Scenario-Based Functional Regression Testing

Wei-Tek Tsai*, Xiaoying Bai*, Ray Paul**, Lian Yu [+]

*Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85259

**OASD C3I Investment and Acquisition
Washington, D.C.

[+]Department of Industrial Engineering
Arizona State University
Tempe, AZ 85259

Abstract

Regression testing has been a popular quality assurance technique. Most regression testing techniques are based on code or software design. This paper proposes a scenario-based functional regression testing, which is based on end-to-end (E2E) integration test scenarios. The test scenarios are first represented in a template model that embodies both test dependency and traceability. By using test dependency information, one can obtain a test slicing algorithm to detect the scenarios that are affected and thus they are candidates for regression testing. By using traceability information, one can find affected components and their associated test scenarios and test cases for regression testing. With information of dependency and traceability, one can use the ripple effect analysis to identify all affected, including directly or indirectly, scenarios, and thus the set of test cases can be selected for regression testing. This paper also provides several alternative test-case selection approaches and a hybrid approach to meet various requirements and restrictions. A web-based tool has been developed to support these regression tasks.

1 Introduction

Software system is subject to changes through out development, maintenance, and evolution, due to a variety of reasons such as changed requirements (both functional and nonfunctional), update technology, and upgrade hardware or software platform [15,19,30,40,44,46]. The changes bring much risk to the software system because change propagations may introduce new bugs, sometimes even fatal errors [28]. Regression testing is the technique to verify the integrality and correctness of the modified system. There exist two strategies for selecting regression test cases: retest-all and selective-retest. In most of situations, it is impossible to retest all the test cases of the system under test. Therefore, the selective testing is necessary.

Many techniques have been proposed for selective regression testing. Yau and Kishimoto presented a technique based on input partitioning and cause-effected graphing [52]. Fischer and his colleagues established a linear programming model based on four matrices: connectivity, reachability, test-case dependency (or cross-reference), and variable set/use, reflecting various views of the program such as control flow and test coverage [16]. This model was further extended to include dataflow information [20]. Gupta, Harrold and Soffa [17] examined traditional program slicing [41], while Bates and Horwits [3] exercised program dependency graphs, to select regression tests based on program dataflow and control flow analysis. Agrawal, Horgan, Krauser and London presented a set of slicing techniques with the purpose to determine the test cases on which the new and the old program may produce different output [2]. Chen and Tsai proposed class, message, constrained and recursive slicing techniques for maintaining and understanding C++ programs based message, class and declaration dependencies [11].

Most of the existing regression test selection techniques are code-based using program slicing

[2,17,44], program dependence graphs [3,7], data flow and control flow analysis [21]. This paper presents a test scenario-based methodology for functional regression testing in the context of End-to-End (E2E) integration testing, which focuses on the functional correctness of large integrated system from the end users' point of view [45,55].

Section 2 introduces the E2E testing and presents the test scenario model, including template definition, dependencies, and traceability. Section 3 describes the model-based regression test selection technique supported by test scenario slicing and characterized ripple effect analysis. Section 4 briefly outlines the regression testing support tool. Section 5 draws conclusion of the paper.

2 Test Scenario Model

This section presents the E2E testing specification, and describes the test scenario model from three perspectives: test scenario specification, the dependencies between test scenarios, and the traceability to other software artifacts.

2.1 End-to-End Testing

E2E testing is focused on the functionality of the integrated system from the end user's viewpoint. It is different from integration testing which can be performed on any subset of subsystems. It assumes that both module testing and integration testing have been performed and approved, possibly including integration testing at multiple levels. It is to verify that the integrated software, include internal subsystems and external supporting systems, will collectively support the organizational core business functions [45].

E2E testing is a life cycle process that is made up of following procedures [55]:

?? *Test planning*: specifies the key tasks as well as the associated schedule and resources;

?? *Test design*: designs an E2E testing, including test specifications, test case generation, risk analysis, usage analysis, and scheduling tests;

?? *Test execution*: executes the test cases and documents the testing results; and

?? *Result analysis*: analyzes the testing results, evaluates testing and performs additional testing if necessary.

E2E testing specifications, defined in thin-threads and conditions, are semi-formal representations of restructured and refined software requirements. A thin thread is a minimum usage scenario of the integrated system from the end user's point of view. Conditions identify the factors that will affect the execution of the functionality by a thin thread. Both thin threads and conditions are identified based on a thorough understanding of the system under test, including system functionality (requirements), the connection (static topology) and interactions (dynamic communication structure) between subsystems in the integrated system. Therefore, in addition to functional requirements, thin threads and conditions also carry information of software configuration.

A thin-thread serves as a basic test scenario with input, output and actions specified. By applying sequencing and structural computations on basic and/or existing scenarios, one can also generate complex test scenarios. Test cases are generated from test scenarios by associating the input(s) of a test scenario with actual data based on various test criteria.

2.2 Test Scenario Specification

Test scenarios are semi-formal specifications that sit between descriptive system requirements and executable test cases. The test scenarios used in this paper differ from the common sense scenarios in two ways: (1) they describe detailed test requirements in terms of data, conditions and constraints; (2) In addition to normal inputs, they also capture abnormal inputs and exceptions that the system should also handle reliably.

Test scenario is described as a template as shown in Table 1.

Section	Attributes	Representation
General	ID	String Id;
	Name	String Na;
	Description	String Des;
Policy	Test Strategy	String TSt;
	Test Criteria	String TCr;
Input/Output	Input	{{idat},act}; //multi-entry
	Expected Output	{{odat},deli}; //multi-entry
Execution	Modules Involved	{md }; //multi-entry
	Interfaces Involved	{inf}; //multi-entry
	Persistent Data	{pd}; //multi-entry
	Execution Path	{pa}; // ordered set
Conditions	Pre-conditions	{pr}; //multi-entry
	Post-conditions	{po}; //multi-entry
Linkage	Requirements	{req}; //multi-entry
	Test Scripts	{ts}; //multi-entry
Others	Status	String St;
	Agents	{ag}; //multi-entry
	Schedules	{sc}; //multi-entry
	Risks	float Ri;

Table 1. Template for Test Scenario definition

?? *General* describes identification of a test scenario with ID and Name, as well as a brief summary of the context and objective of the test scenario.

?? In *Policy*, user defines the testing policy in terms of test *strategy* and test *criteria*.

?? Attribute *Input* identifies the set of data to be exercised, and the action that triggers the execution. Attribute *Expected Output* identifies the expected output data and execution delivery.

?? *Execution* defines the set of software components that are verified by the test scenario composed of *Modules*, *Interfaces*, *Persistent data* storage, and *Execution Path* through these components.

?? *Condition* defines the pre- and post-conditions for a test scenario. *Pre-conditions* are those factors that must be met before the execution, and *Post-conditions* are the expected system status after the execution.

?? *Linkage* involves information for a test scenario to trace to *Requirements* and *Test Scripts*.

?? This template also provides other attributes, which are important for test management: *Status*, *Agents*, *Schedules* and *Risk*.

The number of test scenarios is large. For better management and understanding, test scenarios are arranged hierarchically into a tree structure [45]. Test scenarios that share certain commonalities can be grouped together. This grouping can be recursive, that is, a collection of lower-level scenario groups can be further grouped into a higher-level group.

2.3 Test Scenario Dependencies and Traceablity

Dependence analysis provides the foundation for regression testing and ripple effect analysis. The following lists the kinds of dependence:

Functional Dependence: A functional requirement may be tested from various perspectives, such as normal input, abnormal input and exception handling, and represented by a set of test scenarios. Thus functional dependence identifies how a set of test scenarios is related to each other with respect to a functional requirement.

Input Dependence: Input dependence identifies the common inputs shared by a set of test scenarios, including input data, actions and triggering events. If any change, such as data definition and validation rules, is made to these inputs, all these test scenarios need to be examined and possibly re-run.

Output Dependence: Similar to input dependence, test scenarios may also share common output, either data or messages. If a change is made to an output, all the test scenarios that take it as their expected outputs are subject to change.

Input/Output Dependence: An output of a test scenario may be an input of another test scenario, and any change to one of the pair of test scenarios may affect the other one. Input/Output dependence is used to capture the data produce-usage relationships between test scenarios. It is important when a sequence of test scenarios is executed in one session to test a complex usage of the system.

Persistent Data Dependence: In addition to user input data, test scenarios may be input dependent, output dependent, or input/output dependent through persistent data from files or databases. Any change to the persistent data, such as definition change, configuration change, and content change, will affect the test scenarios that use the data. Persistent data dependence is used to analyze the use relationships between test scenarios and persistent data.

Execution Dependence: Execution dependence captures component and interaction relationships among the individual execution paths of test scenarios. Test scenarios may be execution dependent in three ways: identical paths, covered paths and crossing paths.

Condition Dependence: Test scenarios may share either the pre-conditions or post-conditions, or both. Test scenarios, which are involved in pre-condition dependent, may start from the same system status, triggering event, and environment setting. While for test scenarios that are post-condition dependent, their executions may result in the same system status.

The traceability enables global change impact analysis among software artifacts In *Linkage* of the temple, attribute *Requirement* records all the requirements what the test scenario is associated; attribute *Test Script* stores all the test cases that are generated from the test scenario. Test scenarios are also traced to software elements, including subsystem components, interfaces and data, specified in *Execution* of the template.

In general cases, a requirement can be traced to multiple test scenarios and a test scenario to multiple test cases (test scripts). Each test scenario may cover multiple software components along its execution path; and each software component may be tested by multiple test scenarios from various aspects.

3 Scenario-Based Regression Testing

Different to program slicing, which are performed on source code, test scenario slicing in ths paper is performed on test scenarios and enables global change impact analysis based on dependency and traceability analysis. Using test scenario slicing level-by-level, *Ripple Effect Analysis* technique directs the iterative process of regression test selection and revalidation.

3.1 Test Scenario Slicing

Definition 1. *Slicing Criterion:* A slicing criterion is a 2-tuplet, denoted as |?|?|, meaning that upon the given attribute ?, slice corresponding attribute ?.

The attributes in the two fields are identical, except that when a tester analyzes the input/output dependence, where the input attribute and output attribute flip over in the two fields.

Definition 2. *Slice:* Given test scenario *s*, and slicing criterion |*s*.?|s_i.?| (s ? s_i), search scenario(s) s_i that match with the given scenario's attribute *s*. ?.

The results of a scenario slice are a set of scenario(s), which may full-match, or semi-match the slicing criterion, dependent on that the slicing criterion is a simple attribute or a complex attribute.

3.2 Ripple Effect Analysis

Ripple Effect Analysis (REA) is used to analyze and eliminate side effects due to changes and to ensure consistency and integrity after changes are made to software [27,43,51]. It is an iterative process of change request, software modification, impacts identification and validation. It ends when there are no more ripples.

The notion of ripple effect analysis, originally studied in the context of software programs, can be extended to all software artifacts including specifications, design, and test cases. Also, it can be extended to different artifacts across different phases of the software life cycle. In particular, the REA process is not specific to any particular programming language or design paradigm.

In this research, two perspectives characterize the REA process:

?? Keep consistence among scenarios, software components, and requirements, no matter where the change are initiated. Whenever there is an inconsistence, modifications and impact analysis are required.

?? At each iterate of the process, impacts are identified and validated using test scenario slicing with various slicing criteria.

The following steps illustrate the characterized REA process for regression test selection.

1. Associate a change request, wherever it is originated from, with a set of test scenarios. If changes come from requirements and implementations, a tester can identify the test scenarios through the traceability analysis.
2. Trace each of the test scenarios identified in step 1 to software components through the traceability analysis.
3. Identify and revalidate the affected software components correspondingly.
4. Revalidate the test scenarios identified with respect to inputs, outputs, conditions, test data and configurations. For each of the test scenarios that need to be changed, check out the corresponding test cases.
5. For each of the test scenarios, determine slicing criteria based on its changes. Perform slicing to identify the potentially affected test scenarios with the criteria. The following are some policies for determining slicing criteria:
   ? ? Slicing with all of the defined slicing criteria. This is to identify all the dependent test scenarios.
   ? ? Slicing with only some slicing criteria. If an experienced test engineer is sure that changes on some parts of the test scenario will not affect the other parts and will not propagate along certain directions, he can perform impact analysis with a subset of slicing criteria. For example, suppose only the inputs of a test scenario need to be changed such as a layout of the interface, the tester may slice with only input slicing criterion and backward input/output slicing criterion.
   ? ? Slicing by whole set of attribute values. This is to select all the test scenarios that are dependent with respect to the attribute.

- ? ? Slicing by a subset of values. This is to select all the test scenarios that are dependent with respect to some specific aspects of the attribute.
- ? ? Hybrid approach. For some test criteria, the tester may perform slicing by whole set of attribute value, while for the others, slicing by a selected subset of attribute values.

6. If there is any scenario identified in step 5, go to step 2; otherwise prepare for the following regression testing. The results of this process are:
   a. A set of modified software components;
   b. A set of affected test scenarios; and
   c. A set of test cases corresponding to the affected test scenarios.

A partial example in Figure 1 shows how a change request is propagated among test scenarios, and traced to software components and test cases. The change request is first mapped to two scenarios s1 and s2. s1 is traced to software components c1 and c3, and s2 is traced to c4. c1 and c4 are identified for changes. Performed scenario slicing with certain slicing criterion, s3, s4, s5, and s6 are identified as potentially affected scenarios, and s3, s5 and s6 are validated as candidates to changes where s3 tests c1 and s6 tests c4. Because there is no change to s4, stop impact analysis on s4; while s3, s5, and s6 need further to be propagated using scenario slicing technique. Test cases are also identified by the traceability analysis from test scenarios to test cases, such as from s2 to t1 and from s6 to t2.

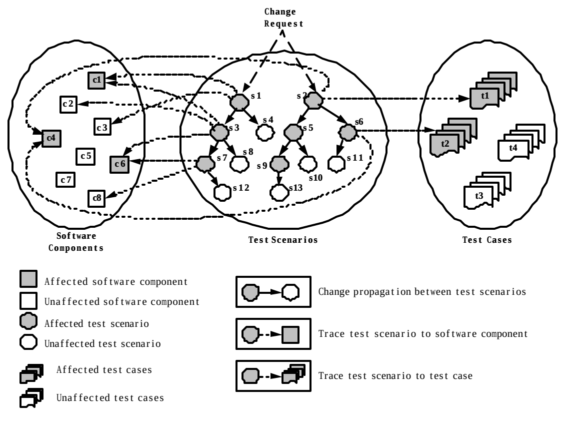

Figure 1. Change impacts Analysis

3.3 Hybrid Regression Testing

Incorporating other testing criteria, this subsection also provides the following hybrid strategies for selecting and scheduling regression tests to balance cost and reliability.

- ?? *Risk-Based Strategy*: Selects high-risk test scenarios, which have high probabilities to fail and/or cause serious consequences in case of fail, and to exercise them with high priority.
- ?? *Usage-Based Strategy*: Selects highly used test scenarios as candidates for regression testing. This strategy has been advocated in the Cleanroom methodology [14].
- ?? *Random Testing*: Selects a sample set of test scenarios. If the sample size is large enough, it is possible to achieve reasonable statistical confidence. This is the basis for reliability testing and assurance based testing [44].
- ?? *Time-Wise Strategy*: Regression testing may be performed at different stages of software development, such as daily build, weekly build, milestone, and major release.

4 Tool Support

A web-based tool has been developed to support the regression testing. The tool has a three-tier architecture and runs on the J2EE platform using Enterprise JavaBean (EJB), as shown in Figure 3.

- ?? At the backend lie the file/database servers, the storage of test scenarios model, change request and regression report;
- ?? The middle layer performs the core functions of test data management, analysis, and query, such as test scenario definition management, dependence management, traceability management, impact analysis, and report generation;
- ?? The front tier is the presentation layer. It provides users with various graphical views of the test data and analysis results.

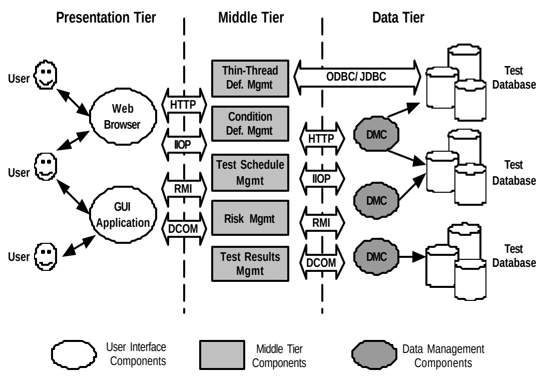

Figure 2. Tool Architecture

5 Conclusion

Regression testing on large integrated software system is hard because the number of test cases is huge and the change impacts may be widely spread. For high-level integration regression testing, the retest-all approach is usually time and resource consuming.

To counter this challenge, this paper proposes a test scenario based approach for functional regression testing. Test scenarios are semi-formal representations of detailed system requirements, with test inputs, outputs, and conditions defined. By using test dependency information, one can obtain a test slicing algorithm to detect the scenarios that are affected and thus they are candidates for regression testing. By using traceability information, one can find affected components and their associated test scenarios and test cases for regression testing. With information of dependency and traceability, one can use the ripple effect analysis to identify all affected, including directly or indirectly, scenarios, and thus the set of test cases can be selected for regression testing.

Reference

[1] H. Agrawal and J. Horgan, “Dynamic Program Slicing”, Proceedings of the ACM SIGPLAN’90 Conference on Programming Language Design and Implementation, 1990, pp. 246-256.

[2] H. Agrawal, J.R. Horgan, E.W. Krauser and S.A. London, “Incremental Regression Testing”, Proceedings of the Conference on Software Maintenance, 1993, pp. 348-357.

[3] S. Bates and S. Horwitz, “Incremental Program Testing Using Program Dependence Graphs”, Proc. 20th ACM Symp. of Principles of Programming Languages, Jan. 1993, pp. 384-396.

[4] G. Baradhi and N. Mansour, “A Comparative Study of Five Regression Testing Algorithms”, Proceedings of Software Engineering Conference, Australian, 1997, pp. 174-182.

[5] Boris Beizer, *Black-box testing: techniques for functional testing of software and systems*, John Wiley & Sons, Inc., 1995.

[6] R.V. Binder, *Testing Object-Oriented System: Models, Patterns, and Tools*, Addison-Wesley, 1999.

[7] D. Binkley, “Semantics Guided Regression Test Cost Reduction”, IEEE Transactions on Software Engineering, Vol. 23, No. 8, Aug. 1997, pp.498-516.

[8] S.A. Bohner and R.S. Arnold, *Software Change Impact Analysis*, IEEE Computer Society Press, Los Alamitos, 1996.

[9] G. Booch and T. Quatrani, *Visual Modeling With Rational Rose and UML*, Addison-Wesley, 1998.

[10] J.M. Carroll, *Scenario-Based Design: Envisioning Work and Technology in System Development*, John Wiley & Sons, Inc., New York, 1995.

[11] X. Chen, W.T. Tsai, H. Huang, M. Poonawala, S. Rayadurgam, and Y. Wang, “Omega-An Integrated Environment for C++ Program Maintenance”, Proc. Int’l Conf. Software Maintenance, 1996, pp. 114-123.

[12] Y.F. Chen, D. S. Rosenblum, and K.P. Vo, “TestTube: A System for Selective Regression Testing”, Proc. 16th Int’l Conf. Software Eng., May 1994, pp. 211-222.

[13] D. Harel, “From Play-In Scenarios to Code: An Achievable Dream”, IEEE Computer, Vol. 34, No. 1, Jan. 2001, pp. 53-60.

[14] M. Dyer, *The Cleanroom Approach to Quality Software Development*, Wiley, New York, New York, 1992.

[15] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, “Does Code Decay? Assessing the Evidence from Change Management Data”, IEEE Transactions on Software Engineering, Vol.

27, No. 1, Jan. 2001, pp. 1-12.

[16] K. Fischer, F. Raji and A. Chruscicki, “A Methodology for Re-testing Modified Software”, Proc. National Telecommunications Conf., 1981, B6.3.1-B6.3.6.

[17] R. Gupta, M.J. Harrold and M.L. Soffa, “An Approach to Regression Testing Using Slicing”, Proceedings of the Conference on Software Maintenance, 1992, pp. 299-308.

[18] J. Han, “Supporting Impact Analysis and Change Propagation in Software Engineering Environments”, In Proceeding s of the 8th IEEE International Workshop on Software Technology and Engineering Practice, July 1997, pp. 32-41.

[19] J. Hartmann and D.J. Robson, “Approaches to Regression Testing”, Proceedings of the Conference on Software Maintenance, 1988, pp. 368-372.

[20] J. Hartmann and D.J. Robson, “Techniques for Selective Revalidation”, IEEE Software, Vol. 7, No. 1, Jan. 1990, pp. 31-36.

[21] M.J. Harrold and M.L. Soffa, “An Incremental Approach to Unit Testing During Maintenance”, Proceedings of The Conference on Software Maintenance, 1988, pp. 362-367.

[22] S. Horwitz, T. Reps and D. Binkley, “Interprocedural Slicing Using Dependence Graphs”, ACM Transactions on Programming Languages and Systems, Vol. 2, 1990, pp. 29-60.

[23] H. Huang, W.T. Tsai and S. Subramanian, “Generalized Program Slicing for Software Maintenance”, Proceedings of International Conference on Software Engineering and Knowledge engineering, 1996, pp. 261-268.

[24] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgarrd, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison Wesley, Reading, MA, 1992.

[25] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process,* Addison Wesley, Reading, MA, 1999.

[26] P. C. Jorgensen, *Software Testing: A Craftsman’s Approach*, CRC Press, 1995.

[27] J. K. Joiner and W. T. Tsai, “Ripple Effect Analysis, Program Slicing and Dependency Analysis”, TR-93-84, Computer Science Department, University of Minnesota, 1993.

[28] C. Jones, “Software Change Management”, Computer, Vol. 29, No. 2, Feb. 1996, pp. 80-82.

[29] D. Kulak and E. Guiney, *Use Cases: Requirements in Context*, Addison Wesley, Reading, MA, 2000.

[30] M.M Lehman and L.A. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1985.

[31] H.K.N. Leung and L. White, “Insights into Regression Testing”, Proceedings of the Conference on Software Maintenance, 1989, pp. 60-69.

[32] H.K.N. Leung and L. White, “A Study of Integration Testing and Software Regression at the Integration Level”, Proc. Conf. Software Maintenance, Nov. 1990, pp. 290-301.

[33] V. Matena and B. Stearns, *Applying Enterprise JavaBeans^{TM}: Component-Base Development for the J2EE^{TM} Platform*, Addison-Wesley, Boston, 2000.

[34] A.K. Onoma, W.T. Tsai, M. Poonawala, and H. Suganuma, “Regression Testing in an Industrial Environment”, Communications of the ACM, Vol. 41, No. 5, May 1998, pp. 81-86.

[35] M. Poonawala, S. Subramanian, R. Vishnuvajjala, W.T. Tsai, R. Mojdehbakhsh, and L. Elliot, “Testing Safety-Critical Systems – A Reuse-Oriented Appraoch”, Proc. of 9th International Conference on SEKE, 1997, pp. 271-278.

[36] R.S. Pressman, *Software Engineering: A Practitioner’s Approach*, McGraw Hill College Div., 5th edition, 2000.

[37] Ed Roman, *Mastering Enterprise JavaBeans^{TM} and the Java^{TM} 2 Platform, Enterprise Edition*, John Wiley & Sons, Inc. New York, New York, 1999.

[38] G. Rothermel and M.J. Harrold, “Selecting Regression Tests for Object-Oriented Software”, Proc. Conf. Software Maintenance, Sept. 1994, pp. 14-25.

[39] G. Rothermel and M.J. Harrold, “Analyzing Regression Test Selection Techniques”, IEEE Transactions on Software Engineering, Vol. 22, No. 8, Aug. 1996, pp. 529-551.

[40] R.C. Sugden and M.R. Stens, “Strategies, Tactics and Methods for Handling Change”, Proceedings of 1996 IEEE Symposium and Workshop on Engineering of Computer-Based System, 1996, pp. 457-463.

[41] F. Tip, “A Survey of Program Slicing Techniques”, Journal of Programming Languages, Vol. 3, 1995, pp. 121-189.

[42] W. T. Tsai, W. Xie, I. A. Zualkernan, and S. K. Musukula, "A Framework for Systematic Testing of Software Specifications", Proc. of International Conference on Software Engineering and Knowledge Engineering 1993, pp. 380-387.

[43] W.T. Tsai, R. Mojedehbakhsh, F. Zhu, “Ensuring System and Software Reliability in Safety-Critical Systems”, Proceedings of 1998 IEEE workshop on Application-Specific Software Engineering Technology, ASSET-98, 1998, pp. 48-53.

[44] W.T. Tsai, X. Bai, R. Paul, G. Devaraj and V. Agarwal, “An Approach to Modify and Test Expired Window Logic”, in First Asia-Pacific Conference on Quality Software, 2000, pp. 99-108.

[45] W.T. Tsai, X. Bai, R. Paul, W. Shao, V. Agarwal, T. Sheng, B. Li and J. Honnavalli, “End-to-End Integration Testing Design”, Department of Computer Science, Arizona State University, 2001.

[46] Y. Wang, W.T. Tsai, X.P. Chen and S. Rayadurgam, “The Role of Program Slicing in Ripple Effect Analysis”, in Proc. of Software Engineering and Knowledge Engineering, 1996, pp. 369-376.

[47] Y. Wang, R.Vishnuvajalla and W.T. Tsai, “Sequence Specification for Concurrent Object-Oriented Applications”, Proc. of IEEE WORDS, 1997, pp. 163-170.

[48] M. Weiser, “Program Slicing”, IEEE Transaction on Software Engineering, Vol. 10, No. 4, 1984, pp. 352-357.

[49] L. White and H.K.M Leung, “On the Edge, Regression Testing”, IEEE Micro, Vol. 12, No. 2, April 1992, pp. 81-84.

[50] L.J. White V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha, “Test Manager: A Regression Testing Tool”, Proc. Conf. Software Maintenance, Sept. 1993, pp. 338-347.

[51] S.S. Yau, J.S. Collofello and T. MacGregor, “Ripple Effect Analysis of Software Maintenance”, IEEE COMPSAC’78, 1978, pp. 60-65.

[52] S.S. Yau and Z. Kishimoto, “A Method for Revalidating Modified Programs in the Maintenance Phase”, IEEE COMPSAC’87, 1987, pp. 272-277.

[53] DoD OASD C3I Investment and Acquisition, “Year 2000 Management Plan”, 1999.

[54] DoD OASD C3I Investment and Acquisition, “Repairing Latent Year 2000 Defects Caused by Date Windowing”, 2000.

[55] DoD OASD C3I Investment and Acquisition, “End-to-End Integration Testing Guidebook”, 2001.

[56] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990, 10 Dec. 1990.

[57] “MSCs: ITU-T Recommendation Z.120: Message Sequence Chart (MSC)”, ITU-Y, Geneva, 1996.