Applying Regression Test Selection for COTS-based Applications

Jiang Zheng[1], Brian Robinson[2], Laurie Williams[1], Karen Smiley[2]
[1] Department of Computer Science, North Carolina State University, Raleigh, NC, 27695
{jzheng4, lawilli3}@ncsu.edu
[2] ABB Inc., US Corporate Research
{brian.p.robinson, karen.smiley}@us.abb.com

ABSTRACT
ABB incorporates a variety of commercial-off-the-shelf (COTS) components in its products. When new releases of these components are made available for integration and testing, source code is often not provided. Various regression test selection processes have been developed and have been shown to be cost effectiveness. However, the majority of these test selection techniques rely on access to source code for change identification. In this paper we present the application of the lightweight Integrated - Black-box Approach for Component Change Identification (I-BACCI) Version 3 process that select regression tests for applications that use COTS components. Two case studies, examining a total of nine new component releases, were conducted at ABB on products written in C/C++ to determine the effectiveness of I-BACCI. The results of the case studies indicate this process can reduce the required number of regression tests at least 70% without sacrificing the regression fault exposure.

Categories and Subject Descriptors
D.2.4 [**Software/Program Verification**]

General Terms: Reliability, Verification.

Keywords: software testing, regression testing, commercial-off-the-shelf, COTS

1. INTRODUCTION
ABB[1], a global power and automation technologies company, incorporates a variety of commercial-off-the-shelf (COTS) components in its products. Upon receiving a new release of a COTS component, users such as ABB 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, but only to the binary files and reference documents.

ICSE'06, May 20-28, 2006, Shanghai, China.

[1] http://www.abb.com/

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 [10]. To minimize the time and resource costs of regression testing, a variety of regression test selection (RTS) processes have been developed [4, 8, 20]. However, most of these processes rely on source code, and therefore, are not suitable when source code is not available for analysis, such as when an application uses COTS components.

As a result, the default RTS strategy would be to retest all the functions involving the glue code[2] due to the lack of information. The retest-all strategy is straightforward but can be prohibitively expensive in both time and resources [8]. North Carolina State University and ABB are collaborating with a common research goal: *to safely reduce the testing required when components change and only binary code and documentation is available.*

The theory we are building via our research is as follows:

When components change and only binary code and documentation are available, regression test selection can safely be based upon the glue code that interfaces with sections of the component that changed.

To this end, we have evolved a six-step process with supporting tools for RTS for COTS-based applications [32, 33]. We call our process the Integrated - Black-box Approach for Component Change Identification (I-BACCI) process. The input artifacts to the process are the binary code of the components (old and new versions) and the source code and test suite of the application. These artifacts are generally available to the COTS user. The output of the I-BACCI process is a reduced suite of regression test cases, related to the changes in the COTS components.

This paper furthers our prior work in (1) presenting the results of two industrial case studies of using I-BACCI Version 3 to reduce the number of test cases in the regression test suite; (2) presenting detailed tool support; (3) investigating the legality of analyzing binary code of purchased COTS components; and (4) examining the product failure records for a retest-all black box testing effort that was executed on the product. 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 [10]. We use these results to analyze the safeness and effectiveness of I-BACCI as an RTS process. A *safe*

[2] *Glue code* is application code that interfaces with the COTS components, integrating the component with the application. Terms used in this paper are illustrated in Appendix A.

RTS process 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 [4, 14, 20].

The rest of this paper is organized as follows. Section 2 outlines the background and related work. Section 3 describes the I-BACCI Version 3 process and its limitations. Tool support is discussed in Section 4. Section 5 presents the two case studies of applying I-BACCI Version 3 on two ABB products and their library components. Section 6 discusses our lessons learned. Finally, Section 7 presents conclusions and future work.

2. BACKGROUND AND RELATED WORK

In this section, we discuss prior work in 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 challenges in user-oriented component testing [6, 7, 26]. The functions and behaviors of the components can be hard to understand because a third-party component user only has the access to component specification, user interfaces, and reference manual [7]. 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. When only binary code is available, binary reverse engineering can be a feasible approach to automatically derive a design structure of a component from its binary code, such as, call graphs [17].

Harrold et al. [9] presented techniques that use component metadata for regression test selection of COTS components. They illustrated their technique with a controlled example and seven releases of a real component-based system, demonstrating an average savings of 26% of the testing effort [9]. Their techniques utilize three types of metadata to perform the regression test selection: (1) the branch coverage achieved by the test suite with respect to the component to associate test cases with branches; (2) the component version; and (3) a means to query the component for the branches affected by changes in the component between two given versions [9]. However, the component provider may not provide this information. In our research, we focus on using information that is typically available to a component user.

2.2 Regression test selection

The purpose of RTS processes is to reduce the high cost of retest-all regression testing by selecting a subset of possible test cases [8]. In the selection of test cases, an RTS process might not be safe. A variety of RTS techniques [4, 8, 20] have been proposed, such as methods based upon path analysis techniques or dataflow techniques. However, these techniques rely upon source code.

Srivastava and Thiagarajan at Microsoft have developed a test prioritization system, Echelon [22], that prioritizes an application's set of tests based on a binary code comparison of two versions. Echelon takes as input two versions of the program in binary form, and 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). The researchers analyzed the efficacy of Echelon based on two runs of a comparison between two binaries of a 1.8 million line of code office productivity application [22]. In the first run, Echelon detected 87% of the defects in the first two of 148 test sequences; the remaining 13% of the defects were not detected by any tests in the test suite. In the second run of different binaries, Echelon detected 98% of the defects in the first three of 221 test sequences; the remaining 2% of the defects were not detected by any tests.

Srivastava and Thiagarajan also discussed the advantages of comparing at the binary level rather than the 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 [22]. 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 [18] 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 [2, 13, 19, 20, 23, 24]. These approaches are not suitable for component testing when source code is not available.

Laski and Szermer [13] proposed a formal method to identify modifications made in a program. Vokolos and Frankl [23, 24] utilized a textual differencing technique to perform regression test selection. Apiwattanapong et al. [2] presented a technique for comparing object-oriented programs that identifies both differences and correspondences between two versions of a program. The algorithm is based on a method-level representation that models the object-oriented features of the language. Given two programs, their algorithm identifies matching classes and methods, builds a representation for each pair of matching methods, and compares the representation to identify similarities and differences. Empirical results show the efficiency and effectiveness of the technique on a real program [2]. Ren et al. [19] developed Chianti, a change impact analysis tool for Java. Chianti analyzes two versions of a Java program, decomposes their difference into a set of atomic changes, and calculates partial order inter-dependencies of these changes. Change impact is then reported in terms of affected (regression or unit) tests whose execution behavior may have been modified by the applied changes. For each affected test, Chianti also determines a set of affecting changes that were responsible for the test's modified behavior [19].

Wang et al. [25] developed the Binary Matching Tool (BMAT) which compares two versions of a binary program without knowledge of the source code changes. The implementation uses a hashing-based algorithm and a series of heuristic methods to find correct matches for as many program blocks as possible. The algorithm first matches procedures, then basic blocks within each procedure. The implementation of BMAT is built on Windows NT® for the x86 architecture, using the Vulcan binary analysis

tool [21] to create an intermediate representation of x86 binaries, which frees the BMAT developers from the tasks of separating code from data and identifying program symbols. The process enables good matching even with shifted addresses, different register allocations, and small program modifications [25]. BMAT underlies Echelon [22] (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)[3] tool to perform the same function for I-BACCI, as will be discussed in Section 4.2.

Although a comparison between versions of documentation is potentially helpful [14, 16], the documentation for COTS components may not reflect all changes. Implementation may change without necessitating any documentation changes, such as for a code fix. Thus, users of COTS software should perform thorough change identification which does not rely solely on the component documentation. I-BACCI addresses this.

2.4 Firewall analysis

Leung and White [1, 14, 15, 29] developed firewall analysis for regression testing with integration test cases (tests that evaluate interactions among components [10]) in the presence of small changes in functionally-designed software. Firewall analysis limits regression testing to potentially-affected system elements directly dependent upon changed system elements [29, 30]. I-BACCI utilizes firewall analysis for RTS.

Module dependencies, control-flow dependencies, and data dependencies are considered in firewall analysis [29]. 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 [29]. Test cases that need to be re-run over these modules are identified and/or new test cases to exercise new code or functionality are generated. Kung et al. [11, 12] utilized the firewall concept on an object-oriented system, and White and Abdullah [27] expanded the firewall to address more features of an object-oriented system. Firewall was also utilized in the regression testing of graphical user interfaces [28].

Firewall methods can only be guaranteed to select all modification-revealing [20] tests and to be safe if all unit and integration tests initially used to test system components are reliable. Tests are *reliable* if the correctness of modules exercised by those tests for the tested inputs implies correctness of those modules for all inputs [20]. However, test suites are typically not reliable in practice [30], so the firewall technique may omit modification-revealing tests and/or may admit some non-modification-traversing tests. Via empirical studies of industrial real-time systems, firewall was effective despite these theoretical limitations [30]. These limitations thus should not impair the effectiveness of I-BACCI in practice.

3. I-BACCI VERSION 3

The I-BACCI process is an integration of the firewall analysis RTS process with our Black-box Approach for Component Change Identification (BACCI) process (initially proposed in [31]) for identifying change. I-BACCI Version 3 involves six steps, as shown in Figure 1. The inputs to the I-BACCI process are shown in gray blocks. 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 functions in the components. The remaining four steps are done via firewall analysis (in dashed line frame), which requires the glue code functions, the full test suite for the application, and the output of BACCI.

3.1 I-BACCI process

There are two sub-steps for **the first step** of the BACCI process: (1a) decomposing[4] the binary files of the component; and (1b) filtering trivial information to facilitate comparisons by differencing tools. Prior to distribution, component source code is compiled into binary code formats, such as .lib, .dll, or .class files. Information on the data structure, 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)[5] [33], 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 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)[5]. The DUMPBIN output is suitable 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)[6] to perform the decomposition and remove trivial information. D-TIZ is described more fully in Section 4.1.

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[7] 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 unnecessarily identified for retesting. 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 identify real code changes only, eliminating differences due to registers and addresses. The algorithm used in TID-BITZ will be introduced in detail in Section 4.2.

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

[4] We use the term *decomposing* to refer to breaking up the binary code down into constituent elements, such as code sections and relocation tables.

[5] MSDN Library - Visual Studio .NET 2003

[6] http://www4.ncsu.edu/~jzheng4/D-TIZ/index.htm

[7] http://www.araxis.com/

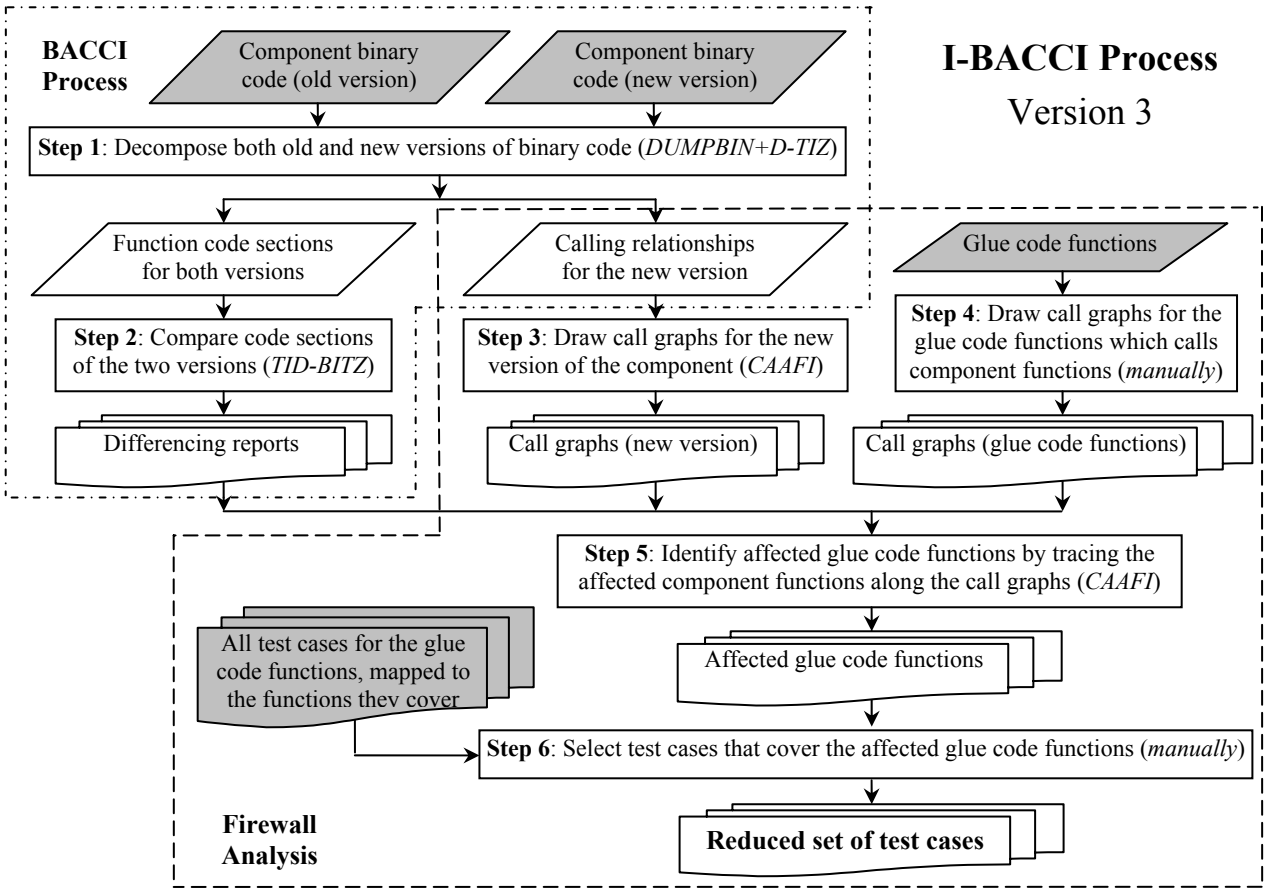


Figure 1: I-BACCI Version 3 Regression Test Selection Process

The third and fourth steps of the I-BACCI process produce function call graphs. The input for Step 3 is the calling relationships among functions in a component, and the input for Step 4 is the calling relationships for the glue code. In Step 4, only the exported component functions[8] and the glue code functions calling them need to be included in the call graphs. In the first two versions of I-BACCI, the call graphs were drawn and analyzed manually. We have created Call-graph Analyzer – Affected Function Identifier (CAAFI)[9] (as will be discussed in Section 4.3) to represent and analyze the call graphs of the components automatically in I-BACCI Version 3. The call graphs for the glue code are still drawn and analyzed manually via adjacency-matrix representation [5] because no proper existing tool was found to represent call graphs for source code. The call graphs can be drawn using graph generation tools such as GraphViz[10], and the call graphs generated from the two steps can be integrated together to identify affected glue code functions.

In **the fifth step**, the affected glue code functions are identified using directed graph theory algorithms. These are the functions within the application that are potentially affected by the changed function(s) in the component, and therefore need to be re-tested. Analysis starts from each component function identified as changed, and that change is propagated along the call graphs from Step 4 until the glue code functions are reached.

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 glue code functions that directly call these functions. An alternative method can be used when there are only a few glue code functions and but many component function changes that directly call component functions. Analysis may start from the glue code 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 glue code function is affected by the change in the component, so that it needs to be re-tested; the latter situation indicates the initial glue code function does not need to be tested. The output of Step 5 is a list of all affected glue code functions which need to be re-tested.

CAAFI identifies the affected component functions. Affected glue code functions can be identified based on the changed and/or affected component functions. The algorithm used in CAAFI will be discussed in detail in Section 4.3.

[8] Exported component functions are functions within the COTS component that interfaces with application, as illustrated in Appendix A.

[9] http://www4.ncsu.edu/~jzheng4/CAAFI/index.htm

[10] An open source tool, http://www.graphviz.org/

In **the sixth step**, the set of test cases which are mapped to the glue code functions they cover are used to select test cases that cover only 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.

3.2 Limitations of I-BACCI

I-BACCI shares an acknowledged technical limitation with all existing 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 degrade the level of safeness of the I-BACCI method below that of its underlying firewall RTS technique.

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

4. TOOL SUPPORT

In this section, three tools that were developed and used in I-BACCI Version 3 are described. More details and pseudocode for all the tools is available online[3,6,9].

4.1 D-TIZ

D-TIZ is used in the first step of the I-BACCI process with DUMPBIN. Information on the functions and structures of the source code is stored in some areas of the binary files according to pre-defined formats, so that a system is able to call the functions in corresponding code sections. DUMPBIN translates the illegible binary library files into readable plain text. An example of the output of DUMPBIN is shown in Appendix B. Function names, binary code representation of the functions, and relocation tables are all clearly described in the output text of DUMPBIN. D-TIZ scans the output of DUMPBIN, saves the code sections of functions into separate files, and collects and saves the relocation tables of the functions into a text file (henceforth called "*relocation table set*"). The function list is fed into TID-BITZ to perform differencing. CAAFI utilizes the relocation table set to generate and analyze call graphs of the components. Currently D-TIZ can only be used with library (.lib) files, but we are currently extending to handle additional component types, such as dynamic link library (DLL).

4.2 TID-BITZ

A large number of false positives were observed in the initial case study of I-BACCI Version 1 [33], which increased the number of glue code functions that were identified for retesting. To explore the cause of the false positives, the analyzer 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.

For example, as shown in bold in Appendix B, the binary code `8B89A0060000` means "copy the operand in the address of register ECX plus offset `0x06A0` to register ECX", where `8B89` is the instruction and `A0060000` is the address offset[11]. Therefore, in this example, the only difference in binary is that the address offset was changed from `A0060000` to `CC060000`. Further examination of the source code showed that seven new function declarations and one new variable definition were added before the variable `state` was defined in class `SM_USM_envoy` in one of the header files included in the source file of Release B2. As a result, the offset of the variable `state` was changed accordingly. In this case, the binary code change identified is not a real change and can be ignored in the change identification. We call binary code like `8B89A0060000` is an example of a "*binary code comparison false positive pattern.*" Many such false positive patterns were found in the initial case studies. The full list of these patterns can be found online[2]. The TID-BITZ tool was created to reduce the false positives, and then I-BACCI was promoted to Version 2 to include the use of this tool in Step 2 [32]. TID-BITZ did reduce the false positive rate to less than 8% in the case studies that will be discussed in Section 5, as shown in Table 1.

However, TID-BITZ may introduce false negatives by eliminating real code changes. As shown in Table 1, there were no false negative without using TID-BITZ. The use of TID-BITZ introduced less than 2% false negatives in our case studies. The high false positive rate without TID-BITZ would lead to much more affected functions in both component and application.

Table 1. TID-BITZ Results
(FP is false positive; FN is false negative.)

Comparisons	% of FPs without TID-BITZ	% of FPs with TID-BITZ	% of FNs without TID-BITZ	% of FNs with TID-BITZ
A1 vs. A2	90.1%	5.6%	0%	0%
A2 vs. A3	0%	0%	0%	0%
A3 vs. A4	0%	0%	0%	0%
A4 vs. A5	0%	0%	0%	0%
A5 vs. A6	0%	0%	0%	0%
B1 vs. B2	59.3%	4.9%	0%	0.5%
B2 vs. B3	12.4%	6.1%	0%	1.6%
B3 vs. B4	0%	0%	0%	0%
B4 vs. B5	76.9%	7.7%	0%	0%

4.3 CAAFI

Due to the large amount of functions in the components, it is time-consuming to identify affected functions given changed functions in the components according to the calling relationships

[11] http://developer.intel.com/design/pentium4/manuals/index_new.htm

produced in Step 3 and 4. CAAFI was created in I-BACCI Version 3 to save analysis time and resources. The input to CAAFI is the relocation table set generated by D-TIZ, and changed functions identified by TID-BITZ. First, the relocation table set of a component is converted into an adjacency-matrix [5] to represent call graphs of the functions in the component. For each changed function, CAAFI then backtracks the call graphs to identify all functions that directly or indirectly call the changed function. The output includes a list of all affected functions in the component, and the subgraphs that show how the functions are affected by each changed function. For each case study, CAAFI reduced the time cost from approximately 16 person hours to less than three minutes.

5. CASE STUDIES

Case studies of ABB products have been conducted on I-BACCI Version 1 and Version 2 [32, 33]. For I-BACCI Version 1, an initial case study (henceforth called Case 1) had been conducted on a 757 thousand lines of code (KLOC) ABB application (henceforth called Application A) written in C/C++. Application A uses a 67 KLOC internal ABB software component (henceforth called Component A) of library (.lib) files written in C. Six incremental releases of Component A were analyzed and compared (henceforth referred to as Release A1 through Release A6, respectively). Each Component A release contains a library file with size of about 800 kilobyte. The initial result of the case study indicated that I-BACCI Version 1 can reduce the number of regression tests by 40% on average [31]. 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.

I-BACCI Version 2 was applied on a second case study (henceforth called Case 2) conducted on a 40 KLOC ABB application (henceforth called Application B) written in C/C++. This product uses a 30 KLOC internal ABB software component (henceforth called Component B) of library (.lib) files written in C. Five incremental releases of Component B were analyzed and compared (henceforth referred to as Release B1 through Release B5, respectively) to study the effectiveness of I-BACCI Version 2 for safely reducing regression test cases. Each Component B release contains eight libraries with total size of 1.39 ~ 1.65 megabyte. The full, retest-all strategy takes over four man months of effort to run. With TID-BITZ incorporated into I-BACCI Version 2 to reduce false positives, this case study showed that on average 54% of the number of regression tests can be reduced [32].

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.

In this paper, we report the re-analysis of both case studies with I-BACCI Version 3 which includes all the three tools discussed in Section 4. Additionally, we report the safeness of I-BACCI based upon an examination of the failure records of retest-all black-box testing.

5.1 I-BACCI Version 3 on library files

The library files analyzed in the two case studies contain the raw binary code of many object files. The library files are organized in segments similar to the COFF file format [33]. The calling relationships among functions in the whole component can be ascertained by tracing the calls in the relocation tables throughout the library file.

The analyzer (the fist author of this paper) conducted the first five steps of I-BACCI Version 3. During the first step of the I-BACCI process, each library file in each of the 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 releases and to generate differencing reports. This change identification part (Step 1 and 2) of the case studies 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 22 seconds and 29 seconds in total for Case 1 and 2, respectively, once the cache was warm. TID-BITZ then spent about one second on each comparison and generating a differencing report.

In the third, fourth, and fifth steps, call graphs were drawn for changed functions to identify the affected glue code functions by tracing the affected component functions along the call graphs. CAAFI took about 90 seconds and 170 seconds, respectively, to identify affected component functions for Case 1 and 2. The remaining manual work for each case study, including drawing call graphs for the glue code functions (Step 4) and identifying affected glue code functions (part of Step 5) in the application source code, took the analyzer less than two person hours. The results of the identified changes for all comparisons and all call graphs for the components were preliminarily verified by the analyzer, using source code for the component to determine the accuracy of the analysis post hoc. Then, the second author determined the numbers and percent reduction of the regression test cases needed, based on the list of all the affected glue code functions and the original test suite. The second author also verified the efficacy of the RTS process by examining the failure records of retest-all black-box testing. With the help of the tools, the whole I-BACCI process was done in approximately two person hours for each case study.

5.2 Results of I-BACCI Version 3 on Case 1

The results of applying I-BACCI Version 3 on Case 1 (Application A and Component A) are shown in Table 2. The interface between Application A's glue code functions and Component A was examined, to establish a baseline of affected functions in the application. In total, 60 functions (in 50 C++ files) in Application A call 89 functions of Component A. In the worst case, all 60 functions would be affected by the changes in the component and would need to be re-tested.

The first analysis was conducted between Release A1 and Release A2 of Component A. The BACCI analysis showed that 18 functions were changed out of the 941 functions in Release A2, including three new functions. However, firewall analysis showed that 319 exported functions in Component A were affected by the

identified changes. All 60 functions in Application A were affected. As a result, there was no regression test case reduction.

The second analysis correctly identified 23 changed component functions, and 71 exported functions in the component were affected by the identified changes. Only two glue code functions call the affected exported component functions. Therefore, we achieved 98.7% regression test case reduction.

The latter three analyses identified only a few changes and no function in Application A calls any affected functions in the components, although the changes did affect some exported functions in the components. Therefore, we achieved 100% regression test case reduction for these three comparisons. Overall, approximately an average 80% test case reduction was achieved for these five new releases.

The second author examined the failure records of retest-all black-box testing. There were no regression test failures found.

Table 2. Case 1 Results by I-BACCI Version 3

Metrics	Comparisons				
	1 vs 2	**2 vs 3**	**3 vs 4**	**4 vs 5**	**5 vs 6**
Total changed functions identified	18	23	1	10	3
True positive ratio[12]	100%	100%	100%	100%	100%
False positive ratio[13]	5.6%	0%	0%	0%	0%
Affected exported component functions[14]	319	71	2	55	39
% of affected exported component functions	96.4%	21.5%	0.6%	16.6%	11.8%
Affected glue code functions[15]	60	2	0	0	0
% of affected glue code functions	100%	3.3%	0%	0%	0%
Total test cases needed	592	8	0	0	0
% of test cases reduction	0%	98.7%	100%	100%	100%
Actual regression failures found	0	0	0	0	0
Regression failures detected by reduced test suite	0	0	0	0	0

Compared to the results of I-BACCI Version 1 on this case, the regression test cases reduction increased from 40% to 80%, on average [33], indicating the utility of the TID-BITZ tool.

[12] *True positives ratio* is number of real changed functions found divided by total number of real changed functions.

[13] *False positive ratio* is number of identified changed functions that are not really changes, divided by number of (correctly and incorrectly) identified changed functions.

[14] *Affected exported component functions* are functions within the COTS component that interfaces with application, either changed or affected by other component functions, as illustrated in Appendix A.

[15] *Affected glue code functions* are functions within the glue code that directly call affected exported component functions, as illustrated in Appendix A.

Additionally, the postmortem source code difference analysis showed that the change identification was correct except that one false positive existed in the comparison between Release A1 and Release A2. No false negative was found in all analyses.

5.3 Results of I-BACCI Version 3 on Case 2

Similarly, the results of applying I-BACCI Version 3 on Case 2 (Application B and Component B) are shown in Table 3.

Table 3. Case 2 Results with I-BACCI Version 3

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
% 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

The interfaces between Application B's glue code functions and the Component B were also examined to establish a baseline of affected functions in the application. The glue code functions changed in Release B3. For the former version of glue code functions (i.e. in Release 2), there are 123 exported functions in the component. In total, 46 glue code 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 glue code functions in the latter version would be affected.

The first analysis was conducted between Release B1 and Release B2 of the component. The BACCI analysis showed that 388 functions were changed out of 1143 functions in Release B2. Firewall analysis showed that 84 exported functions in Component B were affected by the identified changes and 38 glue code functions were affected. As a result, 30% of the regression test cases can be reduced.

More reduction was achieved in the latter two comparisons: only 5% and 9% of the test cases needed to be re-run. However, due to the great extent of changes between Release B2 and B3, no regression test case reduction was found.

Examination on the failure records of retest-all black-box testing indicated the safeness of I-BACCI Version 3 process as all 13

regression test failures would be detected by the reduced regression test suite.

In the second case study, source code difference analysis showed that 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%), as shown in Table 1. The false negatives were caused because a changed function contained a function call which was replaced by another function call. TID-BITZ ignored the address changes of the changed function call. Additionally, with the help of CAAFI, the time cost of the whole process reduced from about 24 person hours [32] to two hours. As shown in the results of the two case studies, I-BACCI is more effective when there are small incremental changes between revisions.

5.4 Case Study Limitations

A limitation of the case studies is that all the applications and components used were software developed by ABB Inc. involving .lib library files. We chose to conduct our research in this manner because, ultimately, we needed to check the accuracy of our process and needed to develop and evolve the support tools by manually examining source code. We strived to maintain as much objectivity as possible in our work. The first author conducted the I-BACCI process in both case studies. The second author then revealed the actual changes and the black-box testing results.

6. LESSONS LEARNED

We list the experiences with our effort of regression test selection when source code is not available in order of importance as follows:

- We gathered 28 license agreements to investigate the legality of analyzing binary code of purchased COTS components. Relevant sentences in the license agreements were reviewed by lawyers of North Carolina State University. The lawyers deemed that the approaches and algorithms used in I-BACCI process are legal due to the purpose of the analysis. Many of the license agreements of commercial components prohibit the user of components from reverse engineering, decompiling, disassembling, or otherwise attempting to discover the source code of the software except to the extent that this restriction is expressly prohibited by law. Copyright law does not prohibit the analysis on the code, only prohibit reproducing the components, making derivative works, or distributing copies of the products. The purpose of our research is to reduce the testing required when components change and only binary code and documentation is available. The information of the components that we need in the analyses is in function level (function signatures and function calling relationships) instead of the source code or any logic/algorithm of the software.
- Currently the TID-BITZ tool works only when the releases of components are built by the same compiler. An initial run of I-BACCI on the second and fourth comparisons of Case 1 revealed that more than 95% of the initially-identified changes were not real changes (a.k.a. false positives). A re-analysis on versions rebuilt by the same compiler showed the results discussed in Section 5.2.
- In Case 2, all the regression test failures can be detected by the reduced regression test suite. However, for Case 1, no regression test failures were found. Therefore, we could not obtain support for the safeness of I-BACCI through this case study. In the future, we will select case studies that contain regression test failures.

7. CONCLUSIONS AND FUTURE WORK

ABB desires a method of selecting regression tests when the COTS components included in their applications change. In this paper, we proposed Version 3 of the I-BACCI process for regression test selection for applications that use software components when source code is not available. Two case studies were examined to verify the potential efficacy of this process. The results showed that I-BACCI is an effective RTS process for COTS-based applications. We achieved at least 70% regression test case reduction for the nine new releases of these two case studies. The safeness of the RTS process had been verified by examining the failure records of retest-all black-box testing. No failures would have escaped our reduced test suite. I-BACCI can most beneficial when there are small incremental changes between revisions. These results supported our theory:

When components change and only binary code and documentation are available, regression test selection can safely be based upon the glue code that interfaces with sections of the component that changed.

We plan to pursue several directions in our future work. First, additional breadth is required to expand this process to adapt to more of the COTS file types and programming languages. 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. Additionally, to address the limitations of I-BACCI which are discussed in Section 3.2, we will explore other approaches, such as applying other code-based regression selection techniques and examining other available artifacts of COTS components. Also, we will consider changes caused by factors other than source code (e.g. build tools, environment, and target platforms), and improve the supporting tools to remove as many false negatives as possible.

8. ACKNOWLEDGEMENTS

This research was supported by a research grant from ABB Corporate Research. We would like to thank Dr. Tao Xie and the North Carolina State University Software Engineering Realsearch Reading Group for their helpful suggestions. Additionally, we thank North Carolina State University council Mrs. Judy Curry and Mr. David Drooz for their help in analyzing component licensing agreements.

9. REFERENCES

[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] Apiwattanapong, T., Orso, A., and Harrold, M. J., "A Differencing Algorithm for Object-Oriented Programs," 19th International Conference on Automated Software Engineering (ASE'04), Linz, Austria, 2004, pp. 2-13.

[3] Basili, V. R. and Boehm, B., "COTS-Based systems Top 10 List," *IEEE Computer*, vol. 24, no. 5, 2001, pp. 91-93.

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

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

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

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

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

[9] 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.

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

[11] Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., and Chen, C., "Change Impact Identification in Object-Oriented Software Maintenance," International Conference on Software Maintenance, Victoria, Canada, 1994, pp. 202-211.

[12] 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*, Vol. 8, No. 2, 1995, pp. 51-65.

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

[14] 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.

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

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

[17] Memon, A. M., "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.

[18] 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, Scotland, 2004, pp. 491-500.

[19] Ren, X., Ryder, B. G., Stoerzer, M., and Tip, F., "Chianti: A Change Impact Analysis Tool for Java Programs," the 27th International Conference on Software Engineering, St. Louis, MO, USA, 2005, pp. 664-665.

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

[21] Srivastava, A., "Vulcan," TR-99-76, MSR 1999.

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

[23] Vokolos, F. and Frankl, P., "Pythia: A regression test selection tool based on textual differencing," 3rd International Conference on Reliability, Quality and Safety of Software-intensive System, Athens, Greece, 1997, pp. 3-21.

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

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

[26] Weyuker, E. J., "Testing Component-Based Software: A Cautionary Tale," *IEEE Software*, 15(5), 1998, pp. 54-59.

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

[28] 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.

[29] 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.

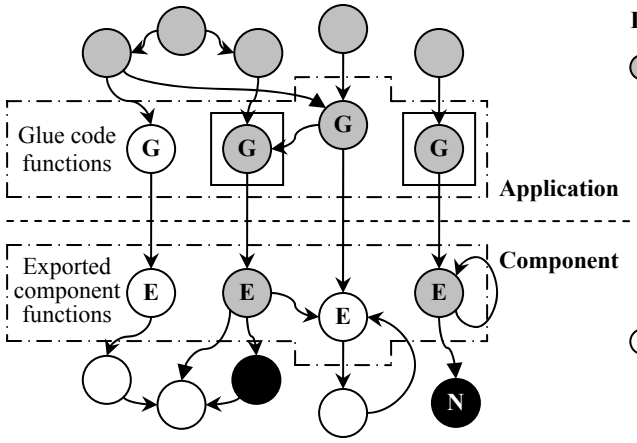
[30] White, L. and Robinson, B., "Industrial Real-Time Regression Testing and Analysis Using Firewall," International Conference on Software Maintenance, Chicago, 2004, pp. 18-27.

[31] 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, 2005.

[32] Zheng, J., Robinson, B., Williams, L., and Smiley, K., "A Lightweight Process for Change Identification and Regression Test Selection in Using COTS Components," the 5th International Conference on COTS-Based Software Systems, Orlando, FL, USA, 2006, pp. 137-143.

[33] 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," the 16th IEEE International Symposium on Software Reliability Engineering, Chicago, IL, USA, November, 2005, pp. 225-234.

Appendix A. Illustration of the terms



Legend

- Each circle stands for a *function*. Each arrow connector stands for a *function call*.
- The dotted line comparts *application* and *component* functions.
- Black circles stand for the *changed functions*.
- Black circles with letter 'N' stand for the *added functions*.
- Gray circles stand for the *affected functions* (functions that are affected by the changed or added component functions).
- White circles stand for the *unchanged or unaffected functions*.
- Circles with letter 'G' and 'E' stand for the *glue code functions* and *exported component functions*, respectively.
- Real line squares stand for the *firewalls*. Functions in the firewall are the *affected glue code functions*.

Appendix B. An example of DUMPBIN output

Source code in both Release B1 and B2

```
int SM_USM_envoy::GetState(int s) {
   return state==s;
}
```

Corresponding section in the output of DUMPBIN for Release B1

```
SECTION HEADER #78
   .text name
       0 physical address
       0 virtual address
      20 size of raw data
    722B file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60501020 flags
         Code
         Communal; sym= "public: virtual int __thiscall SM_USM_envoy::GetState(int)"
         16 byte align
         Execute Read
RAW DATA #78
  00000000: 8B 89 A0 06 00 00 8B 54 24 04 33 C0 3B CA 0F 94
  00000010: C0 C2 04 00 90 90 90 90 90 90 90 90 90 90 90 90
```

Corresponding section in the output of DUMPBIN for Release B2

```
SECTION HEADER #79
   .text name
       0 physical address
       0 virtual address
      20 size of raw data
    73D1 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60501020 flags
         Code
         Communal; sym= "public: virtual int __thiscall SM_USM_envoy::GetState(int)"
         16 byte align
         Execute Read
RAW DATA #79
  00000000: 8B 89 CC 06 00 00 8B 54 24 04 33 C0 3B CA 0F 94
  00000010: C0 C2 04 00 90 90 90 90 90 90 90 90 90 90 90 90
```