de Matos *et al. Journal of the Brazilian Computer Society* (2016) 22:8
DOI 10.1186/s13173-016-0048-1

Journal of the
Brazilian Computer Society

RESEARCH Open Access

An empirical study of test generation with BETA

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

Abstract

Background: BETA (Bbased testing approach) is a toolsupported approach to generate test cases from Bmethod specifications through the application of input space partitioning and logical coverage criteria. The BETA tool automates the whole process, from the design of abstract test cases to the generation of executable test scripts.

Methods: In this paper, we present an empirical study that was performed to evaluate and contribute to the development of BETA. The study evaluated the applicability of BETA on problems with different characteristics and used techniques of code coverage and mutation analysis to measure the quality of the generated test cases. The study was carried out in different rounds, and the results of each round were used as a reference for the improvement of the approach and its supporting tool.

Results: These case studies were relevant not only for the evaluation of BETA but also to evaluate how different features affect the usability of the approach and the quality of the test cases and to compare the quality of the test cases generated using different coverage criteria.

Conclusions: The results of this study showed that (1) BETAgenerated test scenarios for the different criteria follow theoretical expectations in terms of criteria subsumption; (2) the BETA implementation of the logical criteria generates more efficient test sets regarding code and mutation coverage than the input space partitioning ones; (3) it is important to go beyond the strict requirements of the criteria by adding some additional variation (randomization) of the input data; and (4) the algorithms designed to combine test requirements into test cases need to deal carefully with infeasible (logically unsatisfiable) combinations.

Keywords: Software testing, Formal methods, B-method, Empirical evaluation

Background

Industry needs reliable and robust software, increasing the demand for methods and techniques that focus on software quality. *Formal methods* and *software testing* are two techniques with this purpose. Many researchers have tried to combine these two techniques to take advantage of their most interesting aspects ([1–16]). This combination can bring benefits such as reduction of development costs through the application of verification techniques in the initial development phases, when faults are cheaper to be fixed, and automatic generation of tests from formal specifications [17]. 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 [18]. These tests can help to verify the conformance of the abstract model and the actual implementation.

In [19, 20], a tool-supported approach called BETA (B-based testing approach) is presented. BETA generates unit tests from formal specifications written in B-method [21] notation. Using input space partitioning and logical coverage techniques [22], BETA generates test cases to verify conformance between the code implemented and the model that originated it.

*Correspondence: ernestocid@ppgsc.ufrn.br
[1]Department of Informatics and Applied Mathematics – DIMAp, Federal University of Rio Grande do Norte, Natal RN, Brazil
Full list of author information is available at the end of the article

Initially, BETA was evaluated through two case studies [19, 23]. These case studies showed the feasibility of the approach and tool and were important for an initial evaluation. However, the final stages of the testing process—execution of the tests and evaluation of results—were not performed, and the evaluation was limited to the test case design. Since there were no concrete test cases to execute, it was not possible to assess the quality of the proposed test cases, evaluating, for instance, code coverage achieved by their execution. Recently, new features and improvements were introduced in the approach, and it became necessary to submit BETA to new case studies.

Contributions

In this context, the work presented here extends [24] and aims to perform an empirical study of BETA, analyzing the whole approach and tool in new situations, focusing not only on quantitative aspects but also on the quality of the test cases. In [24], we evaluated a version of BETA (1.2) which only supported input space partitioning coverage criteria. In this paper, we include an evaluation of BETA 2.0, which contains new features (logical coverage criteria and test data randomization) and improvements in the partitioning strategies. In all, three rounds of experiments were executed, using models with different complexity and objectives. In the three rounds, the entire testing process was performed, from test case design to test result evaluation. Quality of the results was evaluated with statement and branch coverage and mutation analysis [25] metrics. Evaluation of BETA showed that the test scenarios it generates for the different criteria follow theoretical expectations in terms of criteria subsumption[1]. It also indicated that the BETA implementation of logical criteria generates more efficient test sets regarding code and mutation coverage than the input space partitioning ones. The positive evolution of BETA from previous case studies and the different rounds of the current study reinforce the utility of such a cyclic, exploratory, process where each evaluation generates improvement requirements which are evaluated in new experiments, making the whole development of BETA a case for such kind of development process, where requirements evolve over time, a very common situation in research projects. Finally, two important results for criteria-based test case generation projects concern the utility of going beyond the strict requirements of the criteria by adding some additional variation (randomization) of the input data and the significant influence of infeasible (logically unsatisfiable) combinations on coverage results, requiring the algorithms designed to combine test requirements into test cases to deal carefully with logically infeasible combinations.

The remainder of this paper is organized as follows: Section The B-method and the BETA tool introduces the B-method and presents the BETA approach and tool; Section Related work presents an overview of relevant related work; Section Methods presents the goals and methodology of the study; Section Case Studies: presentation and results presents the case studies performed, giving an overview of the process and results; Section Discussions presents an analysis of the different experiments; ultimately, Section Conclusions concludes with final discussions and future work.

The B-method and the BETA tool

The B-method

The B-method is a formal method that uses concepts of first-order logic, set theory, integer arithmetic, and contracts defined by a generalized substitution language (GSL [21]) 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 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.

The B-method has a history of success in different "real-world" projects [26] and is supported by mature tools such as AtelierB[2] and ProB[3]. These tools focus on model specification and verification. In this paper, we evaluate another tool called BETA—that is also a test case generation approach—which focuses on test case generation from B models.

BETA

BETA is a model-based testing approach to generate unit tests from B-method abstract state machines. The tool [20] receives an abstract B machine as input and produces test case specifications and executable test scripts written in Java and C. 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. 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. An overview of BETA's test generation process is presented in Fig. 1. The whole process is automated by a tool that is also called BETA.

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. To generate the test cases, BETA supports coverage criteria based on the ones defined in [22] for two different strategies: *input space partitioning* and *logical coverage.*

For input space partitioning, BETA uses *equivalent classes* (ECS) and *boundary value analysis* (BVS) to create partitions based on *characteristics* (restrictions applied to the inputs of an operation under test) extracted from the model. Table 1 presents some examples of how partitions can be created. Each partition contains a set of *blocks* which provide equivalent data to test a particular scenario.

The approach then uses combination criteria to combine the blocks into test cases. It currently supports three combination criteria:

- Each choice (EC): 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 (PW): 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 [27]
- All Combinations (AC): all combinations of blocks from all characteristics must be tested

For logical coverage, BETA inspects the model searching for predicates and clauses to cover (preconditions, conditional statements, etc.) and then applies one of the following supported coverage criteria:

- 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[4] 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*
- 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*
- Combinatorial clause coverage (CoC): the complete truth table of each predicate p in the set of predicates to cover must be tested.

Based on the requirements imposed by each coverage criterion, BETA produces a set of formulas over free

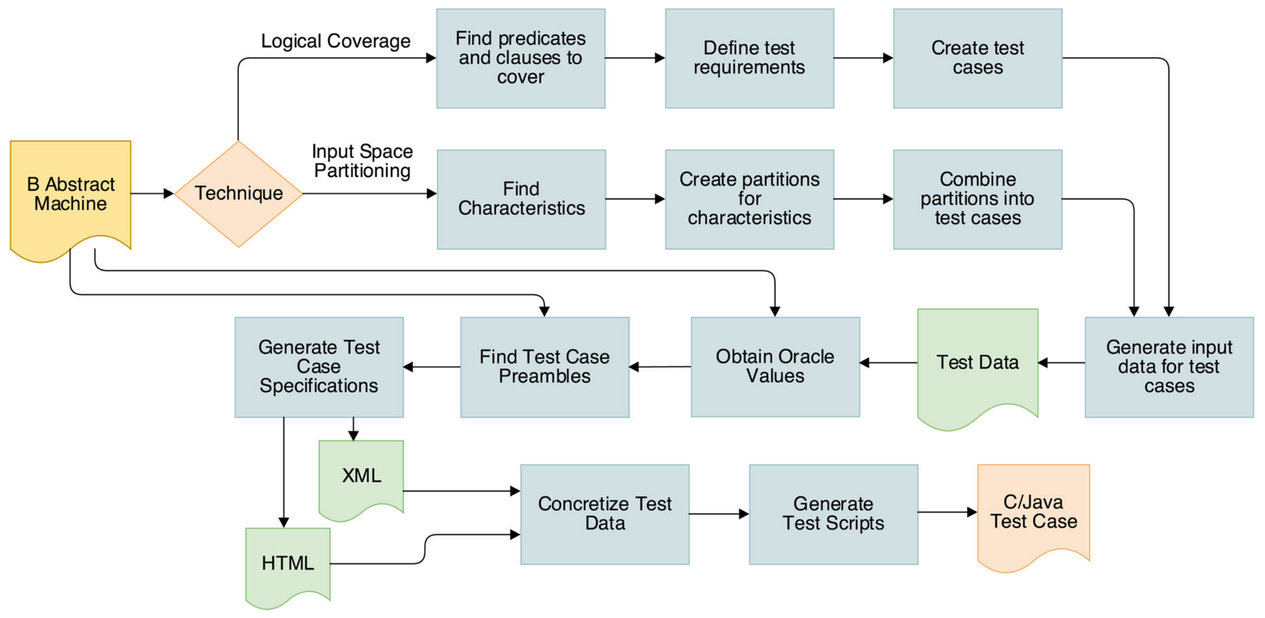

Fig. 1 An overview of the BETA approach. *Rectangular boxes* represent steps of the process, and *sheets* represent input/output artifacts used throughout the process

Table 1 Partitions in BETA

Characteristic	Example	Blocks/values					
Equals to (ECS)	$E_a = E_b$	$E_a = E_b$	$E_a /= E_b$	–	–	–	–
Is boolean (ECS/BVS)	$v \in \text{BOOL}$	$v \in \text{BOOL}$	–	–	–	–	–
Is natural number (BVS)	$v \in NAT$	$v = -1$	$v = 0$	$v = 1$	$v = \text{MAX} - 1$	$v = \text{MAX}$	$v = \text{MAX} + 1$

Examples of how BETA creates partitions using equivalent classes (ECS) and boundary value analysis (BVS) for three types of characteristics. The first example shows the type of blocks that are created using ECS. The third example shows the type of blocks that are created using BVS. Typing characteristics are not partitioned, as seen on the second example. A complete guide explaining how partitions are created for each type of characteristic can be found on BETA's website

variables that represent input data for the operation under test (state variables and operation parameters) that represent test case scenarios. For example, if a machine defines a state variable x to be a natural number from 1 to 5 ($x \in NAT \wedge x \in 1..5$) and the tested operation has a parameter y required to be a natural number greater than x ($y \in NAT \wedge y > x$), a test case scenario may be described as:

$$x \in \text{NAT} \wedge y \in \text{NAT} \wedge x \in 1..5 \wedge y > x$$

BETA then uses a constraint solver to evaluate the test case formulas and obtain test data for them. Currently, BETA relies on ProB [28] as a constraint solver to obtain test data for its test cases. BETA feeds the constraint solver the formulas it created to express the test scenarios and the constraint solver outputs a set of values that comply with the given constraints. For the given example, the constraint solver would provide values such as $x = 1$ and $y = 2$. Additionally, BETA uses a recent functionality provided by ProB to offer a test data randomization feature that can generate less trivial values for the parameters and variables. For instance, the value for a simple integer array with size three can be {3, 70, 25} with randomization on, while it would always be {0, 0, 0} without randomization.

Once test input data is obtained, the original model is animated using these inputs to generate oracle data (expected test case results). The approach currently supports four strategies for oracle verifications: *exception checking* (executes the test and verifies if any exception is raised), *invariant checking* (verifies if the invariant is preserved), *state variables checking* (verifies if the values for the state variables are the ones expected), and *return variables checking* (verifies if the values returned by the operation are the ones expected). These strategies can be combined to make weaker or stronger verifications.

Two recent features of BETA were not exercised via the BETA tool on this study and are described in more detail in [29]: the calculation of a preamble for each one of the test cases and the concretization of the test data. The generated preamble is a sequence of operation calls that takes the system to the intended state to execute the test case. BETA uses information about the state required to run the test case, extracted from the test case formulas, and then uses ProB to find a sequence of operation calls that will put the system in the intended state. Test data concretization can be done manually, using the test engineers own criteria, or automatically, using the concretization feature in the BETA tool. When done automatically, BETA uses the B-method's gluing invariant to find the relationship between abstract variables and concrete variables.

Test inputs, expected results, and preambles are then combined into test case specifications. These specifications can either be in HTML or XML format. The HTML test case specifications are used as a guide that helps the test engineer to code the concrete test cases, while the XML version can be used as input for third-party tools, such as code generators.

Ultimately, the test cases are coded using a programming language and a testing framework of the engineer's choice. The engineer can also use BETA's test script generation feature that uses the XML specifications to generate executable test scripts. BETA's test script generator is capable of generating test scripts written in Java and C, using the JUnit[5] and CuTest[6] framework.

The BETA tool is free and open-source, runs on Windows, Linux, and OS X, and can be downloaded on http://www.beta-tool.info.

Related work

In [30], 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. Their taxonomy classifies the tools in six dimensions: *modeling notations* (*state-based*, *transition-based*, *stochastic*, or *data-flow* notations), *artifacts* (which can model *functional behavior*, *extra-functional behavior*, or *architectural descriptions*), *test selection criteria* (*structural model coverage*, *data coverage*, *fault-based coverage*, *requirements-based coverage*, *ad hoc test case specification*, *random*, or *stochastic criteria*), *Test generation method* (*manual*, *random*, based on *graph search*, *model-checking*, *symbolic execution*, *theorem proving*, or *constraint solving*), *technology* (*online*, if test case generation and execution are done in a single run or *offline*, if test cases are generated and executed separately), and *mapping* (test cases are either *abstract* or *executable*). According to this taxonomy, BETA is classified as a tool that:

- Uses a *state-based or pre/post* modeling notation
- Generates test cases based on artifacts that model *functional behavior*
- Relies on *data coverage* or *structural model coverage* test selection criteria
- Uses *constraint solving* and a set of specific algorithms as test generation method
- Uses *offline* test cases
- Provides *abstract* and *executable* test cases

Other model-based testing tools in the current literature share similarities with BETA. These tools and/or approaches were also evaluated using different techniques. The closest ones are probably BZ-TT/LTG. *BZ-testing-tools* [1] is based on boundary value analysis of B and Z specifications, and *LEIRIOS test generator*[7] [2, 3] is an industrial and commercial tool which derived from BZ-TT, supporting new features such as logical criteria and UML inputs. In BZ-TT, the goal is to test every operation of the system at every reachable boundary state, which is a system state where at least one of the state variables has a maximum or minimum boundary value, while LEIRIOS test generator (LTG) offers more flexible goals. To achieve this goal, BZ-TT/LTG first computes test targets that are the possible behaviors of the operation under test (each conditional branch represents a different behavior). Then, for each test target, BZ-TT/LTG applies boundary value analysis or other criteria thus combining two test selection criteria. They both are based on constraint solving. BETA shares many similarities with BZ-TT/LTG, as they use the same type of model/notation and similar test selection criteria and technology. Originally, in BZ-TT, the application of the criteria was different and more rigid; LTG, however, seems to apply these criteria in a way which is quite similar to BETA's, with the difference of applying them for each behavior, while BETA includes the properties describing each behavior (the conditions of the conditionals) on the material to be used by the different criteria, i.e., these conditions are used, together with preconditions and invariants, to describe data properties in input space partitioning or as predicates in the application of the logical coverage criteria. BZ-TT/LTG was evaluated in academy and industry. In [31], BZ-TT/LTG was applied to an industrial case study in the Smart Card domain and its results were compared with manual tests. The assessment showed that the tests generated by BZ-TT/LTG covered 80 % of the manual tests and saved 30 % of the test design time. Furthermore, BZ-TT/LTG was applied in another industrial case study [4], also in the Smart Card domain, and it was used in academy for teaching purposes [3].

In [5], an automatic test environment for B specifications called *ProTest* is presented. It uses model-checking techniques to find test sequences that satisfy its test generation parameters. ProTest uses offline technology and only generates abstract test cases. It requires specifications to be in a single B machine, and it is up to the user to define the requirements to be satisfied by the test cases. These requirements are made of operations to cover and predicates that must hold true at the end of each test case. In [5], the authors presented a simple case study to evaluate ProTest and discuss theoretical differences concerning other approaches.

TTF (*test template framework* [6]) is an approach that generates unit tests from Z specifications. TTF is automated in the tool *Fastest* [7]. BETA and TTF/Fastest have some similarities; both generate unit tests using input space partitioning techniques and have similar steps in their approach. The input partitioning approach is different however. TTF uses as reference for coverage a unique formula describing the complete set of valid data, generating a set of positive test cases, while BETA defines positive and negative test cases by partitioning and combining specific characteristics for each input variable. TTF/Fastest was evaluated in several case studies, mainly focused on the evaluation of abstract test cases and comparisons with other approaches.

Methods

The work presented in this paper aimed at performing an empirical study to evaluate BETA, detect limitations, and contribute to its evolution. This process was incremental: (1) case studies with different characteristics were used to analyze the general behavior of a version of BETA and the quantity and quality of the test cases it generated and then (2) reused to evaluate the evolution of BETA as new features were added to the approach. For all case studies, the difficulties and limitations found were reported and the results were quantitatively and qualitatively analyzed. Evaluation criteria and metrics were:

General behavior evaluation

For which inputs the tool was not able to generate test cases? Which steps of the approach were not supported by the tool? How much user intervention was needed? How hard was the process? Did the tests uncover defects? Which features were included or improved since the previous versions?

Code coverage

Statement and branch coverage metrics are considered baseline coverage criteria. These metrics were calculated, using *GCOV*[8] and *LCOV*[9]. GCOV is a test coverage analysis software for the *C* language, and LCOV is a graphical front-end for GCOV that generates coverage reports in *HTML*. The version of GCOV used was the one integrated into *Xcode 7.2*[10], an integrated development environment

(IDE) for software development on *OS X*, and the version of LCOV used was 1.2.

Mutation score

Mutation analysis is often used as reference to evaluate other criteria due to the high quality of its results in spite of its higher costs [22, 32]. It was performed to evaluate the capability of the test cases to detect faults. Mutation analysis works with fault simulation by injecting syntactic changes in the program under test [22]. The program with a syntactic change is called *mutant*. Mutants are created from the application of syntactic transformation rules (*mutation operators*) in the code according to the programming language used. A mutant is said to be *killed* when a test can differentiate it from the original. If a mutant cannot be distinguished 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 fault simulation, their results provide reliable information about the test effectiveness [25]. The tool we used to generate mutants in these case studies is called *Milu* [33] (version 3.2[11]). Milu implements the mutation operators presented in [34]. In this work, we applied all mutation operators supported by Milu, that are divided in four categories: statement mutations, operator mutations, variable mutations, and constant mutations, including, for instance, the classical operators of arithmetic and logical operators replacement. The equivalent mutants were identified manually, and the execution and analysis were performed automatically by scripts.

Efficiency of the generated test suites, measured by the relation between their coverage metrics and their size, was also considered in the final analysis, as slightly smaller coverage results are sometimes acceptable if the corresponding cost (number of test cases, in this study) is much smaller.

The following questions were addressed in this work:

- Question 1—General evaluation of BETA: What are the difficulties encountered during the application of the BETA approach in its entirety, and what are the limitations of the BETA tool?
- Question 2—Input space partitioning in BETA: 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?
- Question 3—Logical coverage in BETA: 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?
- Question 4—Comparing logical coverage and input space partitioning in BETA: 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?

The first question addresses practical aspects of the use of BETA. The second and third questions address input space partitioning criteria and logical coverage criteria individually (only making comparisons between criteria from the same family). The last question addresses comparisons between the two families of coverage criteria (input space partitioning and logical coverage).

To answer these questions, BETA was submitted to different rounds of two case studies: testing the C API (application program interface) of the Lua programming language [35] (Section Lua API) and verifying the output of two code generation tools for the B-method (Section b2llvm and C4B). These case studies complemented each other for the intended evaluation. The Lua API is a complex, "real world" problem, while the machines used to test the code generators are simpler but exercise the whole B notation. They presented however a common characteristic: there is no information of any expected behavior of the code when an operation's precondition is not respected. So, although BETA specifies negative test cases as well as positive ones, the complete process, including oracle generation and evaluation and corresponding coverage metrics, was only carried out for the positive ones. Further studies focusing on negative test cases are planned as future work.

The first round of those case studies was presented in [24], from which the current paper is an extension. On this first round that we call *original experiment*, both case studies (Lua API and code generators) were executed for the first time. This experiment was performed with BETA 1.2, which only supported input space partitioning criteria and 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. This new version (BETA 2.0, presented in Section BETA) contains improvements in the strategies used to create partitions for input space partitioning criteria (better partitions for case constructs, additional blocks in the partitions for some characteristics, especially for BVS, and treatment of nested conditions) and also supports logical coverage criteria. BETA 2.0 also supports test data randomization as an alternative when generating test case data. Other features included in this version but which were not targeted in our experiments are automatic oracle generation, preamble calculation, and test data concretization.

BETA 2.0 was then used in two new rounds of execution of the case studies, still without the features for automatic oracle generation, preamble calculation, and test data concretization:

Improved experiment

This second round was performed with the main goals of evaluating the influence of the improvements of BETA partitioning criteria on its results and 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 five times and the average code and mutation coverage obtained by the test cases were used.

These last two experiments only concerned the code generators case study. BETA 2.0 was however used with some operations of the Lua API model to informally corroborate some expected results. Also, some runs of BETA with preamble calculation and test data concretization were successfully executed for some operations of the code generators case study. These individual results cannot be generalized but are indicators of the support provided by BETA to the test engineer.

The results for the main rounds of the case studies are reported in Sections Lua API and b2llvm and C4B and discussed in details in Section Discussions.

Case Studies: presentation and results

Lua API

In this case study, BETA was used to generate tests for the C API of the Lua programming language [35]. The case study was performed using a partial B model [36] of the API, derived from its documentation [37], and has a total of 23 abstract machines. The model has a level of complexity that had not been explored before by BETA, such as compound structures, complex types and values, and a high number of operations (71 API operations).

The most critical aspect of this case study for BETA is that Lua is dynamically typed. This flexibility does not have a simple representation in the B-method. The solution used in the model, similar to the one in the C implementation of the API where a union type represents any Lua value, is the use of the cartesian product of all available types, with a tag indicating the actual data type, as shown in Fig. 2. This solution leads to combinatorial explosion difficulties, as each value of each type is a member of an equivalence class with all combinations of values of the other types. A Lua value of the type *nil*, for example, looks like:

$$(\text{LUA\_TNIL} \mapsto ((((((((\text{nil} \mapsto \text{false}) \mapsto -10) \mapsto \text{LUA\_}\mathit{STRING}1) \mapsto \text{LUA\_FUNCTION1}) \mapsto \text{LUA\_}\mathit{USERDATA}1) \mapsto \text{LUA\_LIGHTUSERDATA1}) \mapsto \text{LUA\_THREAD1}) \mapsto \text{LUA\_TABLE1}))$$

for all possible combinations of values for booleans, integers, LUA_STRING1, LUA_FUNCTION1, etc.

This original version of the model was submitted to BETA, but BETA could not deal with its state space and could not generate any tests: the constraint solver used by the tool is not intended for such use and does not present symbolic abstraction that would be needed to cope with this complexity of data structures. The first result of the Lua case study was then the identification of this limitation of the tool.

A simplified version of the Lua API model was then used 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. The reduction on the number of Lua types led to a significant reduction on the cartesian product representing the state space. Secondary consequences of this reduction were that the number of abstract machines was reduced from 23 to 11 and the number of operations was cut down from 71 to 25 (those quantities were not limitations for the tool, however, since tests are derived for each operation individually). With this reduction, a Lua value of the type *nil*, for example, began to look like:

$$(\text{LUA\_TNIL} \mapsto ((\text{nil} \mapsto \text{false}) \mapsto -10)) \qquad (1)$$

significantly reducing the state explosion. This reduced model still presented some important characteristics for the evaluation of BETA: it partially modeled real and complex code, which was not formally developed, presented a significant abstraction gap, and made use of non trivial compounds of the B notation such as constants, auxiliary functions and definitions (macros).

Test case generation

BETA 1.2 was capable of generating the test cases using all partitioning strategies and combination criteria of the simplified Lua API model.

Because this model does not use numerical ranges in the definitions, the partitioning strategies ECS and BVS generate the same results. Figure 3 presents the total amount of test cases generated by BETA for each combination criteria, classified in infeasible and positive and negative feasible test cases.

```
 1  VALUE_TUPLES = (
 2     LUA_NIL ×
 3     LUA_BOOLEAN ×
 4     LUA_NUMBER ×
 5     LUA_STRING ×
 6     LUA_FUNCTION ×
 7     LUA_USERDATA ×
 8     LUA_LIGHTUSERDATA ×
 9     LUA_THREAD ×
10     LUA_TABLE
11  ) ∧
12  LUA_VALUES = LUA_TYPES × VALUE_TUPLES
```

Fig. 2 Definition of the set of Lua values in the B model of the Lua API. In line 12, the definition of the set of all Lua values, a cartesian product between a set of type tags and the cartesian product of all nine sets that represent specific types of Lua (lines 1 to 11)

Test case implementation

Expected results for each test case were obtained using the method proposed by the BETA approach (animation of the abstract machine), and the positive tests for each one of the 25 operations were implemented.

The implementation of the test cases presented challenges related to the gap between the Lua API B model and the Lua API standard implementation. The Lua API has a complex implementation, and the mapping of the variables of the API's B model into variables of the API's standard implementation was not trivial. Once the concrete data for each test case was defined, all positive tests were implemented and executed.

Test case execution

Some tests for three operations of the API failed (the obtained results with the standard implementation were different from the expected results derived from the B model), meaning that the B model of these operations did not match their standard implementation. The tests generated by BETA were then capable of unmasking defects in the Lua API B model and would have been capable of detecting the corresponding defects on the code in a regular development process, if the B model correctly represents the system's specification.

Statement and branch coverage metrics were used on the code implementing each API operation, disregarding any external functions used by them, which are supposed to have been unit tested separately. Bar charts with the average of the code coverage results are presented in Fig. 4.

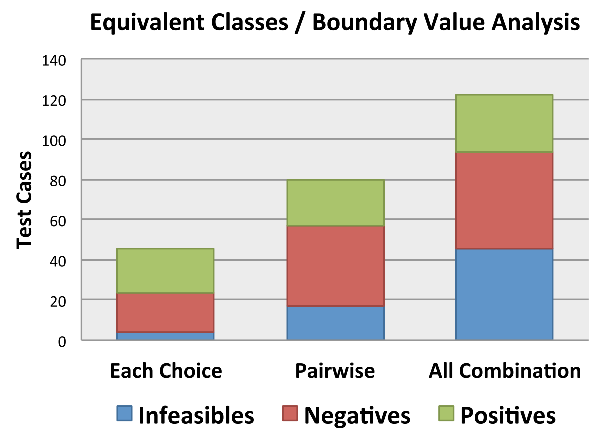

Fig. 3 Amount of the test cases generated by BETA in the Lua API case study. Bar charts with the amount of infeasible test cases and negative and positive feasible test cases generated by BETA in the Lua API case study

b2llvm and C4B

In [38], BETA was used to contribute to the validation of two code generation tools for the B-method, b2llvm [39] and C4B[12]. Both code generators receive as input a B implementation, written in the B0 notation and produce code (LLVM code with a C interface, for b2llvm, and C, for C4B) as output. The tests generated by BETA were used in [38] as oracles to verify the compliance of the code generated by b2llvm and C4B with the original B abstract machine. A set of 13 B modules (*ATM, Sort, Calculator, Calendar, Counter, Division, Fifo, Prime, Swap, Team, TicTacToe, Timetracer* and *Wd*) was used in this evaluation.

The B modules selected use a reasonable range of structures and resources of the B-method to exercise b2llvm and C4B, and, consequently, BETA, focus of the current work. To generate the test cases, the abstract machines of the 13 modules were submitted to the BETA tool and each of the three rounds described in Section Methods (original, improved, and randomization) were fully executed on this case study.

Test case generation

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

Figure 5 presents a bar chart of the total amount of test cases generated by BETA for all 13 modules in the original experiment, and Fig. 6 presents the same information for the improved and randomization experiments, which always lead to the same number of tests since the test data randomization feature does not concern test scenarios generation. In both charts, it is possible to see the

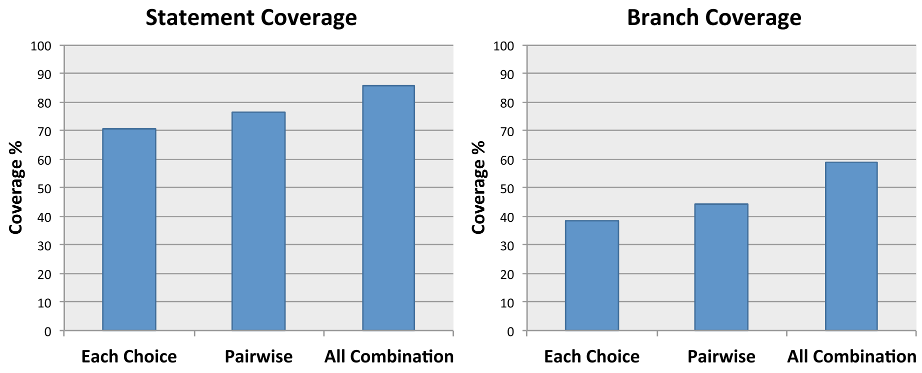

Fig. 4 Statement and branch coverage of the Lua API case study. Bar charts with the average of the statement and branch coverage results obtained with the tests generated by BETA in the Lua API case study

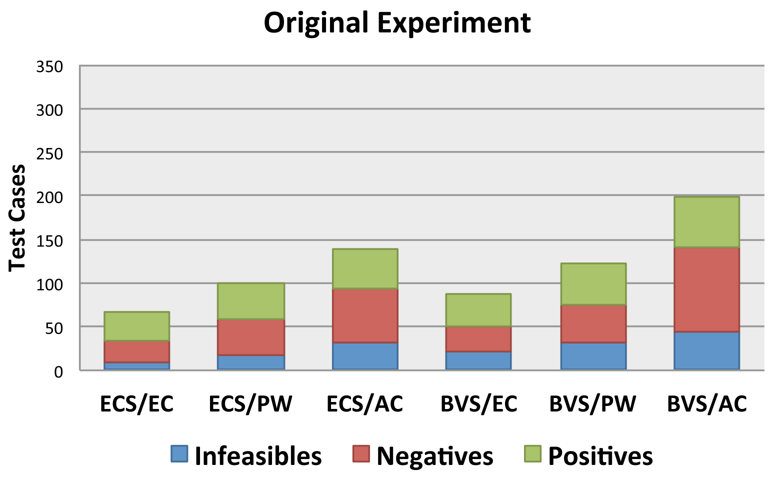

Fig. 5 Amount of the test cases generated by BETA in the original experiment of the b2llvm and C4B case study. Bar charts with the amount of infeasible test cases and negative and positive feasible test cases generated by BETA in the original experiment of the b2llvm and C4B case study

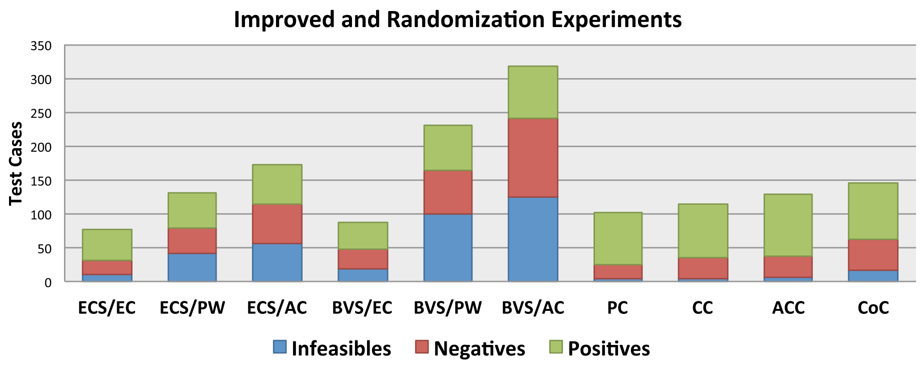

Fig. 6 Amount of the test cases generated by BETA in the improved and randomization experiments of the b2llvm and C4B case study. Bar charts with the amount of infeasible test cases and negative and positive feasible test cases generated by BETA in the improved and randomization experiments of the b2llvm and C4B case study

amount of infeasible test cases and positive and negative feasible test cases generated by BETA for each testing strategy.

Test case implementation

Expected results were obtained for each positive test case of each of the three experiments (original, improved and randomization) using the BETA approach. Implementation of the test cases was easier than in the Lua API case study, although some modules also exercised non-trivial mappings between abstract test data, generated by BETA, and concrete test data. In this case, however, the B implementation of each module was available as reference for the refinement of the test data (transformation of abstract to concrete data).

Test case execution

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. Because b2llvm is still under development, the LLVM code for some modules could not be generated and the tests were not performed. Those tests are being used by the b2llvm development team to guide them through the development. The tests for the B modules for which b2llvm generated code did not reveal any defects.

C4B was capable of generating C code for all B modules, so, all positive tests generated by BETA were executed. The test results revealed problems in the C code generated for the *Timetracer* module. This problem was reported to the C4B development team.

The tests also found problems related to 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 an otherwise not completely validated formal development within the B-method.

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 (without the *Timetracer* module), since b2llvm was under development and could not generate code for all the modules. The results of statement and branch coverage in the three experiments are presented in Figs. 7 and 8.

Mutation analysis was performed to evaluate the capabilities of the test cases to detect faults. Table 2 presents the mutation analysis results in the three experiments. The table shows information on the modules: number of operations, number of non-equivalent mutants, and mutation score by the input space partitioning and logical coverage criteria tests in each experiment. The last row of Table 2 presents an average of the mutation scores in each experiment. 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 them.

Figure 9 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.

Discussions

Question 1: General evaluation of BETA—scope and tool support

The manual application of the BETA approach is a demanding process, as it is expected of any test design approach. The support provided by the BETA tool, however, makes it quite easy to apply. Once a coverage criterion is chosen (from a menu), the process is (semi)

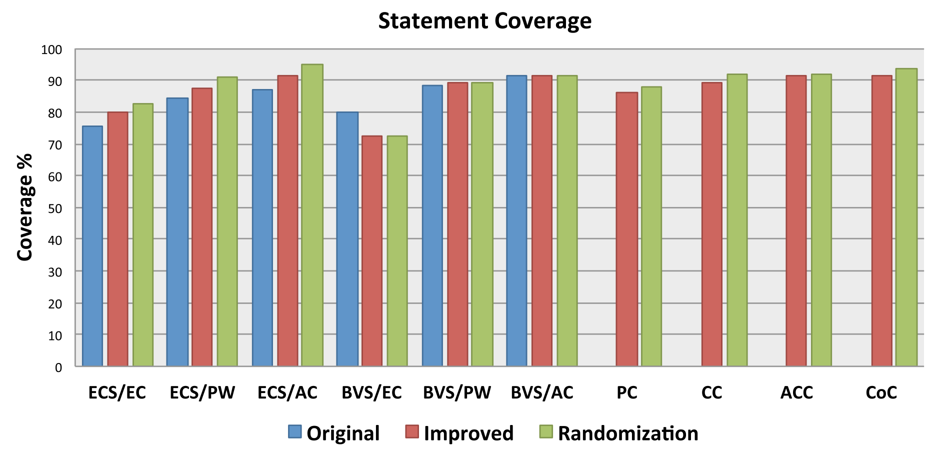

Fig. 7 Statement coverage of the b2llvm and C4B case study. Bar charts with the average of the statement coverage results obtained with the tests generated by BETA in each experiment of the b2llvm and C4B case study (original, improved, and randomization)

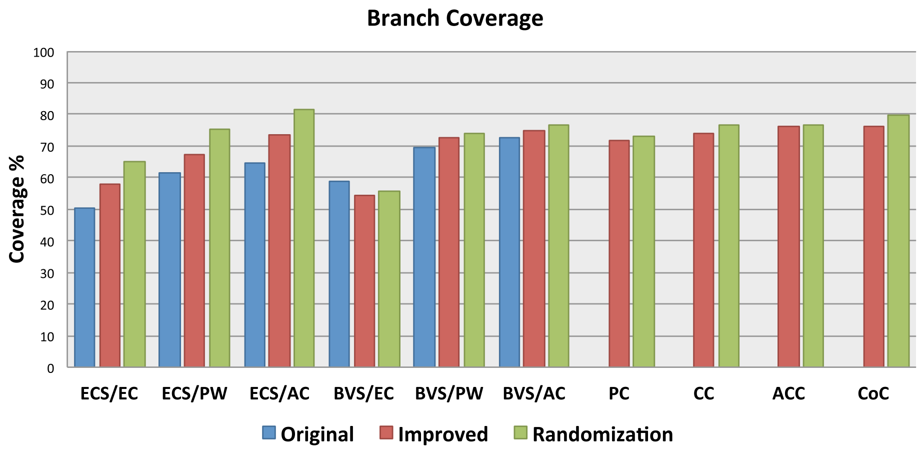

Fig. 8 Branch coverage of the b2llvm and C4B case study. Bar charts with the average of the branch coverage results obtained with the tests generated by BETA in each experiment of the b2llvm and C4B case study (original, improved, and randomization)

automatic up to the generation of concrete test scripts. As it happens with similar tools, it is totally automatic up to the generation of the abstract data. No adaptations or annotations in the model are needed to generate the test cases, making it easier to apply than a user-defined requirement-based coverage tool. Additionally, oracle abstract values are automatically provided by the current version of the BETA tool, so that the user does not need to explicitly run an animator to obtain them.

BETA also supports models structured in several files, while some of the other tools, such as ProTest, require specifications to be contained in a single file. From that point, the user can rely on the provided test case specifications in HTML to guide the implementation of the concrete test cases, but then, there is the need to manually calculate the operation calls that are needed to take the software into the desired input state, convert the abstract data of the specification into the concrete data required by the implementation, and implement the test scripts.

BETA 2.0 also takes care of most of this work automatically. These last features were not assessed during this study, and the difficulties encountered confirmed their importance and the benefits of the automation they can provide.

In both case studies, the last stages of the BETA approach (test data refinement and test implementation) were more challenging, since they were not fully supported by the tool. The automatically generated test scripts were based on the abstract models (abstract B machines). Data concretization can be difficult when there is a significant gap between the abstract model and the implementation, as was the case for the Lua API case study and for some modules of the b2llvm and C4B case study, such as the TicTacToe module. Even though the concretization feature was not targeted in the case studies presented in this paper, initial experiments have shown how this feature can make the implementation of concrete test cases a lot easier.

On the downside, the Lua API case study revealed that the BETA tool suffers from state explosion problems due to the non symbolic constraint solver used. An evaluation of candidate alternatives for this task is mandatory for solving this scalability issue. {*log*} [40], a constraint logic programming language, arises from [41] as a possible alternative, but further studies are needed to see how well it would behave in the presence of a state explosion situation.

Question 2: Input space partitioning in BETA—quantitative and qualitative aspects

BETA input space partitioning tests were capable of identifying faults in both case studies: problems in the B model of the Lua case study; a fault when dealing with structured models in one of the code generators; and refinement mistakes in two B models of the code generators case study. These results showed the effectiveness of BETA in those verification and validation processes.

Regarding the number of test cases generated (Figs. 3, 5, and 6), input space partitioning results of all rounds of the case studies 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 a number of infeasible tests and feasible positive and negative tests.

The corresponding quality metrics (code coverage and mutation analysis), shown in Figs. 4, 7, 8, and 9, followed the quantity patterns. Results also indicate that with PW, it is possible to achieve very close results to those obtained using AC, with smaller costs, since it may generate fewer

Table 2 Mutation analysis results

Modules			Experiment	Percentage of mutants killed—mutation score %									
				Equivalent classes			Boundary value analysis			Logical			
Name	Op.	Mutants		EC	PW	AC	EC	PW	AC	PC	CC	ACC	CoC
ATM	3	11	Original	81.8	81.8	81.8	81.8	81.8	81.8	–	–	–	–
			Improved	81.8	81.8	81.8	81.8	81.8	81.8	81.8	81.8	81.8	81.8
			Randomization	92.7	90.9	87.3	92.7	90.9	87.3	94.5	90.9	96.4	90.9
Sort	1	123	Original	89.4	89.4	89.4	89.4	89.4	89.4	–	–	–	–
			Improved	89.4	89.4	89.4	89.4	89.4	89.4	91.1	91.1	91.1	91.1
			Randomization	90.7	90.7	91.1	90.7	90.7	91.1	91.1	91.1	91.1	91.1
Calculator	6	120	Original	47.5	47.5	47.5	74.2	74.2	74.2	–	–	–	–
			Improved	46.5	46.6	46.6	71.6	71.6	71.6	46.6	46.6	46.6	46.6
			Randomization	58.7	50.8	56.8	78	75	76.2	55.3	59.5	56	51.8
Calendar	1	67	Original	9	19.4	19.4	25.4	35.8	35.8	–	–	–	–
			Improved	80.6	94	94	0	35.8	35.8	94	94	94	94
			Randomization	80.6	94	94	0	35.8	35.8	94	94	94	94
Counter	4	87	Original	41.4	85.1	85.1	41.4	94.2	94.2	–	–	–	–
			Improved	41.4	85.1	85.1	41.4	94.2	94.2	85.1	85.1	85.1	85.1
			Randomization	45.5	86.8	87.1	43.2	95.2	97.5	71.9	80.7	75.4	78.4
Division	1	29	Original	31	31	31	–	–	–	–	–	–	–
			Improved	31	31	31	89.6	96.5	96.5	34.5	34.5	34.5	34.5
			Randomization	59.3	73.8	68.3	89.6	96.5	96.5	53.8	54.5	43.4	74.5
Fifo	2	40	Original	90	90	90	90	90	90	–	–	–	–
			Improved	90	90	90	90	90	90	90	90	90	90
			Randomization	90	90	90	90	90	90	90	90	90	90
Prime	1	66	Original	34.8	34.8	53	–	–	–	–	–	–	–
			Improved	34.8	34.8	53	0	43.9	43.9	62.1	62.1	62.1	62.1
			Randomization	39.1	40.6	75.4	0	43.9	43.9	71.5	71.5	72.4	78.5
Swap	3	8	Original	100	100	100	100	100	100	–	–	–	–
			Improved	100	100	100	100	100	100	37.5	37.5	37.5	37.5
			Randomization	90	85	87.5	90	85	87.5	97.5	80	87.5	95
Team	2	89	Original	68.5	68.5	68.5	68.5	68.5	68.5	–	–	–	–
			Improved	68.5	68.5	68.5	68.5	68.5	68.5	68.5	68.5	68.5	68.5
			Randomization	68.8	69	69	68.8	69	69	67.9	73.3	82.7	80.2
TicTacToe	3	764	Original	0	21.9	40.3	0	20.4	40.3	–	–	–	–
			Improved	0	21.9	40.3	0	20.4	40.3	3.1	23.3	36.8	40
			Randomization	0	20.5	40.5	0	18.1	41.3	3.1	23.4	36.3	40.9
Wd	3	68	Original	91.2	91.2	91.2	91.2	91.2	91.2	–	–	–	–
			Improved	91.2	91.2	91.2	91.2	91.2	91.2	91.2	91.2	91.2	91.2
			Randomization	90	88.8	86.5	90	88.8	86.8	86.5	87.6	88.8	86.5
Average	30	1472	Original	61.9	69.5	71.3	66.2	74.5	76.5	–	–	–	–
			Improved	63	69.5	72.6	60.3	73.6	75.3	65.5	67.1	68.3	68.5
			Randomization	67.1	72.6	77.8	61.1	72.5	75.2	73.1	74.7	76.2	79.1

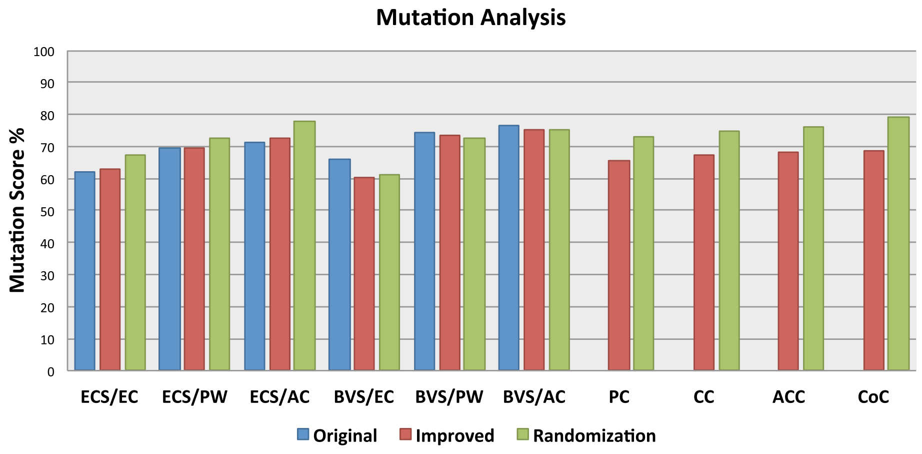

Fig. 9 Mutation analysis results of the b2llvm and C4B case study. Bar charts with the average of the mutation scores obtained with the tests generated by BETA in each experiment of the b2llvm and C4B case study (original, improved, and randomization)

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 cost-benefit ratio [22].

The results also 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 (Fig. 9). 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.

In BETA 2.0, however, the average coverage obtained with the ECS partitioning strategy was better than that obtained with BVS, an unexpected result. This, together with under expected code and mutation coverage, is a consequence of a more general issue of infeasability of test cases in BETA. The combination implementation of BETA when generating test scenarios may lead to infeasible test case 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 logically infeasible, leading to this unexpected behavior. This issue is agravated in BVS 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 pass 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.

Question 3: Logical coverage in BETA—quantitative and qualitative aspects

Considering the number of test cases generated by BETA (Fig. 6), the results for the logical coverage criteria correspond to what should be theoretically expected. The CoC criterion generated 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 appears to have generated slightly more positive tests than CoC. It is a characteristic of the ACC criterion that it sometimes leads to the same test scenario when focusing on different clauses of the operation under test (syntactically different but semantically equivalent formulas). So, some test scenarios are counted twice, accounting for this unexpected behavior.

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. Logical coverage tests were also evaluated using statement and branch coverage analysis and mutation analysis (Figs. 7, 8, and 9). 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 because the models used in the b2llvm and C4B case study 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* [22]. The absence of more complex predicates is a tendency in programming style as observed in [42], but whether this tendency 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.

Question 4: Comparing logical coverage and input space partitioning in BETA

BETA may generate a considerably larger number of test cases using input space partitioning than with logical coverage criteria (Fig. 6). The biggest difference is in the number of infeasible test cases 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 partitioning. In the current case studies, as mentioned before, only positive test cases were executed. Consequently, coverage results for the tests generated using logical coverage criteria were close to or slightly better than the results for the ones generated using input space partitioning criteria, as seen in Figs. 7, 8, and 9.

Threats to validity

An internal validity threat for our study was the manual implementation of the test cases in the Lua API case study. The gap between the model and the Lua's API code may have caused some mismatch between abstract test cases as produced by BETA and concrete test cases executed on the API implementation, influencing the obtained results. We expect that the new features of BETA, such as the test data concretization and test script generation, can minimize this threat in future applications.

The techniques we used to measure the results of our experiments were statement and branch coverage and mutation analysis. The risk associated to that choice is that they may not correspond to actual defect detection capabilities of the test sets. However, they are commonly used with this goal by the software testing community and are considered reliable baseline references for the evaluation of test sets. Smaller issues, often considered as construct validity threats, are related to the auxiliary tools and processes used to measure code coverage and mutation scores such as slightly diverging results observed with different versions of the code coverage tool. The conclusions of the study should not be affected however by those small measurement variations. Another construct validity threat is specifically related to the Lua API case study. The reduction of the Lua API model, forced by a state explosion problem in the original model, had as a secondary consequence that the resulting model, although much more complex than previously performed case studies was not as demanding of BETA as first expected. As one of the main objectives of this case study was to evaluate the behavior of BETA with a complete real world specification, this reduction slightly hindered our goal.

The choice of the B modules for the code generators case study, which focused on exercising the whole B notation instead of focusing on the test generation features of BETA, can be considered as a threat to the external validity of this study. Because only a small percentage of the chosen operations presented complex predicates, we cannot conclude that differences of rigor between the implementation of the different criteria always follows theoretical expectations. The example which presented more complex predicates did correspond to these expectations, however, and there is no reason to think that it would be different for other examples.

Conclusions

In this paper, we presented an empirical study to evaluate a tool-supported test case generation approach called BETA. This work continues the experiments presented on [24], performing new experiments and revisiting old ones. In our empirical experiments, we assessed BETA through two case studies. The experiments were divided in three rounds. In all of them, we evaluated the whole test case generation approach, from design and generation of the test cases to implementation and execution on the system under test. In the first round of experiments, the focus was on test cases generated using input space partitioning criteria. In the second round, we assessed the test cases generated using an improved version of the input space partitioning strategies and the new logical coverage criteria. Finally, in the third round, we experimented with the new test data randomization feature. As a result of these experiments, we presented comparisons between the several improvements made on the tool and how they influenced the quality of the test cases. We could also make comparisons between the quality of the test suites generated by different coverage criteria, with some indicators in favor of logical coverage strategies, and observe the positive influence of test data randomization and the negative influence of infeasible combinations on coverage results.

By using BETA, an engineer can complement the B-method formal development process with units tests. The unit tests generated by BETA are particularly useful to check the conformance between the original abstract model and the actual implementation of the system. Some scenarios that could benefit from the use of BETA are as follows: (1) scenarios where the proofs for the model were not completed, (2) scenarios with manually implemented code which may have human-introduced faults, (3) add an extra verification layer for code produced by code generators, as we have shown in the code generators case study, sometimes they might generate code with faults. The automation provided by the BETA tool provides to the engineer executable criteria-based test suites with low cost.

The construction of the BETA approach and tool is a nice result of an exploratory development process. On [23], the first steps (abstract test case generation) of a preliminary version the approach was applied manually by someone not yet familiar with formal methods or the B method, showing that it is not necessary to be an expert in formal methods to apply it. It showed however that a supporting tool was an essential requirement for the success of the approach. The second case study, on [19], presented an evaluation of those first steps on a second version of the approach with partial tool support. Some further requirements extracted from that study were the need for further definitions and tool support on the steps concerning oracles, preambles, and concretization. Now, despite the promising results obtained by BETA after the three rounds of experiments, it is still necessary to improve the approach so it can generate more effective test sets. A new set or requirements for the next cycle of BETA's development were then conclusions of the current study:

- Fixing infeasible test cases: the current experiments showed that, during the test case generation process, test cases are lost 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 is possible to turn these infeasible test formulas into feasible ones with smarter combination algorithms. An important point of improvement is then to enhance BETA's combination algorithms to avoid contradictory clauses, resulting in fewer infeasible test cases
- Generate test cases from B implementation modules: since B implementations represent more closely the tested code, the ability to also 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 and should also increase coverage results in cases such as the TicTacToe problem where concretization leads to a more complex control structure
- New improvements to the partition strategies: some improvements in the way partitions are created during the test generation process have already been made. We can still improve the partitioning process using knowledge acquired during the latest experiments and using techniques proposed in related work, such as [43], which enumerates a number of partitioning strategies for different constructs used in formal models. In another line of work, some extra flexibility may be provided (with the cost of greater user interaction), allowing for user-defined partition schemas.

Also, as ongoing and future work, we plan to perform new experiments (1) to further explore the limitations of BETA when it comes to complexity of the models and possibly experiment with different constraint solvers for test data generation support, (2) focusing on recent features that were not assessed in this paper, such as preamble calculation and test data concretization, and (3) regarding the negative test cases generated by the approach.

Endnotes

[1] a coverage criterion C_1 subsumes C_2 if every test set that satisfies criterion C_1 also satisfies C_2.

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

[3] ProB's website: https://www3.hhu.de/stups/prob/index.php/The_ProB_Animator_and_Model_Checker

[4] clause = logical formula with no connectives.

[5] JUnit's website: http://junit.org/

[6] CuTests's website: http://cutest.sourceforge.net/

[7] Not publicly available.

[8] GCOV's website: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[9] LCOV's website: http://ltp.sourceforge.net/coverage/lcov.php

[10] Xcode's website: https://developer.apple.com/xcode/

[11] Milu's website: http://www0.cs.ucl.ac.uk/staff/y.jia/Milu/

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

Acknowledgements
The authors would like to thank the anonymous reviewers for the valuable feedback and Yaissa Siqueira for her contribution in some of the experiments. 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).

Authors' contributions

ECBdeM developed the BETA approach and tool as his PhD thesis. He also partially performed some of the case studies presented in this paper. JBdeSN wrote a master thesis where he performed several case studies to evaluate the BETA approach and tool. He is responsible for most of the last experiments performed using BETA. AMM advised both of the previous authors in their projects, contributing in the development of BETA and its empirical evaluation. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Author details

[1]Department of Informatics and Applied Mathematics – DIMAp, Federal University of Rio Grande do Norte, Natal RN, Brazil. [2]Department of Computer Science – DCC, Federal University of Rio de Janeiro, Rio de Janeiro RJ, Brazil.

Received: 28 January 2016 Accepted: 10 October 2016
Published online: 28 October 2016

References

1. Legeard B, Peureux F, Utting M (2002) Automated Boundary Testing from Z and B. In: LarsHenrik E, Lindsay PA (eds). FME 2002: Formal Methods—Getting IT Right: International Symposium of Formal Methods Europe Copenhagen, Denmark, July 22–24, 2002 Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg. pp 21–40. doi:10.1007/3540456147_2
2. Jaffuel E, Legeard B (2006) LEIRIOS test generator: automated test generation from B models. In: Proceedings of the 7th International Conference of B Users. Springer, Berlin. pp 277–280
3. Dadeau F, Julliand J, Tissot R (2008) Leirios test generator: from research to teaching, through industry. In: Int. Workshop on the B Method: from Research to Teaching. APCB, Nantes. pp 1–16
4. Masson PA, Potet ML, Julliand J, Tissot R, Debois G, Legeard B, Chetali B, Bouquet F, Jaffuel E, Van Aertrick L, Andronick J, Haddad A (2010) An access control model based testing approach for smart card applications: Results of the POSÉ project. J Inf Assur Secur 5(1):335–351
5. Satpathy M, Leuschel M, Butler M (2005) ProTest: An automatic test environment for B Specifications. Electronic Notes Theoretical Comput Sci 111:113–136
6. Stocks P, Carrington D (1993) Test template framework: a specification-based testing case study. SIGSOFT Softw Eng Notes 18(3):11–18
7. Cristiá M, Monetti P (2009) Implementing and applying the Stocks-Carrington framework for model-based testing. In: Formal Methods and Software Engineering. LNCS. Springer, Berlin Vol. 5885. pp 167–185
8. Satpathy M, Butler M, Leuschel M, Ramesh S (2007) Automatic Testing from Formal Specifications. In: Gurevich Y, Meyer B (eds). Tests and Proofs: First International Conference, TAP 2007, Zurich, Switzerland, February 12–13, 2007. Revised Papers. Springer Berlin Heidelberg, Berlin, Heidelberg. pp 95–113. doi:10.1007/9783540737704_6
9. Gupta A, Bhatia R (2010) Testing functional requirements using B model specifications. SIGSOFT Softw Eng Notes 35(2):1–7. doi:10.1145/1734103.1734115
10. Singh H, Conrad M, Sadeghipour S (1997) Test case design based on Z and the classification-tree method. In: Formal Engineering Methods., 1997. Proceedings., First IEEE International Conference on. IEEE Computer Society. pp 81–90. doi:10.1109/ICFEM.1997.630406
11. Mendes E, Silveira DS, Lencastre M (2010) TESTIMONIUM: Um método para geração de casos de teste a partir de regras de negócio expressas em OCL. In: Proceedings of the 4th Brazilian Workshop on Systematic and Automated Software Testing (SAST), Natal
12. Burton S, York H (2000) Automated testing from Z Specifications. Technical report, York. Report: University of York
13. Marinov D, Khurshid S (2001) TestEra: A novel framework for automated testing of Java programs. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering. IEEE Computer Society, Washington. pp 22–31. http://dl.acm.org/citation.cfm?id=872023.872551
14. Cheon Y, Leavens G (2002) A simple and practical approach to unit testing: the JML and JUnit way. In: ECOOP 2002 - Object-Oriented Programming. LNCS. Springer, Berlin Vol. 2374. pp 1789–1901
15. Amla N, Ammann P (1992) 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, Gaithersburg. pp 3–10. doi:10.1109/CMPASS.1992.235766
16. Dick J, Faivre A (1993) Automating the generation and sequencing of test cases from model-based specifications. In: FME '93: Industrial-Strength Formal Methods. LNCS. Springer, Berlin Vol. 670. pp 268–284
17. Hierons RM, et al. (2009) Using formal specifications to support testing. ACM Comput Surv 41(2):1–76
18. Carrington D, Stocks P (1994) A tale of two paradigms: formal methods and software testing. In: Z User Workshop. Workshops in Computing. Springer, Cambridge. pp 51–68
19. Matos ECB, Moreira AM (2012) BETA: A B based testing approach. In: Formal Methods: Foundations and Applications. LNCS. Springer, Natal Vol. 7498
20. Matos ECB, Moreira AM (2013) BETA: a tool for test case generation based on B specifications. In: Proceedings of CBSoft Tools Session 2013, Brasília
21. Abrial JR (2005) The B-book: assigning programs to meanings. Cambridge University Press, New York
22. Ammann P, Offutt J (2008) Introduction to software testing. Cambridge University Press, New York
23. Matos ECB, Moreira AM, Souza F, Coelho RdS (2010) Generating test cases from B specifications: an industrial case study. In: Proceedings of 22nd IFIP International Conference on Testing Software and Systems, Natal
24. Matos ECB, Moreira AM, Souza Neto JB (2015) An empirical study of test generation with BETA. In: Proceedings of the 9th Brazilian Workshop on Systematic and Automated Software Testing (SAST), Belo Horizonte
25. Andrews JH, Briand LC, Labiche Y (2005) Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering. ACM, New York. pp 402–411. doi:10.1145/1062455.1062530
26. Behm P, Benoit P, Faivre A, Meynadier JM (1999) Météor: A successful application of B in a large project. FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems. Springer, Berlin
27. Lei Y, Tai KC (1998) InParameterOrder: a test generation strategy for pairwise testing. In: The 3rd IEEE International Symposium on HighAssurance Systems Engineering. IEEE Computer Society, Washington. pp 254–261. http://dl.acm.org/citation.cfm?id=645432.652389
28. Leuschel M, Butler M (2003) ProB: A model checker for B. In: FME 2003: Formal Methods. LNCS. Springer, Berlin Vol. 2805. pp 855–874
29. Matos ECB (2016) BETA: a B based testing approach. PhD thesis, Federal University of Rio Grande do Norte, Natal
30. Marinescu R, Seceleanu C, Le Guen H, Pettersson P (2015) Chapter Three A Research Overview of ToolSupported Modelbased Testing of Requirementsbased Designs. In: Hurson AR (ed). Advances in Computers. Elsevier, Amsterdam Vol. 98. pp 89–140. doi:10.1016/bs.adcom.2015.03.003. http://www.sciencedirect.com/science/article/pii/S0065245815000297
31. Bernard E, Legeard B, Luck X, Peureux F (2004) Generation of test sequences from formal specifications: GSM 11-11 standard case study. Softw Pract Experience 34(10):915–948. doi:10.1002/spe.597
32. Delamaro ME, Maldonado JC, Jino M (2007) Introdução Ao Teste de Software. Editora Campus – RJ, Rio de Janeiro, Brazil
33. Jia Y, Harman M (2008) 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. IEEE Computer Society, Windsor. pp 94–98. doi:10.1109/TAICPART.2008.18
34. Agrawal H, DeMillo RA, Hathaway B, Hsu W, Hsu W, Krauser EW, Martin RJ, Mathur AP, Spafford E (1989) Design of mutant operators for the C programming language. Report SERC-TR-41-P, Purdue University, West Lafayette, Indiana
35. Ierusalimschy R (2013) Programming in Lua. 3rd ed.. Lua. Org, Rio de Janeiro
36. Moreira AM, Ierusalimschy R (2013) Modeling the Lua API in B. Draft
37. Ierusalimschy R, Figueiredo LH, Celes W (2014) Lua 5.2 reference manual. http://www.lua.org/manual/5.2/. Accessed 28 Jan 2016
38. Moreira AM, Cleverton H, Déharbe D, de Matos ECB, Souza Neto JB, de Medeiros V (2015) Verifying code generation tools for the B-method

using tests: a case study. In: Blanchette JC, Kosmatov N (eds). Tests and Proofs: 9th International Conference, TAP 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 22–24, 2015. Proceedings. Springer International Publishing, Cham. pp 76–91. doi:10.1007/9783319212159_5

39. Déharbe D, Medeiros Jr V (2013) Proposal: translation of B implementations to LLVM-IR. Brazilian Symposium on Formal Methods
40. Dovier A, Omodeo EG, Pontelli E, Rossi G (1996) {log}: A language for programming in logic with finite sets. J Log Program 28(1):1–44
41. Cristiá M, Rossi G, Frydman C (2013) {log} as a test case generator for the test template framework. In: Software Engineering and Formal Methods. LNCS. Springer, Berlin Vol. 8137. pp 229–243
42. Durelli VHS, Offutt J, Li N, Delamaro ME, Guo J, Shi Z, Ai X (2016) What to expect of predicates: An empirical analysis of predicates in real world programs. J Syst Softw 113:324–336
43. Cristiá M, Cuenca J, Frydman C (2014) Coverage criteria for logical specifications. In: Proceedings of the 8th Brazilian Workshop on Systematic and Automated Software Testing (SAST), Maceió. pp 11–20