Applying Safety Critical Requirement Based Test Practices for Non-Safety Critical Systems

[#1] Venkata TulasiRamu P [*2] Dr.S.V.Naga Srinivasu [#3] V.V.A.S.Lakshmi

[1] *Research Scholar, Acharya Nagarjuna University, Nagarjuna Nagar,Guntur, A.P*
[3] *Principal and professor in CSE, Indira Institute of Technology and Sciences, Markapur, A.P*
[2] *Associate Professor, Narasaraopeta Institute of Technology, Narasaraopet, A.P*
[1] sreetulasisree@gmail.com
[3] vvaslakshmi@gmail.com
[2] dr.svnsrinivasu@gmail.com

***Abstract*— Requirement Based Testing (RBT) approach desegregates testing throughout the software development life cycle and focuses on the early fault detection and quality improvement of the requirements specification. Early defect detection which has been proven to be surely less expensive than finding defects in the further stages. In the context of present work, we briefed on the design issues associated with RBT, the best RBT practices followed in safety critical systems and how the best practices would benefit the non-safety critical systems.**

***Index Terms*—Requirement Based Testing, Software Verification and Validation, Fault detection, Coverage Analysis**

I. INTRODUCTION

The Requirements-Based Testing[1] (RBT) addresses two major issues like validating the requirements which are correct, complete and logically consistent followed by designing a necessary and sufficient set of test cases from those requirements to ensure the design and code comply with requirements. The major issues in test design could be reducing the test cases to a reasonable size and ensuring that each test has a specific objective. This particular process does not assume that software requirements are correct. The RBT process will drive out requirements ambiguity. This process is more useful as it focuses on defect prevention rather than just detection. It has wide applications in critical safety systems in various engineering fields like defense, aerospace, automotive systems and medical systems as well.

A safety critical system is the system whose failure may result in death or serious injury to people or loss/ severe damage to equipment or environmental harm. The best examples of safety critical systems are defense, aerospace and medical systems. A non-safety critical system is the system whose failure may result in reduced efficiency of the system or loss of the asset. The best examples of non-safety critical systems are entertainment systems and any application which would not harm the people and environment. To be the most successful project, quality must be maximized, while minimizing cost and keeping delivery time short. Quality can be measured by customer satisfaction with the resulting system based on the requirements that are incorporated successfully in the system. RBT verifies the software against its requirements, which assures the customer that requirements are incorporated successfully in the software. However RBT is mandated for safety critical systems. A generic RBT process was described by Mogyorodi[5], and BenderRBT[4].

The present work describes the best practices followed in RBT for safety critical software systems and also it describes how the suggested best practices can be implemented in non-safety critical software systems.

II. RBT PROCESS IN SAFETY CRITICAL SYSTEM SOFTWARE

The software verification[3] test cases are to be created based on the software requirements specification. The first step is to develop functional and robustness tests to completely test and verify the implementation of the software requirements. The second step is to measure coverage (functional and structural). The measure of structural coverage will help in providing an indication of the software verification campaign completion status. The tests stop criteria is not limited to a specific step but rather applied for all tests. For example, some low level requirements can be covered by high level tests, i.e., structural coverage are measured on all tests levels.

Testing of safety critical software has two objectives. First objective is to demonstrate that the software satisfies its requirements. The second objective is to demonstrate with a high degree of confidence that errors which could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed. To satisfy the software testing objectives:

a. Test cases should be based primarily on the software requirements.
b. Test cases should be developed to verify correct functionality and to establish conditions that reveal potential errors.
c. Software requirements coverage analysis should determine what software requirements were not tested.
d. Structural coverage analysis should determine what software structures were not exercised.

Figure 1, a diagram of the software testing process[3]. The objectives of the three types of testing are described below.

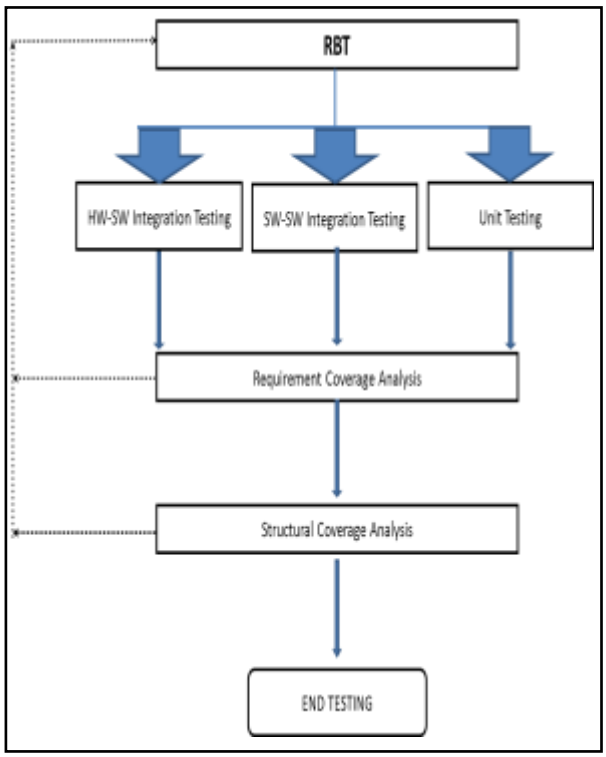

Figure 1- RBT process-Safety Critical Systems

A. Hardware-Software Integration Testing

The objective of requirements-based hardware/software integration testing is to ensure that the software in the target computer will satisfy the high-level requirements. Typical errors revealed by this testing method include:

- Incorrect interrupt handling.
- Failure to satisfy execution time requirements.
- Incorrect software response to hardware transients or hardware failures. For example, start-up sequencing, transient input loads and input power transients.
- Errors in hardware/software interfaces.

B. Software -Software Integration Testing

The objective of requirements-based software integration testing is to ensure that the software components interact correctly with each other and satisfy the software requirements and software architecture. This method may be performed by expanding the scope of requirements through successive integration of code components with a corresponding expansion of the scope of the test cases. Typical errors revealed by this testing method include:

- Incorrect initialization of variables and constants.
- Parameter passing errors.
- Data corruption, especially global data.
- Inadequate end-to-end numerical resolution.
- Incorrect sequencing of events and operations.

C. Unit Testing

The objective of requirements-based low-level testing is to ensure that the software components satisfy their low-level requirements. Typical errors revealed by this testing method include:

- Failure of an algorithm to satisfy a software requirement.
- Incorrect loop operations.
- Incorrect logic decisions.
- Failure to process correctly legitimate combinations of input conditions.
- Incorrect responses to missing or corrupted input data.
- Inadequate algorithm precision, accuracy or performance.

D. Requirements Coverage Analysis

The objective of this analysis is to determine how well the RBT verified the implementation of the software requirements. This analysis may reveal the need for additional RBT cases. RBT coverage analysis shows the following:

- Test cases exist for each software requirement.
- Test cases satisfy the criteria of normal and robustness testing.

E. Structural Coverage Analysis

The objective of this analysis is to determine which code structure was not exercised by the requirements-based test procedures. The RBT cases may not have completely exercised the code structure, so structural coverage analysis is performed and additional verification produced to provide structural coverage.

- The analysis should confirm the degree of structural coverage appropriate to the software level.
- The structural coverage analysis may be performed on the source code. The analysis should confirm the data coupling and control coupling between the code components.

Structural coverage analysis reveals code structure that was not exercised during testing.

III. Requirement Based Test Case Selection

RBT is emphasized because this strategy has been found to be the most effective at revealing errors. Guidance for RBT case selection includes:

- To implement the software testing objectives, two categories of test cases should be included: normal range test cases and robustness (abnormal range) test cases.
- The specific test cases should be developed from the software requirements and the error sources inherent in the software development processes.

A. Normal Range Test Cases

The objective[3]of normal range test case is to demonstrate the ability of the software to respond to normal inputs and conditions. Normal range test cases include:

- Real and integer input variables should be exercised using valid equivalence classes and boundary values.
- For time-related functions, such as filters, integrators and delays, multiple iterations of the code should be performed to check the characteristics of the function in context.
- For state transitions, test cases should be developed to exercise the transitions possible during normal operation.
- For software requirements expressed by logic equations, the normal range test cases should verify the variable usage and the Boolean operators.

B. Robustness Test Cases

The objective[3] of robustness test cases is to demonstrate the ability of the software to respond to abnormal inputs and conditions. Robustness test cases include:

- Real and integer variables should be exercised using equivalence class selection of invalid values.
- System initialization should be exercised during abnormal conditions.
- The possible failure modes of the incoming data should be determined, especially complex, digital data strings from an external system.
- For loops where the loop count is a computed value, test cases should be developed to attempt to compute out-of-range loop count values, and thus demonstrate the robustness of the loop-related code.
- A check should be made to ensure that protection mechanisms for exceeded frame times respond correctly.
- For time-related functions, such as filters, integrators and delays, test cases should be developed for arithmetic overflow protection mechanisms.
- For state transitions, test cases should be developed to provoke transitions that are not allowed by the software requirements.

IV. RBT -Test case Selection Example

The following example illustrates how to derive test data from a requirement. One sample requirement has been taken to illustrate it.

Requirement001:

Software SHALL set FAULT, if fail_Counter is greater than 100.

V. Proposed RBT Process For Non safety critical system softwares

Below is the Proposed RBT process for Non safety critical systems. It is a 13 step process and each of the steps is described below.

1. Understand the requirements
2. Review the requirements
3. Partition the requirements for testing
4. Generate one-line test cases
5. Review the one-line test cases
6. Design the test cases
7. Review the test cases
8. Design the test procedures
9. Review the test procedures
10. Test execution
11. Test results review
12. Generate traceability matrix
13. Review the traceability matrix

1. Understand the Requirements

In RBT, the understanding of requirements is very much required for each tester. RBT will not make sense without clear knowledge of requirements. It is the tester's responsibility is to make sure that each requirement is clearly understood.

Supported data for Requirement001:

Assume as per Data Dictionary Data type for "fail_Counter" is UINT16.

Range for "fail_Counter" is [0, 200]

Test data derivation:

UINT16 data type range: [0, 0xFFFF]

"fail_Counter" range: [0, 200]

Normal range: [0, 200]

Robust range: [201,0xFFFF]

Normal test data:

fail_Counter = 0 (Lower Boundary)

fail_Counter = 200 (Upper Boundary)

fail_Counter = 150 (Middle value any)

As fail_Counter is compared with 100, fail_Counter should be exercised for following values to ensure the relational operator testing.

fail_Counter = 99 (100 -1)

fail_Counter = 100 (100)

fail_Counter = 101 (100+1)

Robustness test data:

fail_Counter = 201 (Lower Boundary)

fail_Counter = 0xFFFF (Upper Boundary)

Table 1 shows the test data for "fail_Counter"

Table 1- Example- Test values

Variable	Nominal range	Nominal test values	Robustness range	Robust test values
fail_Counter	[0, 200]	0,150,200 99,100,101	[201, 0xFFFF]	201, 0xFFFF

2. Review the requirements

Tester has to perform the requirements review to ensure following objectives. Review requirements in advance to testing will helps in the early fault detection

- Comply with associated requirement specifications
- Correct, complete and consistent
- Compatible with target computer
- Verifiable
- Conform to standards
- Traceable
- Algorithms are accurate

3. Partition the Requirements for Testing

Partition the requirements logically to make testing simple. For example, all functional requirements, all communication related requirements, and all maintenance requirements can be grouped together. It makes test design simple.

4. Generate One-line Test Cases

One-line test cases are test cases which describes the test objective alone. Test objective denotes the purpose of the test case. One-line test case generation

will be the first step to develop the test cases. One-line test cases contain test objective and the requirement identifier (which it is traced to).

5. *Review the One-line Test Cases*

Review the one-line test cases against the requirements. One-line test case review has to ensure the following objectives.

- All one-line test cases are derived from requirements
- All one-line test cases are correct, complete and consistent
- Each requirement has at least one one-line test cases
- Traceability between one-line test cases and requirements are correct.

6. *Design the Test Cases*

Design the test cases with the help of one-line test cases. Test case design should ensure the following objectives.

- All normal range and robustness range scenarios are included.
- Each requirement is completely tested by one or more test cases.

7. *Review the Test cases*

Review the test cases against the requirements. Test case review in advance to test execution will avoid multiple iterations in testing. Test case review has to ensure the following objectives.

- Test coverage of requirements is achieved.
- Test coverage of normal range inputs, boundary inputs, out-of-range inputs, equivalence classes, and Boolean.
- Relational operator scenarios.
- Test coverage for Structural coverage analysis is achieved.
- Test coverage for data and control coupling is achieved.

8. *Design the Test Procedures*

Design the test procedure for each test case. Main parameters for test procedure design are the test environment and verification criteria. Test procedure should be clear and simple. Each test procedure should have a unique id to map with its associated test case.

9. *Review the Test procedures*

Review the test procedure against test cases and test environment configuration. Test procedure review has to ensure the following.

- Each test procedure is clear.
- Each test procedure has covered all test verification criteria.
- Each verification statement has PASS/FAIL criteria.
- Each test procedure has overall PASS/FAIL criteria.

10. *Test Execution*

Execute the test cases as per the test procedures in intended test environment. Capture the test results in required format.

11. *Test Results Review*

Review the test results against test cases. Ensure the following objectives in Test results review.

- All test cases are executed.
- Actual output and expected output are same.
- Each verification criteria has marked with PASS/FAIL.
- Overall test PASS/FAIL is marked.

12. *Generate Traceability Matrix*

Generate traceability matrix for test case versus requirements and requirement versus test cases. Traceability matrix will reveal the following.

- Each test case has associated requirement trace or not.
- Each requirement has associated test cases trace or not.

13. *Review the Traceability Matrix*

Traceability matrix review ensures that, all test cases have traced to associated test cases and all requirements have traced to associated test cases.

VI. Safety Critical Vs non Safety Critical

The succinct difference between safety critical system software and non-system critical system[2] is as follows.

- Safety critical software involving the potential for loss of life due to software failure.
- Non-safety critical software involving the potential for aborting a mission due to software failure.

In brief, the goal of safety critical software is containment, whereas a primary aim of non-safety critical software is efficiency and other quality attributes and less on the safety issues of hazards and mishaps that can endanger human life and property.

VII. RBT for non Safety Critical Systems

However, as per the standards, RBT is mandatory for safety critical systems like aerospace, military systems, automotive systems and medical systems. But developers of non-safety critical systems can also benefit from the proposed RBT process (13 step process).

Software development often proves far more expensive than expected; bugs discovered late in the development cycle costs more and risk the integrity and safety of a system, especially if the software has been deployed. Obviously, careful planning, organization and a team with the correct skills will help.

Evidence indicates that the earlier a defect is discovered in development the less impact it has on

both the timescale and cost. There is much to gain by ensuring that requirements are captured in full, they are well understood, and they are specified completely and unambiguously.

It is a general perception in non-safety critical system development, that use of the above specified process may involve huge cost and time; hence we tend to escape this process. If cost is the only factor which prevents following the suggested approach it can be tailored for non-safety critical systems. A lightweight process can be tailored for non-safety critical systems by eliminating Step3 (partition the requirements for testing), Step4 (generate one-line test cases) and Step5 (review the One-line test cases).

However, per the historical data, the analysis reveals that the cost and the time spent in this process will be far less than the cost involved in fixing the issues/bugs raised at the later stage.

The cost of fixing[4] an error is cheaper when it is found at an earlier stage. It results in rework. If a defect is introduced while coding, you fix the code and re-compile. However, if a defect has its origin in poor requirements and is not discovered until integration testing, then you must re-work the requirements, re-work the design, re-work the code and re-work the tests. It is all this "re-work" that sends projects over budget and over schedule. Assume the cost of fixing the defect as 1X for fixing a defect in requirements phase which is found in the same phase. If that same defect is not found until production it will cost hundreds or even thousands of times more.

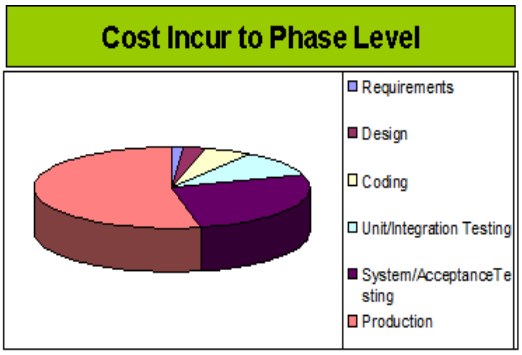

Figure 2 and Table 2 illustrates it.

TABLE 2 – COST INCUR TO PHASE LEVEL

Phase	Cost ratio
Requirements	1
Design	10
Coding	20
Unit/Integration Testing	40
System/Acceptance Testing	100
Production	200

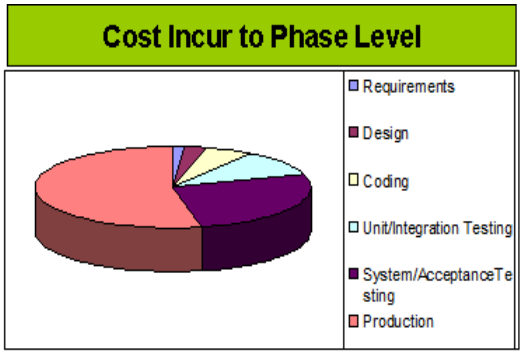

Figure 2 - Graphical Representation-Cost Incur to Phase Level

Here are the benefits, if the proposed RBT is applied for non-safety critical systems.

- Reduces the cost to deliver by early fault detection.
- Reduces the time to deliver by allowing parallel testing with the development activities.
- Improves the overall quality by early fault detection.
- Minimizes rework by early fault detection.
- Delivers maximum coverage with the optimized number of test cases.
- Provides quantitative test progress metrics.

VIII. CONCLUSION

The proposed 13 step RBT process would benefit the non-safety critical systems in terms of cost, time and quality, if they are to be applied. Introducing RBT best practices is usually more realistic for non-safety critical systems. The proposed system drives high quality requirements and early fault detection is likely to reduce the development cost.

REFERENCES

[1] P. G. Gutgarts and A. Termin, "Security-Critical versus Safety-Critical Software," Proc. of the IEEE Homeland Security Technology Conference, Waltham, MA, pp. 507–511, November 2010

[2] 'Software considerations in airborne systems and equipment certifcation'. Document RTCA/DO-178B, RTCA Inc.,December 1992.

[3] Requirements Based Testing Process Overview© 2009 Bender RBT Inc

[4] "Requirements-based testing: an overview" Mogyorodi, G."Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on Digital Object Identifier: 10.1109/TOOLS.2001.941681" Publication Year: 2001 , Page(s): 286- 295

[5] "Requirement-based test case generation and prioritization" Salem, Y.I.; Hassan, R. Computer Engineering Conference (ICENCO), 2010 International Digital Object Identifier: 10.1109/ICENCO.2010.5720443 Publication Year: 2010 , Page(s): 152 – 157

[6] "Designing Generic Safety Test Cases for Airborne Software" Changyong Yang,Xiaohong Bao,Deming Zhong, Zhen Li, 2011 IEEE Publication Year: 2011, Page(s): 737 – 741

[7] "Issues On Software Testing For Safety-Critical Real-Time Automation Systems" Lingfeng Wang, 2004 IEEE Publication Year: 2004, Page(s): 10.B.2-1 – 10.B.2-12

[8] "Requirement Based Test Case Prioritization", R.Kavitha,V.R.Kavitha, Dr.N.Suresh Kumar, 2010 IEEE Publication Year: 2010, Page(s): 826 – 829

[9] "Test Case Prioritization", G. Rotherrmel, Untch C.Chu, M. Harrold, IEEE Transactions on Softare Engineering 27(10)(2001), Page(s): 929-948

[10] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In TACAS, pages 367–381, 2008.

[11] M. L. Bolton and E. J. Bass. Formally verifying human-automation interaction as part of a system model: limitations and tradeoffs. ISSE, 6(3):219–231, 2010.

[12] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu. Using formal methods to predict human error and system failures. In Proceedings of the 2nd Applied Human Factors and Ergonomics International Conference, pages 14–17, July 2008.

[13] J. C. Campos and M. D. Harrison. Model checking interactor specifications. Automated Software Engineering, 8(3–4):275–310, 2001.

[14] J. C. Campos and M. D. Harrison. Systematic analysis of control panel interfaces using formal tools. In Proceedings of the 15th International Workshop on the Design, Verification and Specification of Interactive Systems, number 5136 in Lecture Notes in Computer Science, pages 72–85. Springer-Verlag, July 2008.

[15] L. A. Clarke. A program testing system. In Proceedings of the 1976 annual conference, ACM 76, pages 488–491, 1976.

[16] P. Curzon, R. Rukˇs˙enas, and A. Blandford. An approach to formal verification of human-computer interaction. Formal Aspects of Computing, 19(4):513–550, Nov. 2007.

[17] M. S. Feary. A toolset for supporting iterative human – automation interaction in design. Technical Report 20100012861, NASA Ames Research Center, Mar. 2010.

[18] M. S. Feary. A toolset for supporting iterative human automation interaction in design. Technical Report 20100012861, NASA Ames Research Center, 2010.

[19] D. Giannakopoulou, D. Bushnell, J. Schumann, H. Erzberger, and K. Heere. Formal testing for separation assurance. Annals of Mathematics and Artificial Intelligence, pages 1–26, 2011. 10.1007/s10472-011-9224-3.

[20] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in udita. In ICSE, pages 225–234, 2010.

[21] J. C. King. Symbolic execution and program testing. Communications of the ACM, 19(7):385–394, 1976.

[22] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In ICSM, pages 1–10, 2010.

[23] C. Pˇasˇareanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In Proceedings of the IEEE/ACM international conference on Automated software engineering, pages 179–180. ACM, 2010.

[24] C. S. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. STTT, 11(4):339–353, 2009.

[25] C. S. Pˇasˇareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In ISSTA, pages 15–25, 2008.

[26] J. Rushby. Using model checking to help discover mode confusions and other automation surprises. Reliability Engineering and System Safety, 75(2):167–177, Feb. 2002.

[27] H. Thimbleby. Press On: Principles of Interaction Programming. The MIT Press, Nov. 2007.

[28] H. Thimbleby and J. Gow. Applying graph theory to interaction design.In J. Gulliksen, editor, Engineering Interactive Systems 2007/DSVIS 2007, number 4940 in Lecture Notes in Computer Science, pages 501–518. Springer-Verlag, 2008.

[29] S. Thummalapenta, M. R. Marri, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. In FASE, pages 294–309, 2011.

[30] O. Tkachuk, G. Brat, and W. Visser. Using code level model checking to discover automation surprises. In Digital Avionics Systems Conference (DASC), 2002.

[31] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. Automated Software Engineering, 10(2):203–232, 2003.

[32] X. Yuan and A. M. Memon. Generating event sequence-based test cases using gui runtime state feedback. IEEE Trans. Software Eng., 36(1):81–95, 2010

[33] VE. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: issues, an approach, and case study," IEEE Transactions on Software Engineering (TSE), vol. 26, no. 12, pp. 1147–1156, 2000.

[34] J. Palmer, "Designing for web site usability," Computer, vol. 35, no. 7, pp. 102–103, 2002.

[35] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in International Conference on Software Maintenance (ICSM), 2008, pp. 307–316.

[36] K. C. Foo, J. Zhen Ming, B. Adams, A. E. Hassan, Z. Ying, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in International Conference on Quality Software (QSIC), 2010, pp. 32–41.

[37] W. Shewhart, Economic control of quality of manufactured product. Van Nostrand, NY: American Society for Quality Control, 1931.

[38] K. C. Foo, "Automated discovery of performance regressions in enterprise applications," Master's thesis, School of Computing, Queen's University, 2011.

[39] H. Malik, "A methodology to support load test analysis," in International Conference on Software Engineering (ICSE). Cape Town, South Africa: ACM, 2010, pp. 421–424.

[40] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating instrumentation data to system states: a building block for automated diagnosis and control," in Symposium on Opearting Systems Design Implementation. San Francisco, CA: USENIX Association, 2004, pp. 231–244.

Selected Paper from International Conference on Computing (NECICC-2k15)