On Regression Testing COTS-based Applications

Jiang Zheng
Department of Computer Science, North Carolina State University, Raleigh, NC 27695
jzheng4@ncsu.edu

1. Problem and Motivation

Companies increasingly incorporate a variety of commercial-off-the-shelf (COTS) components in their products. Upon receiving a new release of a COTS component, users of the component often conduct regression testing to determine if a new version of a component will cause problems with their existing software and/or hardware system. 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]. A variety of regression test selection (RTS) techniques have been developed (for example, [2, 6, 12]) to reduce the number of tests that need to be executed without significant risk of excluding important failure-revealing test cases. However, most existing RTS techniques rely on source code, and therefore are not suitable when source code is not available for analysis, such as when an application incorporates COTS components.

Due to the lack of information, the most straightforward RTS technique for COTS-based applications would be to rerun all of the test cases for the application involving the glue code after the new COTS component(s) have been integrated. *Glue code* is application code that interfaces with the COTS components, integrating the component with the application. The retest-all strategy is straightforward but can be prohibitively expensive in both time and resources [6]. The research objective is to *evolve and validate an efficient RTS technique and supporting tools for COTS-based applications. The RTS technique and tools will reduce the test suite required to evaluate changed COTS components when source code is not available.*

2. Background and Related Work

This section provides prior work in testing of software component, static binary code analysis, and firewall analysis.

2.1 Testing of Software Components

Poor testability, due to the lack of access to the component's source code and other artifacts, is one of the challenges in user-oriented component testing [4, 5, 14]. Generally, black-box tests are run on COTS software because users do not have access to the source code to analyze the internal implementation. *Black-box testing*, also called *functional testing* or *behavioral testing*, is testing that ignores the internal mechanisms of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions [7]. Black-box test cases of COTS component functionality 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. When only binary code is available, binary reverse engineering is a technically-feasible approach for automatically deriving information that can inform the RTS. The derived information can be a design structure of a component from its binary code, such as, call graphs [11].

2.2 Static Binary Code Analysis

Binary code analysis (BCA) approaches and tools have been developed and utilized in many software development-related activities, including program comprehension, software maintenance and software security, even by software developers that have access to the source code. BCA can be dynamic or static. Dynamic BCA monitors the execution of programs. In contrast, static BCA provides a way to obtain information about the possible states that a program reaches execution without actually running the program on specific inputs. Static techniques explore the program's behavior for all possible inputs and all possible states that the program can reach. [1]

Srivastava and Thiagarajan at Microsoft developed Echelon [13], a test prioritization system. Echelon is used to prioritize tests based upon changes between two versions identified by a static binary code comparison. Echelon takes as input two versions of a program in binary form and a mapping between the test suite and the lines of code the test suite executes. Echelon outputs a prioritized list of test sequences (small groups of tests) [13]. Srivastava and Thiagarajan also discussed the advantages of comparing software at the binary level rather than the source code level: (1) easier to integrate into the build process because the recompilation step needed to collect coverage data is eliminated; and (2) all the changes in header files (such as constants and macro definitions) have been propagated to the affected procedures, simplifying the determination of program changes [13]. However, Echelon is a large proprietary Microsoft internal product with a significant infrastructure and an underlying bytecode manipulation engine, and therefore cannot be used by the community at large. Also, Echelon prioritizes, but does not eliminate tests [13]. Our goal is to provide information about which test cases are not necessary to rerun.

2.3 Firewall Analysis

Leung and White [10, 15] 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 restricts regression testing to potentially-affected system elements directly dependent upon changed system elements [15, 16]. Affected system elements include modified functions and data structures, and their calling functions. 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 [15]. Via empirical studies of industrial real-time systems, firewall analysis was shown to be effective [16]. The firewall analysis allowed an average savings of 36% of the testing time and 42% of the tests run. No additional errors were detected by the customer on the studied software releases that

were due to the changes in these releases to date. [16] *Our approach extends the traditional concept and scope of firewall analysis for use with binary code.*

3. Approach and Uniqueness

We have evolved a process with supporting tools for RTS for COTS-based applications. The process is called *the Integrated - Black-box Approach for Component Change Identification (I-BACCI)* [18]. The I-BACCI process is an integration of (1) a static binary code change identification process; and (2) firewall analysis RTS technique. *Our uniqueness is the combination of these two parts to identify and localize change with the goal of reducing the regression test suite.*

The I-BACCI process and supporting tools have been evolved to Version 4 through the application of the process on both library (LIB) and Dynamic Link Library (DLL) components written in C/C++. The I-BACCI Version 4 involves seven steps, as shown in Figure 1. The first four steps are completed via a BCA process (in dash-dotted line frame), which identifies semantic changes within binary code and produces a report on affected exported component functions. *Affected exported component functions* are functions within the COTS component that interface with the application, and are either changed or affected by other changed functions. The remaining three RTS steps are completed via firewall analysis (in dashed line frame). The input artifacts to the process are the binary code of the COTS components (old and new versions); the source code and test suite of the development application; and all test cases which are mapped to the glue code functions they cover. These input artifacts are generally available to users of COTS components. The output of the I-BACCI process is a reduced suite of regression test cases necessary to exercise the changed areas in the new COTS components.

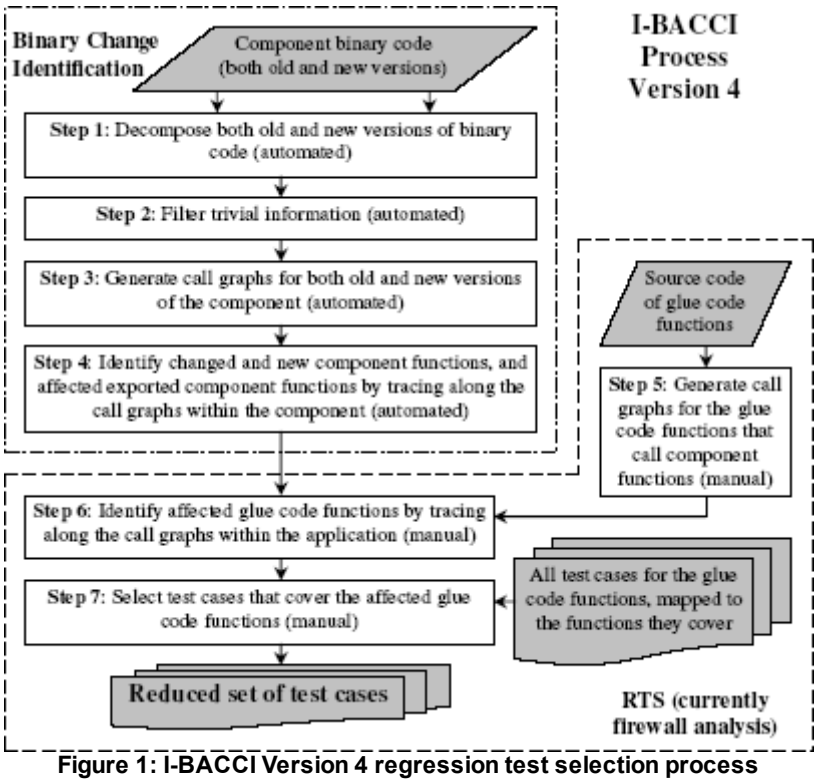

Figure 1: I-BACCI Version 4 regression test selection process

3.1 The Process

The first step of the I-BACCI process is to decompose the binary files of the component. We use the term *decomposing* to refer to breaking up the binary code down into constituent elements, such as code sections and relocation tables. Prior to distribution, component source code is compiled into binary code, such as `.lib` or `.dll` files. Information on the data structures, functions, and function calling relationships of the source code is stored in the binary files according to pre-defined formats, such as Common Object File Format (COFF) and Portable Executable (PE) format, so that an external system is able to find and call the functions in the corresponding code sections. Often the first step can be accomplished by parsing tools available for the language/architecture. For example, COFF and PE binary files can be examined by the Microsoft COFF Binary File Dumper (DUMPBIN).

The second step, filtering trivial information, is frequently necessary because the output from the first step may contain information such as timestamps and file pointers, which are irrelevant to the change identification. Generally, the second step cannot be completed via existing tools. The *Decomposer and Trivial Information Zapper (D-TIZ)* tool was created to perform the decomposition and remove trivial information. Detailed information on the supporting tools will be discussed in Section 3.2. The output of the second step is the raw code section of each function/data, and function/data calling relationships for the new version of the component.

The third step of the I-BACCI process is to generate call graphs for both old and new versions of the component. We created *Call-graph Analyzer - Affected Function Identifier (CAAFI)* applications to represent and analyze the call graphs of components of both LIB and DLL types automatically in the I-BACCI Version 4, as will be presented in detail in Section 3.2.

In **the fourth step**, we identify changed and new component functions, and then identify affected exported component functions by tracing along the call graphs within the component using directed graph theory algorithms. Analysis starts from each component function identified as changed, and that change is propagated along the call graphs from the third step until the exported functions are reached. The output of the fourth step is a list of all affected exported component functions. The *Trivial Identifier of Differences in BInary-analysis Text Zapper (TID-BITZ)* tool was created to remove most of the false positives caused by trivial differences such as shifted addresses and register reallocations, such that true positive changes can be identified in the raw binary code of functions and data. CAAFI is able to identify the affected exported component functions.

Using the source code of glue code functions, **the fifth step** is to generate function call graphs for glue code functions that call exported component functions. The call graphs generated from the third step and the fifth step can be integrated to learn how glue code functions are affected by changed and new component functions. The call graphs can be drawn using existing open source tools such as GraphViz.

Similar to the fourth step, the affected glue code functions are identified in **the sixth step**. *Affected glue code functions* are functions within the glue code that directly call affected exported component functions and therefore need to be re-tested.

In **the seventh step**, the set of test cases which are mapped to the glue code functions they cover are used to select only test cases that cover the affected glue code 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 glue code functions and ignores the unaffected areas in the application. The process is illustrated in Figure 2. Exported component functions E2 and E4 are affected by changed and new component functions which are shown as black circles. Functions in the real line box (G2 and G4) are the affected glue code functions, which need to be re-tested. Although also affected by the component change, glue code function G3 does not need to be re-tested because G2 has been re-tested, according to the firewall analysis concept.

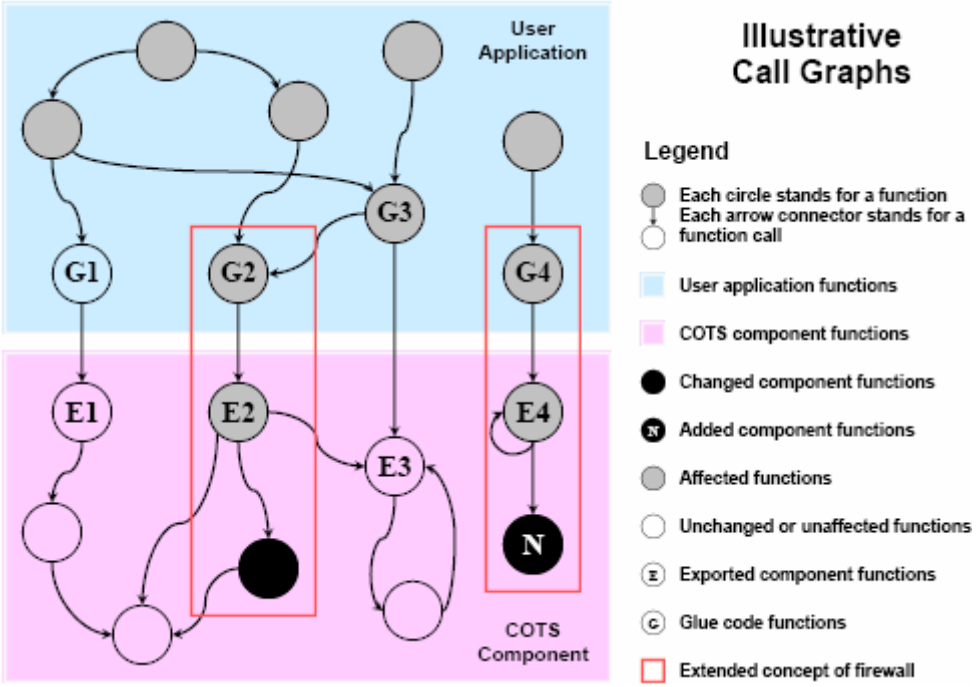

Figure 2: Illustration of I-BACCI process

3.2 Supporting Tools

Due to the different file formats for LIB and DLL components, the tools that were developed for components of LIB and DLL types are described respectively. For LIB components, D-TIZ_LIB was created to scan the output of DUMPBIN, save the code sections of functions into separate files, and collect and save the relocation tables of the functions into a text file (henceforth called "*relocation table set*"). However, a large number of false positives were observed in the initial case study of the I-BACCI Version 1 [17], which increased the number of glue code functions that were identified for retesting. To explore the cause of the false positives, I examined the source code and the associated binary library files of the component. A large amount of false positives were caused by changes in registers used and addresses of variables and functions, which typically would not cause functional changes in the code. TID-BITZ_LIB was created to reduce the number of false positive changes identified due to trivial changes, such as shifted addresses and register reallocations. CAAFI_LIB was created represent and analyze the function call graphs within components. The relocation table set of a component is converted into an adjacency-matrix [3] to represent call graphs of the functions in the component. For each changed function, CAAFI_LIB then backtracks the call graphs to identify all functions that directly or indirectly call the

changed function. For each case study, CAAFI_LIB reduced the analysis time cost from approximately 16 hours to less than three minutes.

Unlike `.lib` files, which are in the COFF format, a DLL binary is in the PE file format. Some characteristics of the PE format make the change identification and call-graph generation more complex and difficult. For example, only the names of exported component functions can be obtained in the binary code, such that functions have to be mapped between two releases after generating the call graphs for exported component functions. Similar to D-TIZ_LIB, D-TIZ_DLL decomposes the DLL binary flies and removes trivial information. The algorithm examines the DLL binaries from coarse to fine granularity step by step. First, the tool invokes DUMPBIN Version 8 to translate the illegible binary library files into readable plain text files. Then a file reader automatically scans the DUMPBIN output and loads useful information, such as instructive information in file header, section headers, export table and import table, into a predefined data structure which is constructed according to the PE file format specification. The next finer granularity is in function/data-level. Binary code of functions and data are stored consecutively in .text section and data sections (*.rdata*, *.data*, *.idata*, etc.), respectively. However, only names of exported component functions are available. Other functions and all data have to be labeled by their *start virtual addresses* (SVA). We abstract function and data as the same class (`DLLFunctionData`) with the following attributes: SVA, *end virtual address* (EVA), raw binary code, and relocation list. The relocation table is read from the .reloc section and then converted into a Hashtable called relocation index. For each key-value pair in the relocation index, the key is a *calling virtual address* (CVA) where the control flow jumps to another function or data, and the SVA of the function or data being called (a.k.a. *target virtual address* (TVA)) is stored as the value. Function CVA and TVA can also be calculated according to the position of each call instruction and the address offset following each call instruction, respectively. Because only binary code is available instead of assembly code, the tool searches opcode `E8` and `E9` which represent "call near" in the Intel instruction set to locate the position of each function call. After finding all functions and data SVAs, an array in `DLLFunctionData` type is constructed and the raw code of the *.text* and data sections is decomposed into separate functions or data.

The function/data call-graphs and full code representation for all exported component functions can be generated recursively following the calling track by CAAFI_DLL. A few steps that remove trivial bytes are also conducted during processing of this level. For example, most raw code of functions/data is followed by a few useless bytes (e.g. `90`, `CC`) for the purpose of alignment.

Further instruction-level comparisons can be conducted by TID-BITZ_DLL after the function/data level if the full code representations of an exported component function in two releases are still different. For example, false positives may be caused by register allocation changes from build to build. After all of the above steps, a report on differencing of exported component functions will be generated. We can use this report to identify affected application code and then select proper regression test cases.

3.3 Limitations of the I-BACCI Process

The I-BACCI process shares an acknowledged technical limitation with all existing firewall methods: the potential for reporting false positives and false negatives when binary differences are due to factors other than changes in source code (e.g. build tools, environment, or target platform). Although such differences are potentially detectable from the binary file comparisons used in the I-BACCI process, the current method of analysis precludes identification of such differences. The second limitation of the I-BACCI process is its potential for identifying false positives by conservatively 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 the I-BACCI process, these unneeded tests may be eliminated from the regression suite. Finally, the I-BACCI process requires (as input) test suites which are traceable to the glue code functions they cover, in order to perform RTS.

4. Results and Contributions

Applications and components used in the case studies are internal software products written in C and C++ provided by ABB Inc. Two case studies of applying the I-BACCI process were completed on two applications and their LIB components and one case study was conducted on a DLL component written in C and its application. For each component, four to six incremental releases were analyzed and compared. These software combinations were chosen for these case studies because (1) the numbers of test cases for each function of the applications were available; (2) multiple releases of the components were available; (3) the high cost of executing the retest-all strategy demonstrates the potential value of achieving regression test reductions. The retest-all strategy takes over four person months of effort. With the help of the tools, the I-BACCI process took approximately two person hours for each case study to determine the reduced test suite.

The results of the case studies, as shown in Tables 1 and 2, indicate that I-BACCI is an effective RTS technique for COTS-based applications. In the best case, the I-BACCI process can reduce the required number of regression tests by as much as 100% if our analysis indicates the changes to the COTS component are not called by the glue code. This fact would not be known to the users of COTS component without I-BACCI analysis, such that they would still be tempted to do retest-all. As with all RTS strategies, the I-BACCI process is most effective when there are small incremental changes between revisions. When there are a large number of changes in the new component, I-BACCI suggests a retest-all regression testing strategy. The results have been verified by examining the failure records of retest-all black-box testing. In our case studies, current tools identified all changes; no failures escaped the reduced test suites. However, current tools reported false positives when two versions were not built by the same compiler or linker.

Table 1: Results of LIB Cases by the I-BACCI Version 4

Metrics	Case 1					Case 2			
	1 vs 2	2 vs 3	3 vs 4	4 vs 5	5 vs 6	1 vs 2	2 vs 3	3 vs 4	4 vs 5
Total changed functions identified	18	23	1	10	3	338	1238	4	13
True positive ratio	100%	100%	100%	100%	100%	99.5%	98.4%	100%	100%
False positive ratio	5.6%	0%	0%	0%	0%	4.9%	6.1%	0%	7.7%
Affected exported component functions	319	71	2	55	39	84	122	1	8
% of affected exported component functions	96.4%	21.5%	0.6%	16.6%	11.8%	68.3%	100%	0.8%	6.6%
Affected glue code functions	60	2	0	0	0	38	59	1	6
% of affected glue code functions	100%	3.3%	0%	0%	0%	82.6%	100%	1.7%	10.7%
Total test cases needed	592	8	0	0	0	151	215	11	20
% of test cases reduction	0%	98.7%	100%	100%	100%	30%	0%	95%	91%
Actual regression failures found	0	0	0	0	0	4	8	1	0
Regression failures detected by reduced test suite	0	0	0	0	0	4	8	1	0

Table 2: Results of DLL Case by the I-BACCI Version 4

Metrics	Comparisons			
	1 vs 2	2 vs 3	3 vs 4	4 vs 5
Total changed functions identified	338	1238	4	13
True positive ratio	99.5%	98.4%	100%	100%
False positive ratio	4.9%	6.1%	0%	7.7%
Affected exported component functions	84	122	1	8
% of affected exported component functions	68.3%	100%	0.8%	6.6%
Affected glue code functions	38	59	1	6
% of affected glue code functions	82.6%	100%	1.7%	10.7%
Total test cases needed	151	215	11	20
% of test cases reduction	30%	0%	95%	91%
Actual regression failures found	4	8	1	0
Regression failures detected by reduced test suite	4	8	1	0

Another potential contribution is that we investigated the legal limits of BCA of purchased software with the help of a professor of Software Engineering who is also a lawyer [8]. The interaction of patent law and mass market license terms, as it affects interoperability, is being actively debated within the legal profession [9]. The I-BACCI process is not used to create the COTS components. The use described here - which we expect to be the normal application of I-BACCI - is for reducing time and cost of regression testing in the face of changes in COTS components. The I-BACCI process promotes the cost-effective continuation of interoperability between the customer's product and the COTS components purchased to build the product.

In the future, extensive validation of both the tool support and RTS technique will require more industrial case studies, data collection, and further RTS analysis. Open source end-to-end automation to facilitate the efficiency and convenience of the whole process is in process.

Acknowledgments

This research was supported by a research grant from ABB Corporate Research. I would also like to thank my advisor Dr. Laurie Williams and the North Carolina State University Software Engineering Realsearch Reading Group for their helpful suggestions.

References

[1] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum, "WYSINWYX: What You See Is Not What You eXecute," in *The IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, 2005.

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

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 2nd Edition ed. Cambridge , Massachusetts London, England: The MIT Press and McGraw-Hill, 2001.

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

[5] J. Z. Gao, H.-S. J. Tsao, and Y. Wu, *Testing and Quality Assurance for Component-Based Software*. Boston: Artech House, 2003.

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

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

[8] C. Kaner, J. Zheng, L. Williams, B. Robinson, and K. Smiley, "Binary Code Analysis of Purchased Software: What are the Legal Limits?," Submitted to *the Communications of the ACM*, 2007.

[9] D. Laster, "The Secret Is Out: Patent Law Preempts Mass Market License Terms Barring Reverse Engineering for Interoperability Purposes."

[10] H. Leung and L. White, "A Study of Integration Testing and Software Regression at the Integration Level," in International Conference on Software Maintenance, San Diego, 1990, pp. 290-301.
[11] A. M. Memon, "A process and role-based taxonomy of techniques to make testable COTS components," in *Testing Commercial-off-the-shelf Components and Systems*, S. Beydeda and V. Gruhn, Eds. Berlin, Germany: Springer-Verlag, 2005, pp. 109-140.
[12] G. Rothermel and M. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22(8), pp. 529-551, 1996.
[13] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in ACM SIGSOFT International Symposium on Software Testing and Analysis, Roma, Italy, 2002, pp. 97-106.
[14] E. J. Weyuker, "Testing Component-Based Software: A Cautionary Tale," *IEEE Software*, vol. 15(5), pp. 54-59, 1998.
[15] L. White and H. Leung, "A Firewall Concept for both Control-Flow and Data Flow in Regression Integration Testing," in International Conference on Software Maintenance, Orlando, 1992, pp. 262-271.
[16] L. White and B. Robinson, "Industrial Real-Time Regression Testing and Analysis Using Firewall," in International Conference on Software Maintenance, Chicago, 2004, pp. 18-27.
[17] J. Zheng, B. Robinson, L. Williams, and K. Smiley, "An Initial Study of a Lightweight Process for Change Identification and Regression Test Selection When Source Code is Not Available," in 16th IEEE International Symposium on Software Reliability Engineering, Chicago, IL, USA, 2005, pp. 225-234.
[18] J. Zheng, B. Robinson, L. Williams, and K. Smiley, "Applying Regression Test Selection for COTS-based Applications," in 28th IEEE International Conference on Software Engineering (ICSE'06), Shanghai, P. R. China, May 2006, pp. 512-521.