Volume: 04 Issue: 04 Pages:1674-1679 (2013) ISSN : 0975-0290

Deriving Differential Unit Test Cases from System Test Cases

Ramesh V
Asst. Prof. Department of Computer Science and Engineering, Kalasalingam Institute of Technology, Tamilnadu
Email: ramesh_8607@yahoo.co.in
Ananthakumar R
Asst. Prof. Department of Computer Science and Engineering, Kalasalingam Institute of Technology, Tamilnadu
Email: r.ananth05@gmail.com
KannuDurai S
Asst. Prof. Department of Computer Science and Engineering, Kalasalingam Institute of Technology, Tamilnadu
Email: kannuduraivlp@gmail.com

ABSTRACT

Differential testing works by creating test suites for both the original system and the modified system and contrasting both versions of the system with these two suites. Differential testing is made possible by recent advances in automated unit test generation. The differential unit testing is one where developers would like to generate tests that exhibit the behavioral differences between the two versions, if any differences exist. Differential unit tests (DUT) are a combination of unit and system tests. DUTs retain some of the advantages of unit tests, can be automatically and inexpensively generated, and have the potential for revealing faults related to intricate system executions. Some examples of differential unit testing include regression testing, N-version testing, and mutation testing. Differential testing discovered 21%, 34%, and 21% more behavior changes using regression testing techniques than using regression testing alone.

Keywords - **Differential Unit Testing, System Testing, Test Cases, Unit Testing.**

Date of Submission: January 10, 2013 Date of Acceptance: February 13, 2013

I. INTRODUCTION

We focus on differential unit testing, where differential testing is applied on a program unit. System tests are usually developed based on documents that describe the system's functionality from the user's perspective, for example, requirement documents and user's manuals. This makes system tests appropriate for determining the readiness of a system for release, or to grant or refuse acceptance by customers. System tests can be developed without an intimate knowledge of the system internals, which reduces the level of expertise required by test developers and which makes tests less-sensitive to implementation level changes that are behavior preserving. System tests may expose faults that unit tests do not, for example, those that span multiple units or those involve very complex usage of units. Finally, since they involve executing the entire system no test harnesses need be constructed. Behavior of an invocation depends on the method's arguments and the state of the receiver at the beginning of the invocation. Behavior of an invocation can often be observed through the method's return and the state of the receiver at the end of the invocation.

The preceding characterization of unit and system tests, although not comprehensive, illustrates that system and unit tests have complementary strengths and that they offer a rich set of tradeoffs. In this paper, we present a general framework for deriving of what we call differential unit tests (DUT) which aim at exploiting those tradeoffs. We termed them differential because their primary function is detecting differences between multiple versions of a unit's implementation. DUTs are meant to be focused and efficient, like traditional unit tests, yet they are automatically generated along with a custom test-harness, making them inexpensive to develop and easy to evolve. In addition, since they indirectly capture the notion of correctness encoded in the system tests from which they are carved, they have the potential for revealing faults related to complex patterns of unit usage.

II LITERATURE SURVEY

A. Unit Testing

This is the most important of all the testing levels. This is the first and the most important level of testing.

a) Unit Testing Tasks and Steps:

Step 1: Create a Test Plan

Step 2: Create Test Cases and Test Data

Step 3: If applicable create scripts to run test cases

Step 4: Once the code is ready execute the test cases

Step 5: Fix the bugs if any and re test the code

Step 6: Repeat the test cycle until the “unit” is free of all bugs

TABLE 1
Unit Testing Test Case Sample

Test Case ID	Test Case Description	Input Data	Expected Result	Actual Result	Pass /Fail	Remarks

Additionally the following information may also be captured:

a) Unit Name and Version Being tested

b) Tested By

c) Date

d) Test Iteration (One or more iterations of unit testing may be performed)

b) Charles' Six Rules of Unit Testing

1. Write the test first
2. Never write a test that succeeds the first time
3. Start with the null case, or something that doesn't work
4. Don't be afraid of doing something trivial to make the test work
5. Loose coupling and testability go hand in hand
6. Use mock objects

c) Steps to Effective Unit Testing:

1. Documentation: Early on document all the Test Cases needed to test your code. A lot of times this task is not given due importance. Document the Test Cases, actual Results when executing the Test Cases, Response Time of the code for each test case. There are several important advantages if the test cases and the actual execution of test cases are well documented.

a. Documenting Test Cases prevents oversight.

b. Documentation clearly indicates the quality of test cases

c. If the code needs to be retested we can be sure that we did not miss anything

d. It provides a level of transparency of what was really tested during unit testing. This is one of the most important aspects.

e. It helps in knowledge transfer in case of employee attrition

f. Sometimes Unit Test Cases can be used to develop test cases for other levels of testing

2. What should be tested when Unit Testing: A lot depends on the type of program or unit that is being created. It could be a screen or a component or a web service. Broadly the following aspects should be considered:

a. For a UI screen include test cases to verify all the screen elements that need to appear on the screens

b. For a UI screen include Test cases to verify the spelling/font/size of all the “labels” or text that appears on the screen

c. Create Test Cases such that every line of code in the unit is tested at least once in a test cycle

d. Create Test Cases such that every condition in case of “conditional statements” is tested once

e. Create Test Cases to test the minimum/maximum range of data that can be entered. For example what is the maximum “amount” that can be entered or the max length of string that can be entered or passed in as a parameter?

f. Create Test Cases to verify how various errors are handled

g. Create Test Cases to verify if all the validations are being performed

3. Automate where Necessary: Time pressures/Pressure to get the job done may result in developers cutting corners in unit testing. Sometimes it helps to write scripts, which automate a part of unit testing. This may help ensure that the necessary tests were done and may result in saving time required to perform the tests.

TABLE 2

Sample data of the Tested using UNIT TESTING

Test No.	Test ID	Initial State	Test	Expected Result
1	U1-S1-C1	Valid entry in all fields.	Erase the last name and click Add.	Error 101: Last name is a required field.
2	U1-S2-C1	Valid entry in all fields.	Enter a last name with 35 characters and no spaces. Click Add.	Name is accepted, record is added and a clear input screen is displayed.
3	U1-S2-C2	Valid entry in all fields.	Enter a last name of 36 characters and no spaces.	Error 103: Last name may not exceed 35 characters.
4	U1-S3-C1	Valid entry in all fields.	Enter a last name in the form: X'Xxxxx	Name is accepted, record is added and a clear input screen is displayed.
5	U1-S3-C2	Valid entry in all fields.	Enter a last name in the form: 'Xxxxx	Error 107: First character must be a letter.

B. System Testing is a crucial step in Quality Management Process.

1. In the Software Development Life cycle System Testing is the first level where The System is tested as a whole.
2. The System is tested to verify if it meets the functional and technical requirements
3. The application/System is tested in an environment that closely resembles the production environment where the application will be finally deployed
4. The System Testing enables us to test, verify and validate both the Business requirements as well as the Application Architecture

a) Steps to perform System Testing:

Step 1: Create a System Test Plan

Step 2: Create Test Cases

Step 3: Carefully Build Data used as Input for System Testing

Step 3a: If applicable create scripts to

- Build environment and
- to automate Execution of test cases

Step 4: Execute the test cases

Step 5: Fix the bugs if any and re test the code

Step 6: Repeat the test cycle as necessary

b) The format of the System Test Cases

- Test Case ID
- Test Case Description:
  - What to Test?
  - How to Test?
- Input Data
- Expected Result
- Actual Result

TABLE 3

Sample System Testing Test Case Format:

Test Case ID	What To Test?	How to Test?	Input Data	Expected Result	Actual Result	Pass /Fail
.	.	.	.	.	.	.

Additionally the following information may also be captured:

a) Test Suite Name

b) Tested By

c) Date

d) Test Iteration (The Test Cases may be executed one or more times)

c) Various factors that affect success of System Testing:

1. Test Coverage: System Testing will be effective only to the extent of the coverage of Test Cases. What is Test coverage? Adequate Test coverage implies the scenarios covered by the test cases are sufficient. The Test cases should "cover" all scenarios, use cases, Business Requirements, Technical Requirements, and Performance Requirements. The test cases should enable us to verify and validate that the system/application meets the project goals and specifications.

2. Defect Tracking: The defects found during the process of testing should be tracked. Subsequent iterations of test cases verify if the defects have been fixed.

3. Test Execution: The Test cases should be executed in the manner specified. Failure to do so results in improper Test Results.

4. Build Process Automation: A Lot of errors occur due to an improper build. 'Build' is a compilation of the various components that make the application deployed in the appropriate environment. The Test results will not be accurate if the application is not 'built' correctly or if the environment is not set up as specified. Automating this process may help reduce manual errors.

TABLE 4

Test cases have been selected for both valid and invalid inputs.

SEQ NO.	TEST CASE [File]	CONDITION BEING CHECKED	EXPECTED OUTPUT
1	[F1.1]	Incorrect course no. format	Print course no. and error message
2	[F1.7]	More than allowed (30) courses	Error message and skip to lecture times
3	[F1.4]	Course list empty to lecture times	Error message and skip
4	[F1.5]	Spelling of header	Error message and stop
5	[F1.1]	Lecture time format	Print time, error message, and continue
6	[F1.2]	More than allowed no. of lecture times (15)	Error message, discard extra and skip to room no.s

III COMPARING UNIT & SYSTEM TESTING

In all reality we don't take either/or approach to unit and system testing. They do compliment each other nicely - unit tests in the projects give a bit more confidence in the overally system stability - but not to rely on unit testing to ensure stability.

But unit testing should not be looked at as just another form of assurance of the overall system is working as-needed. Unit testing will definitly provide the extra assurance that the system works, but it is really designed to ensure that, should the system need to be changed in the future, it can be changed in predictable ways without causing unexpected side effects. Unit testing is a check against the programmers themselves inadvertently wreaking havoc when trying to modify system behavior.

If a coder writes a function (method) how does s/he know that it works and the work is finished? S/he needs to unit test in isolation from the rest of the system (everything else stubbed out with stubs returning controlled values) so that it performs correctly for the different types of inputs, that is just sound coding practice - define a test, then code till the test works, try to think of more tests and reiterate.

The "system tests" (or functional tests or acceptance tests) are not usually performed by the coder but by the requirements people or their helpers. If these tests are not passed, you haven't fulfilled the contract.

If you have to choose, do just the "system tests", but be prepared for high costs in finding the bugs and also high costs for the system testing as such because you need to make that testing much more detailed. A system test is not going to let you test the behavior of code under insane conditions, or simulate non-deterministic things in a deterministic way.

IV DIFFERENTIAL UNIT TESTING

At any point during the execution of a program the program state, S, can be defined, conceptually, as all of the values in memory. As needed, we will define notation for accessing specific portions of a state, for example, the parameters in the current active frame of the call stack.

A program execution can be formalized either as a sequence of program states or as a sequence of program actions that cause state changes. A sequence of program states is written as $\sigma = s_0 , s_1 , \ldots$ where $s_i \in S$ and s0 is the initial program state. A state s_{i+1} is reached from s_i by executing a single *action*. A sequence of program actions is written as $\bar{\sigma}$. We denote the final state of an action sequence s($\bar{\sigma}$).

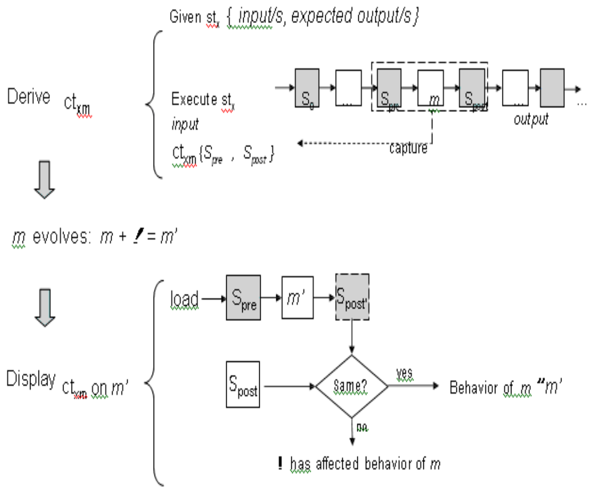

Fig. 1. Deriving process of Testing

Given a system test case st_x , carving a unit test case ct_{x_m} for target unit m during the execution of st_x consists of capturing s_{pre} , the program state immediately before the first instruction of an activation of method m, and s_{post} , the program state immediately after the final instruction of the activation of m has executed. The captured pair of states (s_{pre} , s_{post}), defines a *differential unit test case* for a method, ct_{x_m} . States in this pair can be defined by capturing the appropriate states in σ, or through the cumulative effects of a se- quence of program actions, by capturing $s(\bar{\sigma})$ at the appropriate points in $\bar{\sigma}$. A CR testing approach is said to be *state-based* if it records pairs (s_{pre} , s_{post}) and *action-based* if it records pairs ($\bar{\sigma}_{pre}$, s_{post}) where $s_{pre} = s(\bar{\sigma}_{pre})$.

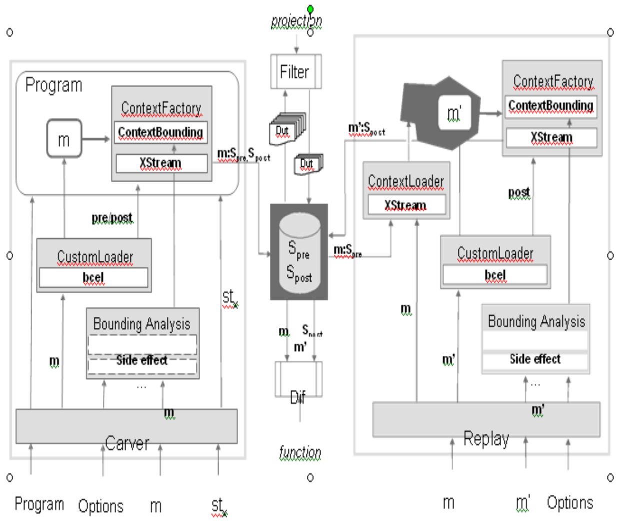

Fig. 2. Architecture of derivative tool

Two set of scripts, represented with double-side rectangles in Figure 4, are utilized to provide the filtering and differencing mechanisms. Once a test suite of DUTs is generated, test case filtering can be performed to remove redundant test cases based on the same set of projections available through Bounding Analysis. Dif scripts compare two s_{post} according to a specified differencing function to determine whether the changes from m to m0 generate a behavioral difference. Currently, differencing functions on return values, on instance fields, on full program state (the default) are fully automated. To facilitate experimentation with different Dif functions our tools currently store the full s_{post} , but we plan to implement options to store only dif(s_{post}) which has the potential to significantly reduce the cost of carving, replay and differencing.

TABLE 3
Sample DUT Test Case Format:

Test Case ID	Test Case Description	What To Test?	How to Test?	Input Data	Expected Result	Actual Result	Pass /Fail
.	-	-	.	.	.	.	.

V .CONCLUSION

The framework incorporates sophisticated projection and differencing strategies that can be instantiated in various ways to accommodate distinct trade-offs. We have implemented a state-based instance of the framework that mitigates testing costs through two types of reachability based projections, and that can adjust the DUTs sensitivity through two differencing functions. Our evaluation of this implementation has revealed that DUTs can be automatically generated from system tests, reduce average test suite execution time to a tenth of our best system selection technique, and still retain most of the fault detection power of system tests.

Differential testing such as regression testing, N-version testing, and mutation testing considers two (or more) versions of the software and seeks test inputs that exhibit behavioral differences between these versions. To reduce the manual effort in checking the outputs between versions and generating inputs that expose behavioral differences, we have proposed the DUT framework for differential unit testing of object-oriented programs. For each public method in the class under test, these annotations invoke the corresponding method in the other version of the class (with the cached method arguments) and compare the return values and receiver-object states of the two corresponding method executions. We can run existing tests on the Java code instrumented by DUT to detect behavioral differences between two versions. Moreover, the Java code instrumented by DUT can be fed to test-generation tools to conduct differential test generation.

REFERENCES

[1]. S. Elbaum, H. N. Chin, M. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proc.14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–264, 2006.

[2]. B. Korel and A. M. Al-Yami. Automated regression test generation. In *Proc. 1998 ACM SIGSOFT International Sym- posium on Software Testing and Analysis*, pages 143–152,1998.

[3]. J. Winstead and D. Evans. Towards differential program analysis. In *Proc. ICSE 2003 Workshop on Dynamic Analy- sis*, pages 37–40, May 2003.

[4]. T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. 20th European Conference on Object-Oriented Programming*, pages 380–403, July 2006.

[5]. D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 76–85, 2004.

[6]. B. Weide. Modular regression testing": Connections to component-based software. In *Workshop onComponent-based Software Engineering*, pages 82–91, May 2001.

[7]. E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 15(4):465–470, 1982.

[8]. T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 2006.

[9]. XStream. Xstream - 1.1.2. http://xstream.codehaus.org, Aug. 2005.

[10]. A. Orso and B. Kennedy. Improving dynamic analysis through partial replay of user's executions. Dagstuhl Seminar: Understanding Program Dynamics, Dec. 2003.

[11]. Orso and B. Kennedy. Selective capture and replay of program executions. In *Workshop on Dynamic Analysis*, May 2005.