A Lightweight Process for Change Identification and Regression Test Selection in Using COTS Components

Jiang Zheng[1], Brian Robinson[2], Laurie Williams[1], Karen Smiley[2]

[1] Department of Computer Science, North Carolina State University, Raleigh, NC, USA
{jzheng4, lawilli3}@ncsu.edu
[2] ABB Inc., US Corporate Research
{brian.p.robinson, karen.smiley}@us.abb.com

Abstract

Various regression test selection techniques have been developed and have shown fault detection effectiveness. The majority of these test selection techniques rely on access to source code for change identification. However, when new releases of COTS components are made available for integration and testing, source code is often not available. In this paper we present a lightweight *Integrated - Black-box Approach for Component Change Identification* (I-BACCI) process for selection of regression tests for user/glue code that uses COTS components. I-BACCI is applicable when component licensing agreements do not preclude analysis of the binary files. A case study of the process was conducted on an ABB product that uses a medium-scale internal ABB software component. Five releases of the component were examined to evaluate the efficacy of the proposed process. The result of the case study indicates that this process can reduce the required number of regression tests by 54% on average.

1. Introduction

Regression testing involves selective re-testing of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [7]. To minimize the time and resource cost of regression testing, a variety of regression test selection techniques have been developed [3, 5]. However, these regression test selection techniques rely on source code, and therefore are not suitable when source code is not available for analysis, such as for COTS components.

COTS software products typically undergo a new release every eight to nine months, with active vendor support for only the latest three releases [2]. Upon receiving the COTS files, users often need to conduct regression testing to determine if a new component or new version of a component will cause problems with their existing software and/or hardware system. However, users of COTS components often do not have access to the source code, only to the binary files and a small set of reference documents. Currently, in this case the functions in the user/glue code (which we call the "user functions" henceforth) that use COTS components would need to be completely retested. This retest-all stategy is prohibitively expensive in both time and resources [5]. North Carolina State University and ABB Corporate Research are collaborating to address the challenge presented by the lack of source code for the reduction and selection of test cases.

Our research objective is to develop a lightweight process for regression test selection for the user functions that use software components when source code of the components is not available. We call our process the *Integrated - Black-box Approach for Component Change Identification* (I-BACCI) process. I-BACCI is applicable when component licensing agreements do not preclude binary code analysis. The input artifacts to the process are the binary code of the components (old and new versions), the source code of the user functions, and the test suite for the user functions. These artifacts are generally available to the COTS user. Once the I-BACCI process is completed, the reduced set of regression test cases can be executed.

In prior research, the first two steps of I-BACCI Version 1 were applied on four releases of an internal ABB product component [24]. In this paper, we report the results of an additional case study conducted with a medium-scale ABB product that uses a medium-scale internal ABB software library. All six steps of I-BACCI Version 2 were applied to five releases of the product and component.

The rest of this paper is organized as follows. Section 2 discusses the background and related work. The I-BACCI process is described in Section 3. Section 4 identifies the limitations of the current approach. Section 5 describes the new case study of applying I-BACCI on the ABB product and its library component. Finally, Section 6 presents conclusions and future work.

2. Background and related work

In this section, we discuss the prior work in software component testing, regression testing, change identification, and firewall analysis.

2.1 Testing of software components

Poor testability, due to the lack of access to the component's source code and internal artifacts, is one of the issues and challenges of component testing [4]. Generally, only black-box tests can be run on COTS software because users do not have access to the source code to analyze the internal implementation. Black-box test cases of COTS components can be based upon the specification documentation provided by the vendor. Alternately, the behavior could be determined by studying the inputs and the related outputs of the component.

Harrold et al. [6] presented techniques that use component metadata for regression test selection of COTS components. Using a controlled example of seven versions of a `VendingMachine` program with a `Dispenser` component, they demonstrated that, on average, 26% of the overall testing effort can be saved by using their technique, with three types of metadata to perform the regression test selection [6]. I-BACCI does not require the collection of this metadata, which the component supplier might not provide. However, Harrold's process may be more applicable when component licensing agreements preclude the binary code analysis needed for I-BACCI.

2.2 Regression test selection

Regression test selection (RTS) techniques attempt to reduce the high cost of retest-all regression testing by selecting a subset of possible test cases [5] which focuses on the software components/functions that have been changed or are most likely to be affected by the change. In the selection of test cases, an RTS technique might not be safe. A *safe* RTS technique guarantees that the subset of tests selected contains all test cases in the original test suite that can reveal faults based upon the modified program [3, 10, 14]. A variety of RTS techniques (e.g. [3, 5])

have been proposed, such as methods based upon path analysis techniques or dataflow techniques. However, these techniques rely upon having information about the source code.

Srivastava and Thiagarajan at Microsoft have developed a test prioritization system, Echelon [16], that prioritizes an application's set of tests based on a binary code comparison. Echelon takes as input two versions of the program in binary form, and the test coverage information of the older version (a mapping between the test suite and the lines of code it executes). Echelon outputs a prioritized list of test sequences (small groups of tests). Although they have not published results of applying Echelon to components, in theory, the tool seems to be applicable to test selection for COTS components. However, Echelon is a large proprietary Microsoft internal product with a significant infrastructure and an underlying bytecode manipulation engine. As will be discussed, I-BACCI is a lightweight, relatively simple process.

2.3 Change identification

A key step in choosing regression tests is applying impact analysis [13] to identify changes between the new release and the previously-tested version with the same source code base. However, most change identification approaches utilize the source code of the old and modified programs [9, 14, 17]. Although a comparison between versions of documentation (such as user manuals, specifications, and samples) is potentially helpful [10, 12], the documentation for COTS components may not reflect all changes, and the implementation may change without necessitating any documentation changes.

Wang et. al. [18] developed the Binary Matching Tool (BMAT) which compares two versions of a binary program without knowledge of the source code changes. The implementation of BMAT is built on Windows NT® for the x86 architecture, using the Vulcan binary analysis tool [15] to create an intermediate representation of x86 binaries. The process enables good matching even with shifted addresses, different register allocations, and small program modifications [18]. BMAT was used by Echelon [16], which is discussed in Section 2.2, to match blocks in the two binaries. However, like Echelon, BMAT is a proprietary tool. We have developed a lightweight non-proprietary Trivial Identifier of Differences in BInary-analysis Text Zapper (TID-BITZ)[1] tool to perform the same function for I-BACCI.

2.4 Firewall analysis

Leung and White [1, 10, 11, 21] developed firewall analysis for regression testing with integration test cases (tests that evaluate interactions among components [7]) in the presence of small changes in functionally-designed software. Firewall analysis has been extended to object-oriented systems and graphical user interfaces [8, 19, 20]. Firewall analysis is intended to limit regression testing to potentially-affected system elements directly dependent upon changed system elements [21, 22]. I-BACCI utilizes firewall analysis for RTS.

Module dependencies, control-flow dependencies, and data dependencies are considered in firewall analysis [21]. Affected areas, including modified functions, structures, and functions that use them, are identified. Dependencies are modeled as call graphs and a "firewall" is drawn around the changed functions on the call graph. All modules inside the firewall are unit and integration tested, and are integration tested with all modules not enclosed by the firewall [21].

[1] http://www4.ncsu.edu/~jzheng4/TID-BITZ/index.htm

Firewall methods can only be guaranteed to select all modification-revealing [14] tests and to be safe if all unit and integration tests initially used to test system components are reliable[2]. However, test suites are typically not reliable in practice [22], so the firewall technique may omit modification-revealing tests and/or may admit some non-modification-traversing tests. White and Robinson [22] have shown firewall to be effective despite these theoretical limitations via empirical studies of industrial real-time systems. These limitations thus should not impair the effectiveness of I-BACCI in practice.

3. I-BACCI

The I-BACCI process is an integration of the firewall analysis RTS method with our Black-box Approach for Component Change Identification (BACCI) process [24] for identifying change. The second version of the I-BACCI process involves six steps as shown in Figure 1. The inputs to the I-BACCI process, which feed into different steps, are shown in gray blocks in Figure 1.

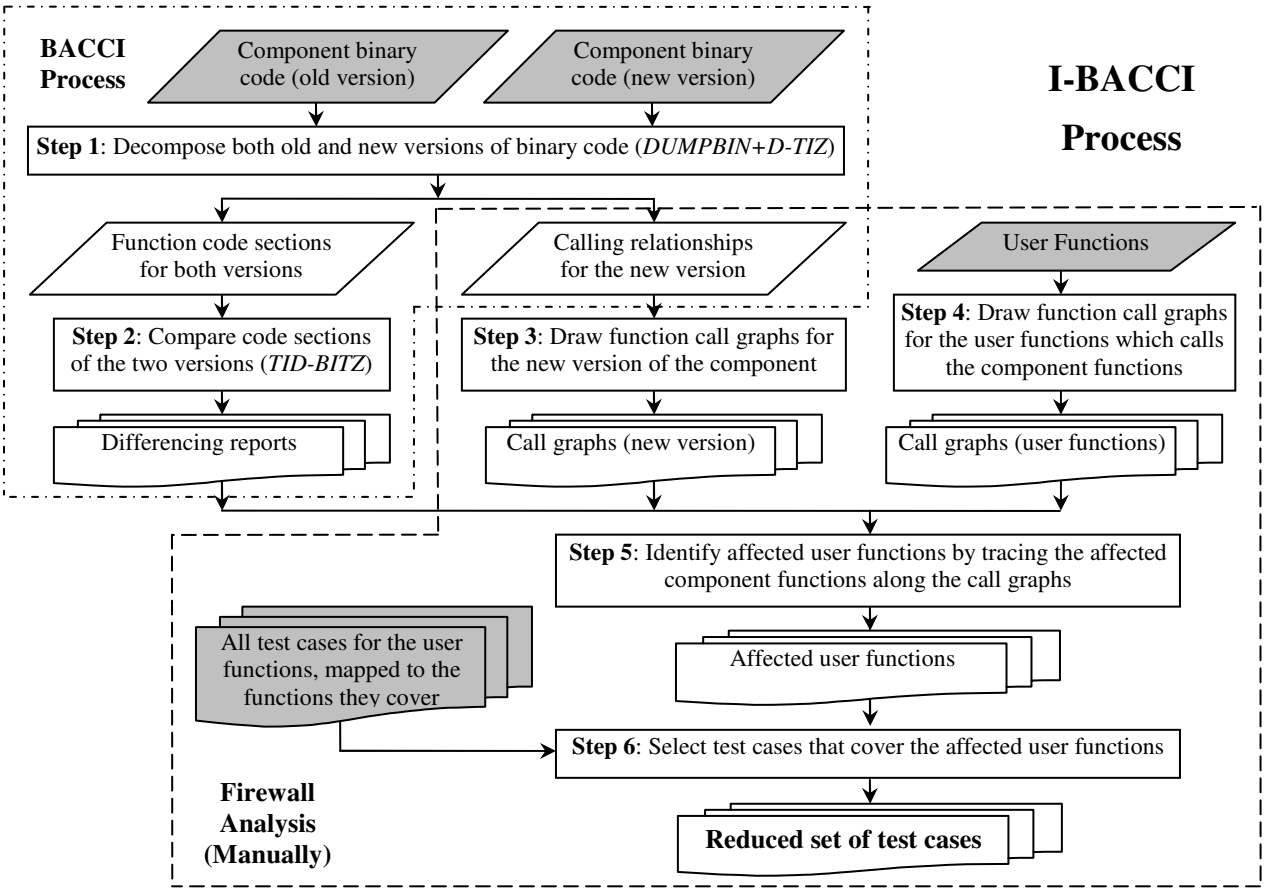

Figure 1: I-BACCI Version 2 Regression Test Selection Process

The first two steps are done via the BACCI process (in dash-dotted line frame), which produces a report on changed functions and the calling relationships among the functions in the components. The remaining four steps are currently done via manual firewall analysis (in dashed line frame), which requires the user functions, the full test suite for the user functions, and the

[2] Correctness of modules exercised by those tests for the tested inputs implies correctness of those modules for all inputs [14].

output of BACCI. Steps 2, 3, and 4 can be performed concurrently. Each identified change is propagated to the roots of the call graph for the component, and all user functions that directly or indirectly call the changed function are identified for retesting.

There are two sub-steps for **the first step** of the BACCI process: (1a) decomposing[3] the binary files of the component; and (1b) filtering trivial information to facilitate comparisons by differencing tools. Prior to being distributed, component source code is compiled into files in binary code formats, such as .lib, .dll, .ocx, or .class files. Information on the data structure, functions, and function call relationships of the source code is stored in the binary files according to pre-defined formats, such as Common Object File Format (COFF)[4] [24], so that an external system is able to find and call the functions in the corresponding code sections.

The output of the first sub-step should be formatted conveniently for differencing tools to identify changes in functions between releases. The output of the second sub-step should be formatted conveniently for a graph generation tool to build call graphs. Often the first sub-step can be accomplished by parsing tools available for the language/architecture. For example, 32-bit COFF binary files can be examined by the Microsoft COFF Binary File Dumper (DUMPBIN). The DUMPBIN output is suitable for use as input to differencing tools. The second sub-step is frequently necessary because the output from the first sub-step may contain trivial information such as timestamps and file pointers, which are "noise" for the change identification. Generally, the second sub-step cannot be done via existing tools. Therefore, we have created the Decomposer and Trivial Information Zapper (D-TIZ)[5] to perform the decomposition and remove trivial information. Currently D-TIZ can only be used with library (.lib) files, but it will be extended to handle all the component types, as will be discussed in future work.

The second step of the I-BACCI process is to compare code sections between two versions. In I-BACCI Version 1, the output of D-TIZ was fed into the commercial differencing tool Araxis Merge[6] to generate reports showing the changed functions. However, a large number of false positives were observed in I-BACCI Version 1, which increased the number of functions in the application that were identified for retesting. This could result in functions in the application being re-tested unnecessarily. Source code of the component was examined to determine the cause of the false positives. We found that a large amount of false positives were caused by the changes in registers used and addresses of variables/functions. Therefore, we have created TID-BITZ to compare the code sections considering real code changes only. The algorithm used in TID-BITZ defines false positive patterns and ignores some changes of registers and addresses in the binary code sections. The algorithm can reduce many false positives but might miss real changed functions, i.e. introduce false negatives. The algorithm has evolved based upon examining the component source code and balancing false positives and false negatives.

The third and fourth steps of the I-BACCI process produce function call graphs. The main difference between the two steps is that the input for Step 3 is the calling relationships among functions in a component, and the input for Step 4 is the user functions. Generally, the call graphs generated from Step 4 are less complex than those from Step 3. In Step 4 only the

---

[3] We use *decomposition* to refer to breaking up the binary code down into constituent elements, such as code sections and relocation tables.
[4] MSDN Library - Visual Studio .NET 2003
[5] http://www4.ncsu.edu/~jzheng4/D-TIZ/index.htm
[6] http://www.araxis.com/

exposed component functions and the user functions calling them need to be included in the call graphs. The call graph can be either represented by a data structure or drawn using graph generation tools such as GraphViz[7]. Currently the call graphs are drawn and analyzed manually, according to the calling relationships in the output of DUMPBIN. For convenience in identifying affected user functions, the call graphs generated from the two steps can be integrated.

In the fifth step, the affected user functions are identified using directed graph theory algorithms. This change identification is currently done manually due to the lack of existing tool support. Analysis starts from each component function identified as changed, and that change is propagated along the call graphs from Step 4 until the user functions are reached. These are the user functions which are potentially affected by the initial changed function(s) in the component, and therefore need to be re-tested.

The method discussed in the prior paragraph is especially suitable when there are only a few function changes in the new version of the component, but many user functions that directly call these functions. An alternative method can be used when there are only a few user functions and but many component function changes that directly call component functions. Analysis may start from the user functions and examine the component functions being called by them along the call graphs, until a changed component function is found or all the leaves are reached but no changed component functions are found. In the former situation, the initial user function is affected by the change in the component, so that it needs to be re-tested; the latter situation indicates the initial user function does not need to be tested. The output of Step 5 is a list of all affected user functions which need to be re-tested.

In **the sixth step**, the set of test cases which are mapped to the user functions they cover are used to select test cases that cover only the affected user functions, as identified by the steps above. The I-BACCI process has the potential to reduce the set of regression test cases because it focuses on the affected user functions and ignores the unaffected areas in the user functions.

4. Limitations of I-BACCI

For use of I-BACCI, the licensing agreement of the COTS component must not preclude the analysis of the binary files. I-BACCI shares an acknowledged limitation with all existing source-based firewall methods: the potential for reporting false negatives in situations where binary differences are due to factors other than changes in source code (e.g. build tools, environment, or target platform). Although I-BACCI does work with the binary files for the component, and such differences are potentially detectable from binary file comparisons, the current method of analysis precludes identification of such differences.

The second limitation of I-BACCI is its potential for identifying false positives by assuming, in tracing the call graphs, that any uses of called functions with changed binaries will be affected by the change. However, an actual use of a changed function might never exercise the changed logic or data. With further development of I-BACCI, these unneeded tests may be eliminated from the regression suite. However, this limitation does not impact the current safeness of I-BACCI.

Finally, I-BACCI requires (as input) test suites which are traceable to the user functions they cover, in order to perform RTS.

---

[7] an open source tool , http://www.graphviz.org/

5. Case study

An initial case study had been conducted on a 757 thousand lines of code (KLOC) ABB application written in C/C++, using a 67 KLOC internal ABB software component in library (.lib) files written in C. The result of the case study indicates that this process can reduce the required regression tests by 40% on average [23]. Some releases required no retesting, as no changes in the component affected the product using the component. Other releases appeared to require full retesting; however, TID-BITZ had not yet been created to reduce false positives.

A second I-BACCI case study was conducted on a 40 KLOC ABB application written in C/C++. This product uses a 30 KLOC internal ABB software component in library (.lib) files written in C. Each component release contains eight libraries with total size of 5.1 ~ 5.8 megabyte. This software combination was chosen because (1) the numbers of test cases for each function of the application were available; and (2) multiple releases of the component were available. The full, retest-all strategy takes over four man months of effort to run. Five incremental releases of the component were analyzed and compared to study the effectiveness of the I-BACCI process at reducing regression test cases.

The analyzer (the first author of this paper) first applied I-BACCI Version 1 to compare Releases 1 and 2, and verify the change identification result using source code for the component for the releases. However, approximately 46% of the identified changes were false positives. To explore the cause of the false positives, the analyzer examined source code of the component, and the associated binary library files. The TID-BITZ tool was created to reduce the false positives and I-BACCI was promoted to Version 2 to include the use of this tool in Step 2. The analyzer then applied I-BACCI Version 2 on all four comparisons. The application source code was also used to analyze uses of the component functions and to draw call graphs for the interfaces between the user functions and the component. The results of the identified changes for all four comparisons and all call graphs for the components were verified by the analyzer, using source code for the component to determine the accuracy of the analysis post hoc. The reduction of test cases for each comparison was determined by the second author.

5.1 I-BACCI Version 2 processing of library files

The library files analyzed in the case study contain the raw binary code of many object files. They are organized in segments similar to the COFF file format [24]. The calling relationship among functions in the whole component can be ascertained by tracing the calls in the relocation tables throughout the library file.

During the first step of the I-BACCI process, each library file in each of the five releases was translated into plain text using DUMPBIN. D-TIZ was used to scan the output of DUMPBIN, save the code sections of functions into separate files, and obtain the relocation table of each function. For the second step, the TID-BITZ tool was used to compare the functions among the five releases and to generate differencing reports. This change identification part of the case study was conducted on an IBM T42 laptop with one Intel® Pentium® M 1.8GHz processor and one gigabyte RAM. Completing the first step of the process with DUMPBIN and D-TIZ took 18 seconds in total. TIDBITZ then spent about one second on each comparison and generating a simple differencing report.

In the third, fourth, and fifth steps, call graphs were drawn for changed functions to identify the affected functions in the source code of the application by tracing the affected component

functions along the call graphs. Manually completing the three steps for the five versions compared took the analyzer approximately 24 person hours. Then, the second author received the list of all the affected functions in the application, verified the correctness of the change identification, and produced the numbers and percent reduction of the regression test cases needed, based on the original test suite. Future automation can reduce the time required for many of these manual steps.

5.2 Results

In this new case study, the proposed I-BACCI process was applied four times between five successively-released versions of the internal ABB component. The results are shown in Table 2.

Table 2: Case Study Results with I-BACCI Version 2

Metrics	Comparisons			
	1 vs. 2	**2 vs. 3**	**3 vs. 4**	**4 vs. 5**
Total changed functions identified	388	1238	4	13
True positive ratio[8]	99.46%	98.39%	100%	100%
False positive ratio[9]	4.90%	6.14%	0%	7.69%
Affected exported component functions	84	122	1	8
% of reduced affected exported component functions	31.71%	0%	99.18%	93.44%
Affected user functions in the application	38	59	1	6
Percentage of reduced affected user functions	17.39%	0%	98.31%	89.33%
Total test cases needed	151	215	11	20
Percentage of reduced test cases	30%	0%	95%	91%

The interfaces between the application's user functions and the internal component were examined to establish a baseline of affected functions in the application. The user functions changed in Release 3. For the former version of user functions (i.e. in Release 2), there are 123 exported functions in the component. In total, 46 user functions (in six C files) call 81 out of the 123 exported functions of the component. In the worst case, all of the 46 functions would be affected by the changes in the component and would need to be re-tested. Similarly, at most 59 user functions in the latter version would be affected.

The first analysis was conducted between Release 1 and Release 2 of the component. The BACCI analysis showed that 388 functions were changed out of 1143 functions in Release 2. A source code difference analysis was performed which showed that 99.46% of the real changed functions in the source code were correctly identified and only two real changes were missed. Also, only 4.90% of the changes identified were not really changes. Firewall analysis showed

[8] True positives ratio is number of real changed functions found divided by total number of real changed functions.
[9] False positive ratio is number of identified changed functions that are not really changes, divided by number of (correctly and incorrectly) identified changed functions.

that 84 exported functions in the component were affected by the identified changes and 38 user functions were affected. As a result, 30% regression test cases can be reduced.

The second analysis comparing Release 2 and Release 3 determined that more than one thousand functions in the library were changed. The large number of changed functions would lead to a large amount of affected exported functions in the component. Therefore, the analyzer checked the user functions that called component functions. Unfortunately, all of these component function calls were identified as affected. Similar to the first analysis, approximately 6.14% of the component functions that were marked as "changed" by TID-BITZ were false positives.

The third analysis identified only a few changes between Release 3 and Release 4. Four changes were correctly identified, and only one exported function in the component was affected by the identified change. One function in the application calls the only affected functions in the component. Therefore, we achieved 95% regression test case reduction. Similarly, the analysis between Release 4 and Release 5 showed that only 91% test cases need to be re-run, although the source code difference analysis showed that one of the changes identified was a false positive. I-BACCI can be very effective when there are small incremental changes between revisions. The TID-BITZ tool was able to reduce false positives to only 6% on average while still having a low false negative rate (about 1%).

6. Conclusions and future work

In this paper, we proposed the I-BACCI process for regression test selection for user/glue code that uses software components for which source code is not available. A medium-scale product and several versions of a library component used in that product were examined as a case study to verify the potential efficacy of this process. The results showed that a reduction in test cases can be effectively determined from a component using I-BACCI when no source code is available for analysis. I-BACCI can be very effective when there are small incremental changes between revisions. In the previous case study, there were times when releases required no retesting, as no changes in the component affected the product using the component.

We plan to pursue several directions in our future work. First, we will conduct black-box testing for all tests to verify the results of our analysis, which will help determine the effectiveness of the RTS. Second, additional breadth is required to expand this process to adapt to all of the COTS file types. We plan to analyze more components in the various formats which can be examined by DUMPBIN, such as dynamic link libraries and executable files, as well as different component types such as the container/control model, in which user programs act as containers for third party controls. Third, a tool will be developed to generate the call graphs and automate the identification of affected functions, and the whole process should be automated into one tool to save both time and resources. Finally, in order to address the limitations of I-BACCI which are discussed in Section 4, we will explore other approaches, such as applying other code-based regression selection techniques, dynamic analysis techniques, and examining other available artifacts of COTS components.

Acknowledgments

This research was supported by a research grant from ABB Corporate Research. Additionally, we would like to thank Tao Xie and the NCSU Software Engineering Realsearch Reading Group for their helpful suggestions.

Bibliography

[1] Abdullah, K., Kimble, J., and White, L., "Correcting for Unreliable Regression Integration Testing," International Conference on Software Maintenance, Nice, France, 1995, pp. 232-241.

[2] Basili, V. R. and Boehm, B., "COTS-Based systems Top 10 List," *IEEE Computer*, 24(5), 2001, pp. 91-93.

[3] Bible, J., Rothermel, G., and Rosenblum, D., "A Comparative Study of Coarse- and Fine-Grained Safe Regression Test-Selection Techniques," *ACM Transactions on Software Engineering and Methodology*, 10(2), 2001, pp. 149-183.

[4] Gao, J. and Wu, Y., "Testing Component-Based Software - Issues, Challenges, and Solutions," in *3rd International Conference on COTS-Based Software Systems*. Redondo Beach, Jan. 2004, pp.

[5] Graves, T. L., Harrold, M. J., Kim, Y. M., Porter, A., and Rothermel, G., "An Empirical Study of Regression Test Selection Techniques," *ACM Trans. on Software Engineering and Methodology*, 10(2), 2001, pp.184-208.

[6] Harrold, M. J., Orso, A., Rosenblum, D., Rothermel, G., Soffa, M. L., and Do, H., "Using Component Metacontents to Support the Regression Testing of Component-Based Software," IEEE International Conference on Software Maintenance, Florence, Italy, 2001, pp. 716-725.

[7] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Standard 610.12*, 1990.

[8] Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., and Chen, C., "Class Firewall, Test Order and Regression Testing of Object-Oriented Programs," *Journal of Object-Oriented Programming*, 8(2), 1995, pp. 51-65.

[9] Laski, J. and Szermer, W., "Identification of program modifications and its applications in software maintenance," International Conference on Software Maintenance, 1992, pp. 282-290.

[10] Leung, H. and White, L., "A Study of Integration Testing and Software Regression at the Integration Level," International Conference on Software Maintenance, San Diego, 1990, pp. 290-301.

[11] Leung, H. and White, L., "Insights into Testing and Regression Testing Global Variables," *Journal of Software Maintenance*, Vol. 2, No. 4, Dec. 1991, pp. 209-222.

[12] Mayrhauser, A. v., Mraz, R. T., and Walls, J., "Domain Based Regression Testing," International Conference on Software Maintenance, Sept. 1994, pp. 26-35.

[13] Orso, A., Apiwattanapong, R., Law, J., Rothermel, G., and Harrold, M. J., "An empirical comparison of dynamic impact analysis algorithms," International Conference on Software Engineering, Edinburgh, 2004, pp. 491-500.

[14] Rothermel, G. and Harrold, M., "Analyzing regression test selection techniques," *IEEE Trans. on Software Engineering*, 22(8), 1996, pp. 529-551.

[15] Srivastava, A., "Vulcan," TR-99-76, Microsoft Research Sept. 1999.

[16] Srivastava, A. and Thiagarajan, J., "Effectively prioritizing tests in development environment," ACM International Symposium on Software Testing and Analysis, Roma, Italy, 2002, pp. 97-106.

[17] Vokolos, F. and Frankl, P., "Empirical evaluation of the textual differencing regression testing technique," International Conference on Software Maintenance, 1998, pp. 44-53.

[18] Wang, Z., Pierce, K., and McFarling, S., "BMAT: A Binary Matching Tool for Stale Profile Propagation," *The Journal of Instruction-Level Parallelism*, Vol. 2, May 2000.

[19] White, L. and Abdullah, K., "A Firewall Approach for the Regression Testing of Object-Oriented Software," in *Software Quality Week*. San Francisco, 1997.

[20] White, L., Almezen, H., and Sastry, S., "Firewall Regression Testing of GUI Sequences and Their Interactions," International Conference on Software Maintenance, Amsterdam, The Netherlands, 2003, pp.398-409.

[21] White, L. and Leung, H., "A Firewall Concept for both Control-Flow and Data Flow in Regression Integration Testing," International Conference on Software Maintenance, Orlando, 1992, pp. 262-271.

[22] White, L. and Robinson, B., "Industrial Real-Time Regression Testing and Analysis Using Firewall," International Conference on Software Maintenance, Chicago, Sept. 2004, pp. 18-27.

[23] Zheng, J., Robinson, B., Williams, L., and Smiley, K., "An Initial Study of a Lightweight Process for Change Identification and Regression Test Selection When Source Code is Not Available," TR-2005-38, North Carolina State University 2005.

[24] Zheng, J., Robinson, B., Williams, L., and Smiley, K., "A Process for Identifying Changes When Source Code is Not Available," the 2nd International Workshop on Models and Processes for the Evaluation of off-the-shelf Components (MPEC '05), St. Louis, MO, May, 2005.