Using Dynamic Execution Data to Generate Test Cases

Rozita Dara, Shimin Li, Weining Liu, Angi Smith-Ghorbani
Research In Motion (RIM)
Waterloo, ON, Canada
{rdara, shili, wliu, aghorbani}@rim.com

Ladan Tahvildari
University of Waterloo
Waterloo, ON, Canada
ltahvild@uwaterloo.ca

Abstract

The testing activities of the Software Verification and Validation (SV&V) team at Research In Motion (RIM) are requirements-based, which is commonly known as requirements-based testing (RBT). This paper proposes a novel approach to enhance the current RBT process at RIM, by utilizing historical testing data from previous releases, static analysis of the modified source code, and real-time execution data. The main focus is on the test case generation phase and the objective is to increase the effectiveness and efficiency of test cases in such a way that overall testing is improved. The enhanced process not only automatically generates effective test cases but also seeks to achieve high test coverage and low defect escape rate.

1. Introduction

Research In Motion (RIM)[1], a leading designer, manufacturer and marketer of innovative wireless solutions for the worldwide mobile communications market, has recently initiated a collaborative project with University of Waterloo to improve its testing quality and processes. The current SV&V testing activities at RIM are requirements-based testing (RBT). The RBT approach has two advantages. For one, it improves the quality of the requirements by examining their completeness, clarity and testability. The second advantage is that, based on these requirements, RBT generates a set of test cases in such a way that ensures complete requirement coverage through the design and implementation [3, 6].

Generally, testing is the most time-consuming activity during software maintenance [1]. The goal of testing in the maintenance phase is three-fold: verifying the new changes in the software application, ensuring that the newly added specifications do not affect old functionalities, and confirming that the software application meets and advances the business needs. This paper focuses on the first two goals. The proposed approach utilizes *data* to enhance RBT performance of the newly modified software application. This data is extracted from three sources: i) historical test and source code data from previous releases, ii) static analysis of the modified/added source code, and iii) real-time testing execution data of the modified software application. The enhanced RBT processes automatically generate effective test cases to achieve high test coverage and low defect escape rate. It may also result smaller number of test cases to achieve the test goal.

The rest of the paper is organized as follows. Section 2 provides an overview of the current RBT processes used at RIM. Section 3 discusses the challenges we are facing during daily testing. Section 4 covers the enhancements to the current RBT processes. Finally, conclusions and future work are discussed in Section 5.

2. Current RBT Process

SV&V testing aims to provide information about potential defects in the product, show confidence that new requirements are being met, and examine whether the changes to the software have introduced new defects. To achieve the above objectives, we previously proposed and implemented a sequence of processes for RBT activities [4, 5].

Figure 1 depicts the current RBT process flow. The *Requirement Modeling* process helps testers to build a test model, named as Product Test (PT) model, from requirement specifications of the software under test. Two major objectives are achieved in this process: i) validating that the requirements are complete, unambiguous, logically consistent, and testable; and ii) modeling requirement specifications written in natural language to capture information as to *what to test* and *how much to test* for automating test case generation. The PT model conveys these test information to the *Test Case Generation* process. By applying predefined test goals (e.g. test coverage), the test case generation process automatically generates test case suites that provide complete coverage of the system requirements. Once the source code (i.e. build) is ready, the generated test cases are executed and test results (defects, code coverage, etc.) are

[1] http://www.rim.com

978-1-4244-4828-9/09/$25.00 2009 IEEE

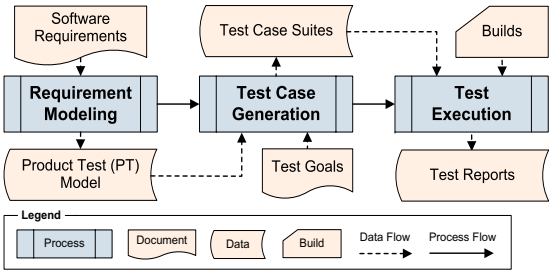

Figure 1. Current RBT Process

reported in the *Test Execution* process.

3. Testing Challenges

Besides requirement coverage, two other test goals have been recently added to the SV&V testing process: achieving higher code coverage and increasing the number of defects discovered. The code coverage data provides an important insight on how effective the test cases are based on what parts of the source code are thoroughly executed and what specific areas are not. To reduce defects that escape to production, we aim to discover more defects during SV&V testing. The number of defects discovered during testing is one of important indicators of testing effectiveness and shows how hard the test team is working to ensure quality goals are being met. Code coverage and defects discovered are measured and reported to management on a timely basis during the course of a maintenance project. These measurements help monitor testing progress, testing efficiency, and facilitate informed decision-making.

Despite all the advantages discussed in Section 2, the current RBT processes do not provide the necessary tools to meet the newly added test goals. Since information about the source code and test execution is not available during test case generation, we came across the following shortcomings:

- Test case redundancy: There may be an extensive redundancy among different test suites. Redundant test cases are costly to execute and do not bring any added value to the testing.
- Low code coverage: Some fragments of the code may be ignored and may not be covered by the test cases. With the current approach, there is no guarantee that the code coverage will be improved.
- Defects escape to production: If the non-executed fragments of the code are the hot-spots, high severity defects may escape to production.
- Time-consuming manual test operations: Choosing the right combination of testing values to achieve higher code coverage and discover more defects is a time consuming process and requires going back and forth from specification to execution and vice versa.

4. Proposed Solution

To help alleviating the aforementioned problems and addressing the challenges we are facing, we propose several enhancements to the current RBT processes: i) utilizing historical testing data and source code information during test case generation phase, ii) generating test cases in an iterative manner to incrementally achieve test goals, and iii) analyzing test execution data (e.g. code coverage and number of defects discovered) of the test cases generated in the pervious iterations and automatically adjusting the algorithm to generate additional test cases in such a way to achieve the test goals faster.

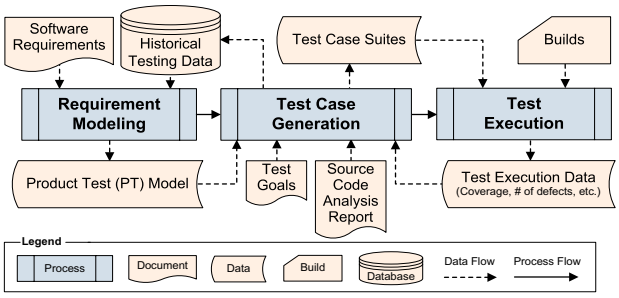

Figure 2. Enhanced RBT Processes.

The enhanced RBT processes are depicted in Figure 2. Compared with Figure 1, three enhancements have been made: i) historical testing data is imported into the requirement modeling process, ii) static analysis for modified part of the code is introduced into the test case generation process, and iii) test execution data of the current release is fed back to test case generation process.

The subsequent two sections elaborate further on the enhancements for requirement modeling and test case generation processes. A simple online hotel reservation system is used to illustrate these two processes. The following is one of requirements of the system:

$R1$ - Users must be able to reserve rooms by specifying location, check-in/check-out date, number of rooms, and number of persons.

4.1. Requirement Modeling

As Figure 3 shows, the improved requirement modeling process consists of the following five sub-processes:

Requirement Review - Reviewing and understanding new requirements from testing point of view, and ensuring that the requirements are complete, unambiguous, logically consistent, and testable.

Feature Identification - Identifying features to be tested (i.e. what to test) from the reviewed requirements. For requirement $R1$ of the online hotel reservation system, two features can be identified:

$F1$ - Users are able to find rooms by specifying location and check-in/check-out dates, using required browsers and operating systems.

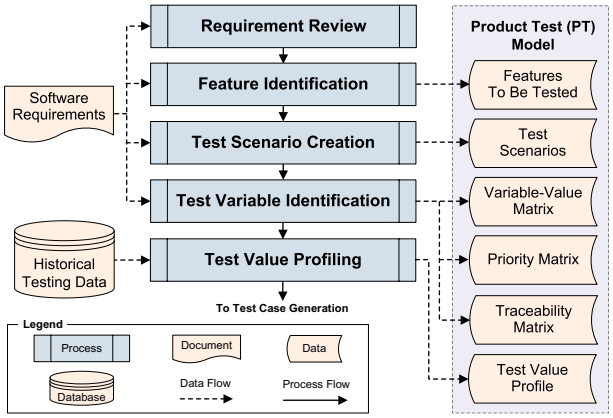

Figure 3. Requirement Modeling Processes.

$F2$ - Users are able to book the rooms found.

Test Scenario Creation - Determining how to test identified features, analyzing use cases for each of the features, and defining (or reusing) one or more test scenarios for each feature. For example, two test scenarios for feature $F1$ and $F2$ can be created as follows:

$TS1$ - Verify that users are able to find rooms by specifying location and check-in/check-out dates.

$TS2$ - Verify that users are able to book found rooms by specifying the number of rooms.

Test Variable Identification - Deriving test variables and their values from test scenarios, and classifying test variables into two categories: functional variables and environment variables. Functional variables represent the functional areas of the system under test. Values of a function variable represent functional units. Environment variables and values represent the environment conditions affecting the execution of a function unit. Table 1 and 2 illustrate partial test variables and values derived from test scenario $TS1$ and $TS2$.

Table 1. Functional Variables and Values

Variable	Values	Environment Variables
Room Reservation	findRoom	location, check-in, check-out, os, browser
	bookRoom	location, check-in, check-out, os, browser

Table 2. Environment Variables and Values

Variable	Values
location	city, airport, attraction, US address
check-in	null, May-10, Jan-1, Dec-31, Feb-29
check-out	null, Mar-20, Jan-12, Dec-31, Feb-29
os	Win2K, WinXP, Vista, Mac
browser	IE6, IE7, Firefox, Chrome

Based on the test variables and values, the Test Case Generation process automatically generates test cases to achieve the test goals (please refer to Section 4.2). The following is an example of the test cases generated:

TC01 - findRoom, location[city], check-in[Feb-29], check-out[Mar-20], os[WinXP], browser[IE7]

In order to automatically link generated test cases to requirements, we need to create the *Traceability Matrix* to capture the relationship between requirements, features, test scenarios, and test variables. Besides, to automatically prioritize generated test cases, we need to assign priority value to each test variable and value in the *Priority Matrix* [5].

Test Value Profiling - Associating test values with code coverage and defect count information by looking up the data from historical testing data repository. Table 3 shows value profile for partial test values of the online hotel reservation system. The numbers in *Code Coverage* column (method m_1, m_2, or m_3) for a value represent the number of test cases that contain that value and cover the method. The number in *Defects* column for a value represents the number of defects found by test cases that contain the value.

Table 3. Test Value Profile

Variable	Values	Code Coverage			Defects
		m_1	m_2	m_3	
location	city	1	1	1	0
check-in	Feb-29	2	1	2	4
check-out	Dec-31	1	2	2	1
os	WinXP	2	1	2	5
browser	IE7	0	1	1	3

4.2. Test Case Generation

Once the PT model has been created and the build is ready for testing, the test case generation process can be iteratively executed to incrementally generate test cases for achieving pre-defined test goals. As Figure 4 shows, the enhanced test case generation process consists of the following four sub-processes:

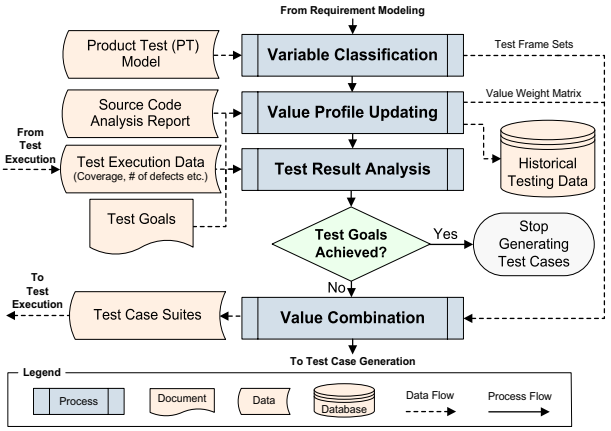

Figure 4. Test Case Generation Processes.

Variable Classification - Categorizing related functional variables into groups. These groups are called *Test Frame Sets*. Each test frame set is passed to the Value Combination sub-process to generate test cases. Test cases generated in each test frame set constitute a test case suite.

Value Profile Updating - Updating the current test value profile with the test execution data from the last iteration, saving the updated profile to test data repository, and computing/updating the weight for each test value. Source code analysis report (e.g. code changes, method dependency matrix, etc.) is used to update code coverage and compute weight for a test value. The weight for a value is determined based on the number of methods (including legacy methods and new methods). Weights may also cover other relevant information (e.g. number of bugs found by a value). The computed/updated weights are then passed to the Value Combination sub-process to generate combinations of different values (i.e. test cases).

Table 4 is an example of test execution data produced by the Test Execution process, and Table 5 contains examples of the computed weights for test values.

Table 4. Test Execution Data

Test Case ID	Code Coverage			Defects
	m_1	m_2	m_3	
TC011	78.3%	45.1%	15.2%	1
TC012	66.1%	48.7%	34.8%	0
TC013	10.0%	36.3%	66.5%	1

Table 5. Value Weight Matrix

Variable	Value	Weight
location	attraction	1
location	US address	1
check-in	Feb-29	0.75
os type	WinXP	0.75
location	city	0.5
location	airport	0.48
check-out	31-Dec	0.54
location	null	0.37
check-out	null	0.37

Test Result Analysis - Analyzing test execution data to determine if the test goals are achieved. If the goals are met, the process stops generating test cases; otherwise, it runs the Value Combination sub-process to generate an additional set of test cases.

Value Combination - Based on the updated value weights from Value Profile Updating, automatically adjusting the pre-defined combination algorithm to generate another nearly-optimized set of test cases to achieve the test goals. The combination algorithm could be any of the linear/multivariate statistical methods, genetic algorithm, and machine learning algorithms [2]. For the example presented here, the greater the weight of the test value, the higher possibility of being selected to create additional test cases.

5. Conclusions and Future Work

This study moves toward a different framework for the test case generation process. We propose the use of data, collected from various sources, to automatically generate efficient test cases. The collected data is processed and utilized in such a way that:

- Allows testers to make informed decision about testing process values and parameters and predict expected testing behavior.
- Allows incremental generation of test cases based on the test goal, which results in smaller number of test cases.
- May result in higher code coverage and large number of bugs found, faster than commonly used processes.
- Offers a self-improving process that memorizes testing behavior in the previous iterations, even previous releases, and adjusts weights accordingly in such a way to achieve higher quality.

It is important to note that the data collected during execution is not limited to code coverage or defects discovered as presented in the example. Data related to code failure is also interesting - to know where and why a failure occurs. Furthermore, depending on the test goal, different algorithms can be used to select and adjust testing values. Machine learning techniques and genetic algorithm are potential candidates for testing value selection and task optimization. The type of algorithm will also affect how weights will be adjusted in each iteration. We plan to apply the enhanced framework to several real testing projects. The above issues will be investigated during the empirical studies.

References

[1] K. H. Bennett and V. T. Rajlich. Software Maintenance and Evolution: a Roadmap. In *Proceedings of the IEEE International Conference on Software Engineering - The Future of Software Engineering*, pages 73–87, 2000.

[2] L. Briand. Novel Applications of Machine Learning in Software Testing. In *Proceedings of the IEEE International Conference on Software Quality*, pages 3–10, 2008.

[3] J. J. Gutiérez, M. J. Escalona, M. Mejías, and J. Torres. Generation of Test Cases from Functional Requirements: a Survey. In *Proceedings of the International Workshop on Software Testing and Validation*, pages 117–126, 2006.

[4] S. Li, L. Tahvildari, W. Liu, M. Morrissey, and G. Cort. Coping With Requirements Changes in Software Verification and Validation. In *Proceedings of the IEEE European Conference on Software Maintenance and Reengineering*, pages 317–318, 2008.

[5] S. Mirarab, A. Ganjali, L. Tahvildari, S. Li, W. Liu, and M. Morrissey. A Requirement-Based Software Testing Framework: an Industrial Practice. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 452–455, 2008.

[6] C. Nebut, F. Fleurey, Y. L. Traon, and J. M. Jézéquel. Automatic Test Generation: a Use Case Driven Approach. *IEEE Trans. Software Eng.*, 32(3):140–155, 2006.