Generating Test Cases From B Specifications: An Industrial Case Study

Anamaria Moreira, Ernesto Matos, Fernanda Souza, and Roberta Coelho

Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte
Natal - RN - Brazil
{anamaria,roberta}@dimap.ufrn.br,ernesto@forall.ufrn.br,fernandamsz@yahoo.com.br
http://www.dimap.ufrn.br

Abstract. In this paper we present a case study in the industry about the application of a test case generation method based on B machine specifications. First we describe the step-by-step process of the test case generation method and then we present its use in a particular case study for door controlling module from a subway system. Lastly, we discuss the results obtained after this study and the future work of our research.

Keywords: B Method, Test Case Generation, Case Study

1 Introduction

The use of formal methods is becoming a common standard in the development of critical systems for providing safe and sound specifications, that can be mathematically verified and proved; and generate code after specification refinements. Even though they bring such benefits, they are not enough to ensure that a system is error free. Thus, testing techniques are considered a complementary practice to formal methods and thats becoming common sense [1] that those two practices, when allied, can result in software that is more safe and reliable.

In this paper we apply a method for test cases generation based on B [2] specifications of a real industrial case. The method was previously defined in [3] and we experimented it on specifications of a door controlling system for subways developed by *AeS* (http://www.grupo-aes.com.br). The used specifications can be found in [4]. For more detailed information on this system see [5].

The remainder of the paper is organized as follows: in Section 2 we list and compare related work, in Section 3 we summarize the test case generation algorithm, in Section 4 we present the door controller specification, its particularities for the case study and the results we have obtained, in Section 5 we discuss our conclusions and in Section 6 we look upon the future of our research.

2 Related Work

During our research we have found three papers that are related to ours [6][7][8]. All of them are using state machine - B or Z - representations to generate test cases, but are limited in the coverage criteria provided, by the kind of test inputs generated, by the limitations of the machines they work with and by the tools they used. Only two types of criteria are provided by those works: equivalent classes and/or boundary values, in our work we add few more criteria based on the use of the orthogonal pairs technique. Another aspect that is improved in our work is the ability to also generate test cases for invalid input data. Those works were also limited by the kind of machines they supported, they only worked with machines that had variables classified as integers, sets and/or booleans, in our work we can also deal with interval type variables. Another improvement of our work is the use of more modern tools like the ProB model cheker [9]. The ProB tool allowed us to work with machines located in different files and also detect possible breaking of the invariants, which was an aspect that was lacking in those previous works.

3 The Test Case Generation Method

A B machine specifies, using set theory, integer arithmetic and first order logic, an abstract state machine. The set of states of the machine are defined by its variables and its invariant, which defines the possible values of these variables. A collection of operations are defined through their preconditions and expected behavior. The applied method gathers information on global variables and operation parameters from the operation's precondition and from the machine invariant and defines equivalence classes for them. Abstract test data can then be selected, combined and refined into concrete test cases. The approach for test case generation consists of the steps illustrated in Fig. 1 and detailed below.

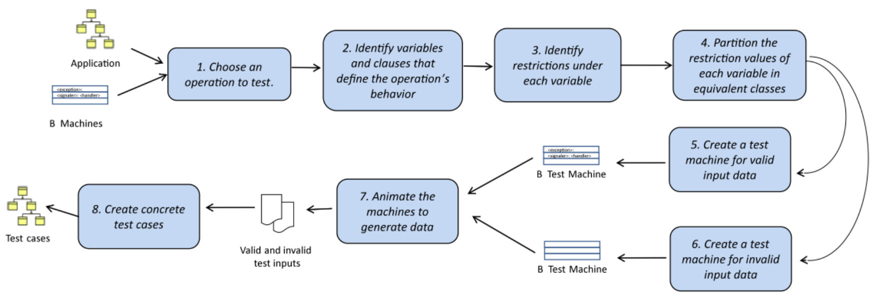
Application
B Machines
1. Choose an operation to test.
2. Identify variables and clauses that define the operation's behavior
3. Identify restrictions under each variable
4. Partition the restriction values of each variable in equivalent classes
5. Create a test machine for valid input data
6. Create a test machine for invalid input data
B Test Machine
B Test Machine
7. Animate the machines to generate data
Valid and invalid test inputs
8. Create concrete test cases
Test cases

Fig. 1. The approach's overview

- ***Step 1: Choose an operation to test.*** The algorithm is applied for each operation we want to test, so the first step is pick up an operation to generate tests for.
- ***Step 2: Identify variables and clauses that define the operations behavior.*** In the second step we have to look at the variables and invariants

of the machine and the parameters and preconditions of the operation we are testing. Two sets are defined: *relevant variables* and *result propositions*. On the *relevant variables* set are placed all the operation's parameters and variables, which occur in the precondition and all the global variables that are directly or indirectly related to them through some restriction on the invariant. The *result propositions* set contains all the operation's precondition clauses and the invariant clauses in which any of the *relevant variables* are restricted.

– ***Step 3: Identify restrictions under each variable.*** In the third step we have to classify all the *relevant variables* according to their type in *intervals*, *belongs to a set*, *sets* or *booleans*. The *classifications* set is created with this information. Besides, the clauses on the *result propositions* set are sort out in two subsets: one containing the *typing clauses* and another containing *non-typing clauses*.
– ***Step 4: Partition the restriction values of each variable in equivalent classes.*** In the forth step we define equivalence classes for the input data of our tests using the equivalence partitioning technique. Based on the *classifications* set we have defined on the previous step we create another set called *formulas* that describe the valid data (valid equivalence classes) for each variable which will be used on our tests. The formulas for each variable are then combined into the *formulas combination* set. Each member of this set is a combination that represents a test case for the operation we are testing.
– ***Step 5: Create a test machine for valid input data.*** To define data that satisfies the restrictions identified for each test case, an auxiliary B machine is created. For each combination of *formulas combination* an operation is inserted in this new machine. These operations have the following structure: each member of *relevant variables* is passed as a parameter of the operation, and the conditions (typing conditions and non-typing conditions) from the combination we are working with are placed as preconditions for the operation.
– ***Step 6: Create a test machine for invalid input data.*** Similarly to the previous step, we define another auxiliary machine to generate data that *do not* satisfy the restrictions on the operation; hence, when we animate it, we will have data for invalid input test cases. We combine the members of *non-typing clauses*, negating one element of each combination. As a result we have the *negative combinations* set and for each of its elements an operation is inserted as we did in step 5.
– ***Step 7: Animate the machines to generate data.*** To generate the data of our test cases we animate the machines created on step 5 and 6 on an animation tool like ProB.
– ***Step 8: Create concrete test cases.*** With the data generated after the machine animations we can implement the concrete tests using a programming language of our choice.

4 Case Study

Our case study uses the B machines specification for a door-controlling module of a real subway system. From the several B machines that specify this module, we have chosen the *General Door Controller* (GDC) machine, which is the main component of the specification.

The GDC machine represents the state of the door-controlling module and defines all the operations and restrictions on these operations. It has operations to open and close the doors on the left and on the right of the subway. Those operations must respect certain preconditions such as the speed of the train at the moment, operation mode of the train, placement of the train in the platform, emergency signals, etc. Bellow we present snippets from the GDC machine:

```
MACHINE GDC
SEES Sets
INCLUDES Opening, Closing, Emergency
ABSTRACT_VARIABLES
    doors_right_open_simultaneously,
    doors_right_close_simultaneously,
    internal_speed_over_6km_h,
    conditions_to_open_satisfied,
    ...
INVARIANT
    doors_right_open_simultaneously : BOOL &
    doors_right_close_simultaneously : BOOL &
    internal_speed_over_6km_h : BOOL &
    conditions_to_open_satisfied : BOOL &

    ((internal_speed_over_6km_h = TRUE) =>
      (doors_right_open_simultaneously = FALSE &
      doors_left_open_simultaneously = FALSE))
      &
    ((doors_right_open_simultaneously = TRUE or
      doors_left_open_simultaneously = TRUE) =>
      (conditions_to_open_satisfied = TRUE))
    ...
INITIALISATION
    ...
OPERATIONS
    simultaneous_opening_all_doors_right =
    PRE internal_speed_over_6km_h = FALSE &
      conditions_to_open_satisfied = TRUE
    THEN ...
    END;
    ...
END
```

The GDC B specification has nineteen operations, four of them define real behavior of the module like opening and closing doors from left and right, and fifteen of them are only "setters" for the machine state. It has twenty-nine abstract variables, forty-six invariant clauses and each operation has at least one precondition clause.

We have chosen the *simultaneous_opening_all_doors_right* operation to illustrate the proposed approach . After applying the algorithm to generate test cases for this operation, we ended up with two test machines, one for valid data and another for invalid data. After we animated the machine for valid data, we had one member of the equivalent class defined by the restrictions, If we wanted to, we could configure the ProB tool to generate more members of this class.

For the invalid data test, we could not animate the test operation because it resulted on a operation with inconsistencies in the precondition. In this case, we could discard this test case since it represented a state that could not happen in the real system.

We could not implement and run the concrete tests on the system because it was not implemented yet, but as soon as its development starts, we can implement those test cases. Since its an embedded system, it will be implemented in C and will probably use a test framework like CUnit.

5 Discussions

The case study was extremely helpful for the method validation. The use of equivalent classes derived from the invariants and operation pre-condition clauses reduced the number of test cases from millions to a few in most cases. By using a machine animation tool, we could easily generate input data for operations that used dozens of parameters, which would be difficult to do by hand.

The test cases obtained from the method were considered satisfactory by the *AeS Group* and the method we proposed could improve their current testing scenario, by providing formalized process for test case generation.

Although the process is usable by the company's current employees, it was considered complex to apply and error prone. Doing each step by hand requires too much attention and if any single mistake is committed the whole process is compromised. Thus, the automation of the process was considered key to its adoption.

6 Concluding Remarks

In this work we proposed a method for test case generation based on B machine specifications that provided three levels of test coverage and tests for both valid and invalid data. The next step of our work is the automation of the process, since it is not a trivial method to apply on a real project and is error prone when done by hand.

We will begin with the first steps, regarding specification interpretation until test machines generation (for both valid and invalid data). We will develop a tool that receives as input the B machine specification we intend to generate tests for and passes it through an interpreter, which will extract the data we need to generate the test cases. With the data extracted by the interpreter, a test case generation module will run the algorithm described in section 3 and output test machines for both valid and invalid data tests. From this point further the method will be applied manually. The user will have to animate the test machines on an animation tool, collect the data generated and implement the concrete tests.

Our interpreter will be integrated to the *B-Compiler* toolset, developed by *ClearSy System Engineering* (http://www.clearsy.com), so we can use the technology already developed and widely tested by them in our tool. The adoption of the B-Compiler also provides support to the B grammar that is most used on the market.

References

1. Bowen, J., Bogdanov, K., Clark, J., Harman, M., Hierons R., Krause, P.: FORTEST: Formal Methods and Testing, In Proc of 26th Annual International Computer Software and Applications Conference, pp. 91 (2002)
2. Abrial, J. R.: The B-Book: Assigning programs to meanings. Cambridge University Press, New York (1996)
3. Souza, F. M.: Geração de Casos de Teste a partir de Especificações B. Master Thesis, DIMAp/UFRN (2009)
4. Barbosa, H.: Desenvolvendo um sistema crtico atravs de formalizao de requisitos utilizando o mtodo B. Thesis, DCC/UFRN (2010)
5. Déharbe, D., Moreira, A. M., Silva, P. S. M., Russo, A. G.: Modelling Control Systems in B: an Industrial Case Study. In: Brazilian Symposium on Formal Methods Proceedings, pp. 195–197 (2001)
6. Ambert, F., Bouquet, F., Chemin, F., Guenaud, S., Legeard, B., Peureux, F., et al.: BZ-TT: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming. In: Proceedings of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02) (2002)
7. Legeard, B., Peureux, F., Utting, M.: Automated boundary testing from Z and B. In: Int. Conf. on Formal Methods Europe, FME02, volume 2391 of LNCS, pp. 21-40 (2002)
8. Bouquet, F., Legeard, B., Peureux, F.: CLPS-B - A constraint solver for B. In: Proceedings of the ETAPS'02 International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2002)
9. Leuschel, M., Butler, M.: ProB: A model-Checker for B. In: FM 2003: 12th International FME Symposium (2003)