BETA: a tool for test case generation based on B specifications [*]

Ernesto C. B. de Matos[1], **Anamaria M. Moreira**[1]

[1]Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte (UFRN)
Natal – RN – Brazil

ernestocid@ppgsc.ufrn.br, anamaria@dimap.ufrn.br

Abstract. *This paper presents BETA, a tool to generate test case specifications based on B Method state machines. BETA uses restrictions described on a B machine specification – such as invariants, preconditions and conditional statements – to create unit test specifications for an operation. The tool uses equivalence classes and boundary value analysis techniques to partition the operation input space and relies on combinatorial criteria to select partitions to test.*

1. Introduction

The process of verification and validation of software systems is not a simple one. It is a difficult task to ensure that a system is safe, robust and error-free. With this concern, there are many methods and techniques that try to improve quality assurance in software development. Two of these techiniques are software testing and formal methods. This paper presents a tool that takes advantage of both formal methods and testing. It allows the generation of test case specifications for an operation under test based on B Method [1] state machines.

The remainder of the paper is organized as follows: Section 2 gives a brief introduction on the B Method and the motivation behind our work; Section 3 presents the tool, its architecture and the approach used to generate test specifications; in Section 4 we reason about the results of two case studies; in Section 5 we present some related work and we conclude in Section 6 with discussions and future work.

2. Motivation

The B Method is a formal method that uses concepts of first order logic, set theory and integer arithmetic to specify abstract state machines which model the software behavior. These specifications can be verified using proof obligations which ensure the specification's consistence. The method also provides a refinement mechanism in which machines may pass through a series of refinements until it reaches an algorithmic level that can be automatically converted into source code.

A B machine may have a set of variables that represent the software state and a set of operations that can modify it. Restrictions on possible values that variables can assume are stated by the machine's 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 respected. Figure 1 presents a snippet of B machine (adapted from [11]) modelling the substitutions on a soccer team.

[*]This work is partly supported by CAPES and CNPq grants 306033/2009-7, 560014/2010-4 and 573964/2008-4 (National Institute of Science and Technology for Software Engineering—INES, www.ines.org.br).

```
1 MACHINE Player      7  OPERATIONS
2 SETS PLAYER         8  substitute(pp,rr) =
3 VARIABLES team      9   PRE pp : team &
4 INVARIANT           10      rr : PLAYER &
5  team <: PLAYER &   11      rr /: team
6  card(team) = 11    12  THEN team := (team \/ {rr}) - {pp}
```

Figure 1. B machine specification for Player (simplified syntax)

The team roster is represented by a set *PLAYER* while the main team is stored on the state variable *team*. There are two restrictions on the machine's invariant. The first one requires that the main team must be a subset of the roster (line 5). The second restriction requires that the main team must have exactly eleven players (line 6). The machine has a single operation named *substitute* (lines 8-12) which makes substitutions on the team. It receives as parameters a player *pp* who will be replaced in the team and a player *rr* who will take *pp's* place. As preconditions, the operation requires that *pp* must belong to the main team (line 9), *rr* must belong to the roster (line 10) but should not be currently in the main team (line 11).

As it happens with every formal method, however, only some properties of the software are verified in the B method. In most cases, testing the resulting software is also necessary. For instance, what happens when the preconditions for an operation call are not satisfied? How to test specified properties if the B Method is not applied rigorously and some part of the code is produced manually? What if adaptations are made to the automatically generated and verified code? Since B specifications provide important information about restrictions on input space variables (using invariants, preconditions and conditionals) we can use such specifications as a good source for test case generation. With that in mind, we developed a model-based testing tool to generate test case specifications based on B machines. The tool uses explicit restrictions in the model to create partitions for the input space of the operation and then combines these partitions using combinatorial criteria to select test data.

3. The tool

The tool uses an abstract B machine specification to generate test case specifications for the operation under test. The test generation process begins by searching for variables that compose the operation's input space (operation parameters and state variables). Considering the *Player* machine example, the set of variables which belong to *substitute*'s input space is composed by the variable *team* and the parameters *pp* and *rr*.

After the definition of the input space variables for the operation under test it is necessary to find restrictions on the values they can assume. We consider these restrictions as characteristics of the input data. We can find such characteristics on precondition and invariant clauses, and on conditional statements defining the operation. There are no restrictions on the form of how invariants and preconditions are specified but we strongly recommend the use of conjunctions as they provide richer partitions. For the *substitute* operation we have the following characteristics to test: the main team must be a subset of the roster ($team <: PLAYER$), the main team must have exactly eleven players ($card(team) = 11$), the player *pp* must belong to the main team ($pp : team$), the player *rr* must belong to the team roster ($rr : PLAYER$) but should not belong to the main team ($rr /: team$). The conjunction of these clauses represents the valid input domain for *substitute*:

$$team <: PLAYER \,\&\, card(team) = 11 \,\&\, pp : team \,\&\, rr : PLAYER \,\&\, rr /: team$$

After the definition of the operation's input domain, the tool is capable of creating partitions for input data according to concepts of equivalence classes partitioning and boundary value analysis as defined in [4]. Still considering the *Player* example, if we use equivalence classes to create partitions we would end up with the blocks presented in Table 1. In this case we have for each characteristic a block for positive tests and another for negative tests. When the characteristic is defining the type of a variable we do not generate a negative block for it because it will commonly result in compilation errors when implementing concrete test cases.

Table 1. Blocks of test data for the substitute operation

Characteristic	Block 1 (Positive)	Block 2 (Negative)
$team <: PLAYER$	$team <: PLAYER$	-
$card(team) = 11$	$card(team) = 11$	$not(card(team) = 11)$
$pp : team$	$pp : team$	$not(pp : team)$
$rr : PLAYER$	$rr : PLAYER$	-
$rr /: team$	$rr /: team$	$not(rr /: team)$

These blocks can then be combined using combination criteria for test data such as *Each-choice* (every block must be present in at least one test) and *Pairwise* (every pair of non-contracditory blocks must be present in at least one test) [4]. As a result of the combination we will have a set of formulas. These formulas represent a set of test cases for the operation under test. For the *Each-choice* criteria the following are accepted formula combinations:

1. $card(team) = 11$ & $pp : team$ & $rr /: team$ & $rr : PLAYER$ & $team <: PLAYER$
2. $not(card(team) = 11)$ & $not(pp : team)$ & $not(rr /: team)$ & $rr : PLAYER$ & $team <: PLAYER$

These formulas can then be animated on a constraint solver (see Section 3.1) which can provide test data according to the given restrictions. For more information about the BETA approach see [8].

3.1. Archictecture

Figure 2 presents an overview of the BETA architecture. The tool receives as input an abstract B machine (a) and passes it through a parser (b) which checks if the specification is syntactically correct. If it is in fact correct, the parser outputs the specification's syntactic tree (c). As a parser we used the *BParser* which is part of the *ProB* tool [7]. The generated syntactic tree is then used by the partitioner module (d) to generate blocks (e) of data based on characteristics from the operation's input domain. The generated blocks are then combined by the combinator module (f) using the chosen test input data combination criteria. The resulted combination of blocks (g) is then passed to the machine builder module (h) which creates an auxiliary B machine (i). The tool needs this auxiliary

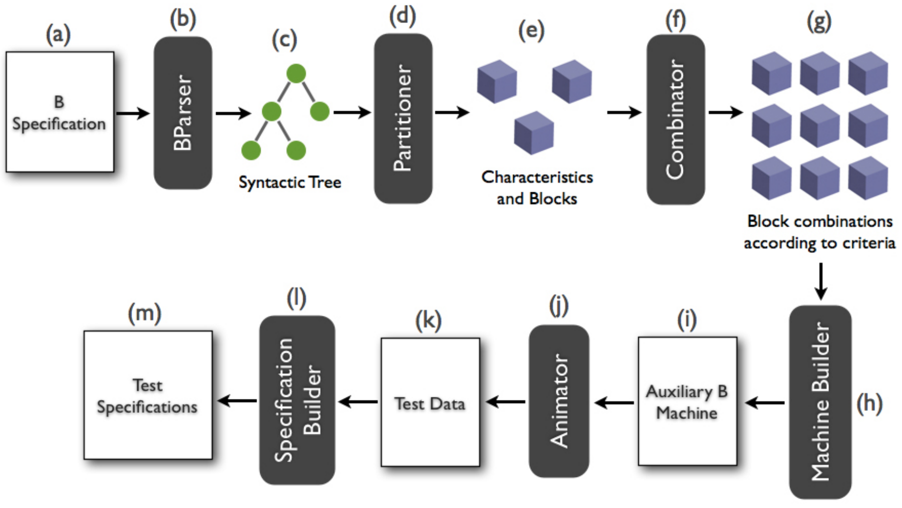


Figure 2. The tool architecture

machine to use a model checker animator (j) to generate test input data for the selected blocks. For this task we use the *ProB* tool command line interface. The data generated by the animator is formatted in a Prolog file which can be read by the specification builder module (l) to generate unit test specifications (m) for the operation under test.

3.2. How it works

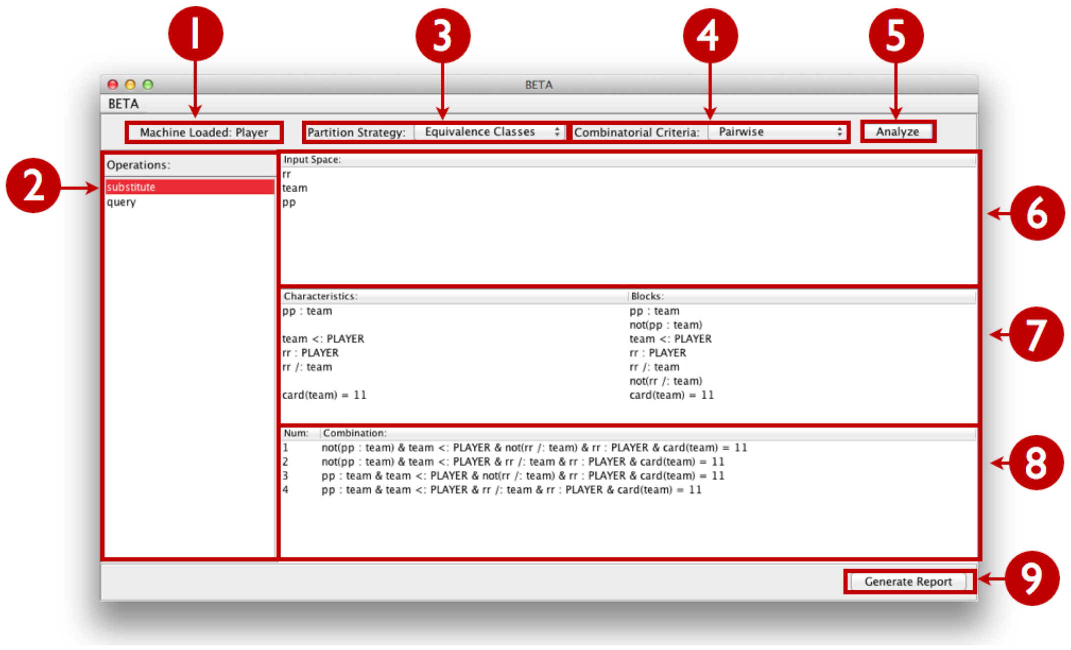
Figure 3 presents the BETA interface and will help us to explain how it works. First we have to load a B machine. Once it is loaded it's name will be shown on (1) and its operations will be listed on the left side navigation bar (2). (3) and (4) show the options for *Partition Strategy* and *Combinatorial Criteria* respectively. On the *Partition Strategy* menu the user can choose between *Equivalence Classes* and *Boundary Analysis* while on the *Combinatorial Criteria* the user can choose between *All-combinations*, *Each-choice* and *Pairwise* criteria. Once the user has selected the operation to test, the strategy and criterion, clicking on *Analyze* (5) will populate the *Input Space* panel (6) with all the variables composing the operation's input space, the *Characteristics* panel (7) with the characteristics and blocks obtained with the chosen partition strategy and the *Combinations* panel (8) with the combinations of the blocks using the chosen combinatorial criterion. At last, clicking on *Generate Report* (9) will ask the user the kind of report he/she wants to generate. The tool can generate an HTML report which is easier to read or an XML that can be translated by other tools into concrete test cases. The tool only takes a few seconds to generate the report.

To obtain oracle values the user has to animate the original machine (we recommend the use of *ProB*) using the input data in the test specification. After this, we have the expected values for the given operation using a particular test input.

4. Tool Evaluation

The approach and the BETA tool itself were evaluated in two case studies. On the first one, we evaluated the approach and the possibility to apply it manually using specfications from a door controlling system developed by the *AES Group*[1]. This system controls many

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



Machine Loaded: Player
Partition Strategy: Equivalence Classes
Combinatorial Criteria: Pairwise

Operations:
substitute
query

```
Input Space:
rr
team
pp
```

Characteristics:	Blocks:
pp : team	pp : team
	not(pp : team)
team <: PLAYER	team <: PLAYER
rr : PLAYER	rr : PLAYER
rr /: team	rr /: team
	not(rr /: team)
card(team) = 11	card(team) = 11

Num:	Combination:
1	not(pp : team) & team <: PLAYER & not(rr /: team) & rr : PLAYER & card(team) = 11
2	not(pp : team) & team <: PLAYER & rr /: team & rr : PLAYER & card(team) = 11
3	pp : team & team <: PLAYER & not(rr /: team) & rr : PLAYER & card(team) = 11
4	pp : team & team <: PLAYER & rr /: team & rr : PLAYER & card(team) = 11

Figure 3. BETA interface

security aspects involving the tasks of opening and closing a subway door. After this case study we concluded that applying the approach manually was impracticable because it would take a long time and its was very error prone [9].

On the second case study we evaluated the BETA tool itself using B specifications for *FreeRTOS*[2] [5], which is a micro kernel for real time systems that provides a layer of abstraction between the application being developed and the hardware, making it easier for applications to access hardware features. We generated test case specifications for its *Queue* module, which is responsible for managing message queues in the system. In some cases, when using *All-Combinations* criterion, due to the complexty of these specifications there were combinatorial explosions. To fix this problem we limited the maximum number of test cases generated. Also, there were some problems with test cases which required infeasible scenarios. This situtation happens when we combine two restrictions in the same formula that are impossible to happen at the same time (e.g. $x > 10$ and $x < 0$). This usually happen when dealing with multiple negative blocks. To reduce the number of infeasible scenarios we generate negative blocks only for charateristics from the precondition and the body of the operation, and do not generate negative blocks for invariant characteristics.

5. Related Work

There is some work in the literature about approaches to generate test cases from B specifications like [3], [10] and [6]. In comparison to these, our work differs in some aspects. First, they focus on the generation of tests cases for the module level of testing. In this level we test a module or a file (e.g. a class) that gathers many units (methods or functions) of the program. In our work we focus on the unit level of testing, generating tests for operations individually. Besides, a problem found in related work – not only for B but for

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

other formal notations – is the lack of tool support which is essential to industry adoption as we concluded after a case study [9]. Another aspect that we tried to improve was the use of classical test concepts and criteria since most of related work use *ad hoc* criteria

6. Discussions and Future Work

In this paper we presented a tool for generation of test case specifications based on B machines. It uses restrictions specified on these machines in the form of invariants, preconditions and conditionals to generate unit test cases for each of its operations. The tool is free and open source. More information and download links can be found on `http://www.beta-tool.info`.

As further work we plan to fix the problems we mentioned on Section 4 and automate the oracle generation process so the whole test generation process can be completely automatic. Besides, we plan to extend the approach to generate test cases for other levels of testing such as system tests using Event-B [2] specifications. Another possibility for improvement of the tool is to integrate it into other tools such as *AtelierB*[3] and *Rodin*[4].

Referências

[1] J. R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1st edition, 1996.

[2] J. R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, Vol. 12, 2010.

[3] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. *FATES (workshop of CONCUR)*, 2002.

[4] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 1st edition, 2008.

[5] S. S. L. Galvão. Especificação do micronúcleo FreeRTOS utilizando Método B. Master's Dissertation, DIMAp/UFRN, 2010.

[6] A. Gupta and R. Bhatia. Testing functional requirements using B model specifications. *SIGSOFT Softw. Eng. Notes*, Vol. 35, 2010.

[7] M. Leuschel and M. Butler. ProB: A model checker for B. In *FME*, volume 2805 of *LNCS*. Springer, 2003.

[8] E. C. B. de Matos and A. M. Moreira. BETA: A B based testing approach. In *Formal Methods: Foundations and Applications*, volume 7498 of *LNCS*. Springer, 2012.

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

[10] M. Satpathy, M. Butler, M. Leuschel, and S. Ramesh. Automatic testing from formal specifications. In *TAP'07: Proceedings of the 1st international conference on tests and proofs*. Springer, 2007.

[11] S. Schneider. *B Method, An Introduction*. Palgrave, 1st edition, 2001.

[3]http://www.atelierb.eu

[4]http://www.event-b.org