plot graph) partitioning strategies, for each combinatorial criteria. The plots show that the BVS strategy generates the largest amount of test cases. As seen in the first case study, the combinatorial criterion AC generated the largest amount of test cases, followed by PW and EC.

B. Test Case Implementation

The expected results were obtained, using the method proposed by the BETA approach, to be used as test oracles. As with the first case study, the negative test cases were not

[1]C4B is a C code generator integrated with Atelier B, which is an IDE for the B-Method.

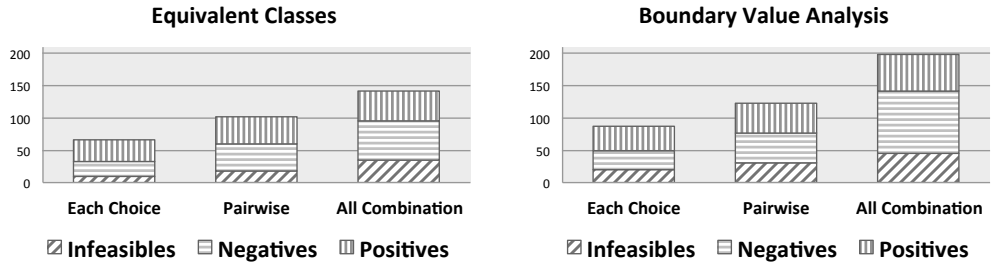

Figure 2. Test cases generated in b2llvm and C4B case study plot

implemented. The reason for this decision is that the code generators (b2llvm and C4B) directly translate the B implementations into executable code, and we can not expect them to behave properly in a situation not foreseen in the specification, and that is exactly what happens in negative tests. The test implementation in this case study did not present the same challenges encountered in the first case study, but still had difficulties related with the differences between the abstract test data, generated by BETA, and the concrete test data. The refinement of the test data (transformation of abstract data to concrete data) was made following the B implementation of each module. With the concrete test data, the test implementation was made without further challenges.

C. Results and Analysis

For each B module, the tests generated by BETA were executed in the LLVM code generated by b2llvm and the C code generated by C4B. The b2llvm tool is still under development, because of that, the LLVM code for some modules could no be generated and the tests were not performed. Nevertheless, the tests generated by BETA can help the b2llvm development team and guide them through the development. The tests for the B modules that b2llvm was capable of generating code did not reveal any problems.

C4B was capable of generating C code for all B modules, because of that, all positive tests generated by BETA were executed. The test results revealed problems in the C code generated by C4B for the *Timetracer* module. This problem was reported to the C4B development team. The tests also found problems related with the refinement process of two modules (*Prime* and *Wd*). This result, that was not related to the code generators, shows that BETA can be used as an alternative to complement the validation in a formal development with the B-Method.

To evaluate the tests generated by BETA, we used statement and branch coverage, and mutation analysis as metrics. In this evaluation, only the test results for the C code generated by C4B were considered (without the *Timetracer* module). The results are summarized in Table 2. The table presents an average of the coverage obtained with ECS and BVS partitioning strategies, and AC, PW and EC combinatorial criteria. To evaluate the capability of the generated test cases to detect faults we performed a mutation analysis, that works with fault simulation by injecting syntactic changes in the program under test [2]. 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. If a mutant can not be distin-

guished from the original, 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 faults simulation, their results provide reliable information about the test effectiveness [3]. The tool *Milu* [8] was used to generate the mutants. The equivalent mutants were detected manually, and the execution and analysis were performed automatically by scripts. Table 3 presents the mutation analysis results. The table shows information about the modules, such as the number of operations and the number of non-equivalent mutants, and the mutation score by the tests generated with each partitioning strategy and each combinatorial criterion. The last row of Table 3 presents an average of the mutation scores. This 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.

	Equivalent Classes			**Boundary Value Analysis**		
Coverage	**EC**	**PW**	**AC**	**EC**	**PW**	**AC**
Statement	72.8%	86.2%	87.6%	78.4%	90.8%	93.3%
Branch	17.7%	32%	45.6%	22.4%	36.7%	50.3%

Table 2. b2llvm and C4B Case Study Coverage Results

Modules			**Percentage of mutants killed - Mutation Score %**					
			Equivalent Classes			**Boundary Value Analysis**		
Name	**Op.**	**Mutants**	**EC**	**PW**	**AC**	**EC**	**PW**	**AC**
ATM	3	11	81.8	81.8	81.8	81.8	81.8	81.8
Sort	1	123	89.4	89.4	89.4	89.4	89.4	89.4
Calculator	6	120	47.5	47.5	47.5	74.2	74.2	74.2
Calendar	1	67	9	19.4	19.4	25.4	35.8	35.8
Counter	4	87	41.4	85.1	85.1	41.4	94.2	94.2
*Division**	1	29	31	31	31	-	-	-
Fifo	2	40	90	90	90	90	90	90
*Prime**	1	66	34.8	34.8	53	-	-	-
Swap	3	8	100	100	100	100	100	100
Team	2	89	68.5	68.5	68.5	68.5	68.5	68.5
TicTacToe	3	764	0	21.9	40.3	0	20.4	40.3
Wd	3	68	91.2	91.2	91.2	91.2	91.2	91.2
Average	**30**	**1,472**	**61.9**	**69.5**	**71.3**	**66.2**	**74.5**	**76.5**

Table 3. Mutation Analysis Results

5. Discussions

In both case studies, the last stages of the BETA approach (test data refinement and test implementation), presented more challenges. The reason for this is that these last steps were not supported by the tool and had to be performed manually. In the other hand, the use of the tool did not present difficulties, requiring the user only to know how to select the parameters (partitioning strategy and the combinatorial criterion) to generate the test cases. Another advantage of the tool when compared to other tools with the same objective is that it requires no adaptations in the model to generate the test cases. It receives the input models as they originally are. The test case specifications are also helpful to guide the implementation of the concrete test cases. About limitations, the first case study revealed a particular limitation related to the constraint solver used by BETA. Because of this, the case study was not executed entirely, a simplified version of the B model had to be used and only a few operations were tested. This result showed that the

constraint solver limitations are propagated to BETA tool and, because of that, the testing process may be affected. These results provide answers to the first question (**Q1**).

The second question (**Q2**) addresses the quantitative aspects of the tests generated by BETA. The results correspond to what should theoretically be expected: BVS generates more tests than ECS when numerical ranges are used in the B module; the combinatorial criterion AC generates substantially more tests than PW, which in turn generates more tests than EC. This pattern was followed by the amount of infeasible tests and feasible positive and negative tests. These results were not different from those obtained in the first case studies performed in [11]. The infeasible test cases are related, generally, to contradictions on the formulas that represent the test cases. Since the criterion AC combines all partitions, it is normal that it generates more infeasible cases.

The third question (**Q3**) addresses the quality of the tests generated by BETA. With the tests generated by BETA we were able to identify errors in both case studies. In the first case study we found errors in a B model through reverse engineering using the tests generated from itself, and in the second case study we found errors in one of the code generators and provided tests to be used in the development of the other. These results showed that BETA is useful to the verification and validation process. To evaluate the tests generated by BETA we used statement and branch coverage, and mutation analysis. The resulting coverage followed the quantity patterns obtained for **Q2**, i.e., more tests lead to greater coverage. However, the results did not show significant variations between the partitioning strategies and the combinatorial criteria as observed in the quantities. These results indicate that, considering only the positive tests, the ECS partitioning strategy obtained almost the same results obtained with BVS. Besides, the results showed that with the combinatorial criterion PW it is possible to obtain very close results from those obtained with AC and with less costs, since it generates less tests. The results obtained for BETA, although derived from a restricted set of models, are consistent with common knowledge that advocates PW combination as providing a good cost/benefit relation [2].

As seen in the coverage analysis, the obtained results reveals that the tests generated by BETA, even in the best cases, could not achieve all statements and decisions of the system under test, which can affect the testing process. The mutation analysis reinforced the coverage results. Even in the best case, the mutation score was not much higher than 75% on average. These results indicate that it is necessary to improve BETA to generate more effective tests, but also provide guidelines on how to do it. For instance, one observation we got from coverage analysis is a small variety of concrete test data provided by the constraint solver which leads to a lower coverage. Requiring the constraint solver to provide more randomly generated data or to provide sets of data for each test case are possible solutions currently in study for integration in BETA.

6. Conclusions and Future Work

This work performed an empirical study to evaluate a tool-supported test case generation approach called BETA. For the evaluation presented in this paper, BETA was submitted to two case studies that presented different scenarios and objectives that were not explored before. As a result, we established parameters on the qualities and limitations of BETA and, because of that, we have a basis to propose and implement improvements in the approach and the tool. Differently from other work in the field ([15, 14, 9]), the evaluation

presented here stands out for doing a complete assessment of the test generation process, from generation of abstract test cases to the implementation of the concrete tests, and also for doing an analysis focusing on the quality of the test cases using mutation testing.

As a consequence of this work, we developed a test script generator to partially automate the last step of the BETA approach, the implementation of concrete tests, and we are adding new testing techniques based on *Logical Coverage* [2] to improve the quality of the test cases. Also, as ongoing and future work we plan to: further explore the limitations of BETA when it comes to complexity of the models, possibly experimenting with different constraint solvers as data generation support; automate the test data refinement; improve the current partitioning and data selection strategies implemented by BETA.

References

[1] J. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge U. Press, 2005.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge U. Press, 2010.

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

[4] D. Carrington and P. Stocks. A tale of two paradigms: Formal methods and software testing. In *Z User Workshop*, Workshops in Computing. Springer, 1994.

[5] D. Déharbe and V. Medeiros Jr. Proposal: Translation of B Implementations to LLVM-IR. *Brazilian Symposium on Formal Methods*, 2013.

[6] R. M. Hierons et al. Using Formal Specifications to Support Testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, 2009.

[7] R. Ierusalimschy. *Programming in Lua*. Lua.Org, 3rd edition, 2013.

[8] 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.

[9] B. Legeard, F. Peureux, and M. Utting. Automated Boundary Testing from Z and B. In *FME 2002:Formal Methods—Getting IT Right*, volume 2391 of *LNCS*. 2002.

[10] E. C. B. Matos, D. Déharbe, A. M. Moreira, C. Hentz, V. Medeiros Jr, and J. B. Souza Neto. Verifying Code Generation Tools for the B-Method Using Tests: a Case Study. In *9th International Conference on Tests & Proofs*, 2015.

[11] E. C. B. Matos and A. M. Moreira. BETA: A B Based Testing Approach. In *Formal Methods: Foundations and Applications*, volume 7498 of *LNCS*. Springer, 2012.

[12] E. C. B. Matos and A. M. Moreira. BETA: a tool for test case generation based on B specifications. In *Proc. of CBSoft Tools*, 2013.

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

[14] M. Satpathy, M. Leuschel, and M. Butler. ProTest: An Automatic Test Environment for B Specifications. *Electronic Notes in Theoretical Computer Science*, 111, 2005.

[15] P. Stocks and D. Carrington. Test template framework: A specification-based testing case study. *SIGSOFT Softw. Eng. Notes*, 18(3):11–18, 1993.