An Initial Study of a Lightweight Process for Change Identification and Regression Test Selection When Source Code is Not Available

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 to improve testing cost effectiveness via improving efficiency. 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 to guide in regression test selection. In this paper we describe 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 binary code analysis. A case study of the process was conducted on an ABB product that uses a medium-scale internal ABB software component. Six 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 regression tests by 40% 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]. A variety of regression test selection techniques [3, 5, 15] have been developed to minimize the time and resource cost of regression testing. However, most of these techniques rely on source code, and therefore are not suitable when source code is not available for analysis.

COTS software products typically undergo a new release every eight to nine months, with active vendor support for only the latest three releases [2]. 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. Upon receiving the COTS files, users often need to conduct regression testing to determine if a new component or new version of an existing component will cause problems with their existing software and/or hardware system. The lack of source code presents a challenge for the reduction and selection of test cases.

Our research objective is to develop a lightweight process for regression test selection for the user/glue code that uses 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. The input artifacts are the binary code of the components (old and new versions), the source code of user/glue code, and the test suite for the user/glue code. Generally these artifacts are available to the COTS user. Once the process is completed, the reduced regression test suite can be run to determine if any of the changes in the COTS components affected the operation of the application.

A case study of the six-step I-BACCI process was conducted by North Carolina State University and ABB Inc. The case study involved a large-scale ABB product that contains a medium-scale internal ABB software component. In prior research, we applied the first two steps of I-BACCI on four releases of a product [25]. In this case study, all six steps of I-BACCI were applied to six releases of the product.

The remainder 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 describes a case study of applying this process on an ABB product that uses a library component. Finally, Section 5 and Section 6 present the conclusions and future work, respectively.

2. Background and related work

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

2.1. Testing of software components

Generally, testing of COTS software is black-box 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 mechanism 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 components can only be derived from the component specification provided by the vendor, and the behavior can only be determined by studying the inputs and the related outputs of the component. 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].

Harrold et al. [6] presented techniques that use component metadata for regression test selection of COTS components. They illustrated their technique with a controlled example of a `VendingMachine` program with a `Dispenser` component. Their code-based technique resulted in an average savings of 26% of the testing effort over seven releases of a real component-based system [6]. 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 [6]. However, the component provider may not provide this information. In our research, we focus on using only the information that is typically available. However, Harrold et al.'s process may be more applicable when component licensing agreements preclude the binary code analysis needed for I-BACCI.

2.2. Regression test selection

The *retest-all* regression technique, whereby all regression tests are re-run, is straightforward but can be prohibitively expensive in both time and resources [5]. Conversely, regression test selection (RTS) techniques attempt to reduce the cost of regression testing by selecting a partial set of possible regression test cases [5]. The selected regression test suite focuses on the software components/functions that have been changed or that 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, 11, 15]. A variety of RTS techniques [3, 5, 15] 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, however, have developed a test prioritization system, Echelon [17], that prioritizes the application's given 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 the test coverage information of the older version (in the form of 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 [17]. The objective of the comparison was to see if Echelon detected defects earlier. In the first run, Echelon detected 87% of the defects in the first 2 of 148 test sequences; the remaining 13% of the defects were not detected by any tests. In the second run of different binaries, Echelon detected 98% of the defects in the first 3 of 221 test sequences; the remaining 2% of the defects were not detected by any tests.

Srivastava and Thiagarajan discuss 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 to constants, macro definitions, etc. have been propagated to the affected procedures, simplifying the determination of program changes. 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 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 to identify changes or the change impact via impact analysis [14] between the new release and the previously-tested version with the same source code base. Laski and Szermer [10] proposed a formal method to identify modifications made in a program. Vokolos and Frankl [18, 19] utilized a textual differencing technique to perform regression test selection. However, most change identification approaches utilize the source code of the old and modified programs [10, 15, 18, 19]. These approaches are not suitable for component testing when source code is not available.

Although a comparison between versions of documentation (such as user manuals, specifications, and samples) is potentially helpful [11, 13], the documentation may not reflect all changes. In some cases, the implementation may change without necessitating any specification changes, such as for a code fix. Thus, to identify an efficient set of regression tests, users of COTS software should perform thorough change identification which does not rely solely on the component documentation. I-BACCI addresses this.

For the purpose of evolution of user profile information, Wang et. al. [20] developed the Binary Matching Tool (BMAT). BMAT matches 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. BMAT uses the Vulcan binary analysis tool [16] 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 allows good matches to be found even with shifted addresses, different register allocations, and small program modifications [20]. BMAT was used by Echelon [17], which is discussed in Section 2.2, to find a matching block in the old binary for each block in the new binary. The BMAT algorithm may be incorporated into our supporting tool for I-BACCI to reduce or eliminate a current false positive problem.

3. I-BACCI

I-BACCI is an integrated, lightweight regression test selection process for user/glue code that uses software components for which source code of the components is not available. The I-BACCI process is an integration of our Black-box Approach for Component Change Identification *(*BACCI) process for identifying change with the firewall RTS method. In this section, we provide information on BACCI, firewall analysis, and I-BACCI.

3.1. BACCI

We have proposed the BACCI process for identifying changed areas in COTS components [25]. The first step of the BACCI process is to decompose the binary files of the components into code sections of exported functions using appropriate binary parsers and using the open source Decomposer and Trivial Information Zapper (D-TIZ)[1] tool. The second step of the process is to compare the code sections between the two versions using standard differencing tools. The goal of BACCI is to feed the change information of various types of binary code into code-based regression test selection methods.

In a feasibility study, the proposed BACCI process was applied three times between successive released versions of an internal ABB product. The result is shown in Table 1. For each comparison, the two numbers in the column of "Numbers of Functions" represent the numbers of the functions in the two releases being compared respectively.

Table 1: Feasibility Study Results

Comp. Releases	Number of Fcns.	Changed Functions Identified	True Pos.	False Pos.	False Neg.
1 and 2	941 / 941	1	100%	0	0%
2 and 3	941 / 941	664	100%	465	0%
3 and 4	941 / 942	2	100%	0	0%

The analysis of Releases 1 and 2 identified a change in one of 941 functions in the library. Once the changed function was determined, a source code difference analysis was performed which showed that the BACCI analysis was correct and only the one identified function was changed between Releases 1 and 2. Similarly, the analysis of Releases 3 and 4 showed that two functions were changed out of the 941 functions in Release 3, and one function was added. The source code difference analysis of the two versions verified that two functions changed and one new function was added. These functions were the same functions identified by the BACCI analysis.

The BACCI analysis of Releases 2 and 3 identified that 664 out of 941 functions in the library were changed. The source code difference analysis was performed, and it was determined that only 199 out of the identified 664 function differences were really due to changes in the code. After further investigation, it was determined that the product had changes to global structures and definition statements. These changes affected the product's uses of void function pointers to allow function callbacks. All addresses of these functions changed, which led to changes in the library for 450+ functions. Other than these changes, the other 199 functions were identified correctly. The false positives lead to a larger bound on the testing needed, but this is still safe, as no areas are missed. Future work has been planned to reduce or even eliminate false positives by improving the D-TIZ tool, potentially using the BMAT [20] algorithm.

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

The above case study indicates the potential of BACCI for determining change in COTS components when no void function pointers or other static addresses are used. These pointers lead to false positives. No false negatives were identified.

3.2. Firewall analysis

Leung and White [1, 11, 12, 23] developed the testing firewall method for regression testing with integration test cases (where integration tests are those that evaluate the interaction between components [7]) in the presence of small changes in functionally-designed software. The testing firewall is intended to limit the regression testing to those potentially-affected system elements directly dependent upon changed system elements [23, 24].

The firewall method considers module dependencies, control-flow dependencies, and data dependencies [23]. 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 in the firewall [23]. 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, Gao, et. al. [8, 9] utilized the firewall concept on an object-oriented system, and White and Abdullah [21] expanded the firewall to address more features of an object-oriented system. Firewall was also utilized in the regression testing of graphical user interfaces by White et al. [22].

The firewall method can only be guaranteed to select all modification-revealing [15] tests and to be safe if all unit and integration tests initially used to test system components are "reliable" and observability can be assured. Tests are reliable if the correctness of modules exercised by those tests for the tested inputs implies correctness of those modules for all inputs. However, test suites are typically not reliable in practice [24], so the firewall technique may omit modification-revealing tests and may also admit some non-modification-traversing tests. White and Robinson [24] have shown firewall to be effective via empirical studies of industrial real-time systems despite these theoretical limitations.

3.3. I-BACCI

The I-BACCI process involves six steps as shown in Figure 1. The first two of these steps are done via the BACCI process (in dash-dotted line frame), and the remaining four steps are done via the firewall analysis process (in dashed line frame). Steps 2, 3, and 4 can be performed concurrently. The goal of the BACCI process is to prepare a report on changed functions and the calling relationships among the functions in the components. The input artifacts to the I-BACCI process are shown in the gray blocks of Figure 1. They are fed into different steps of the process.

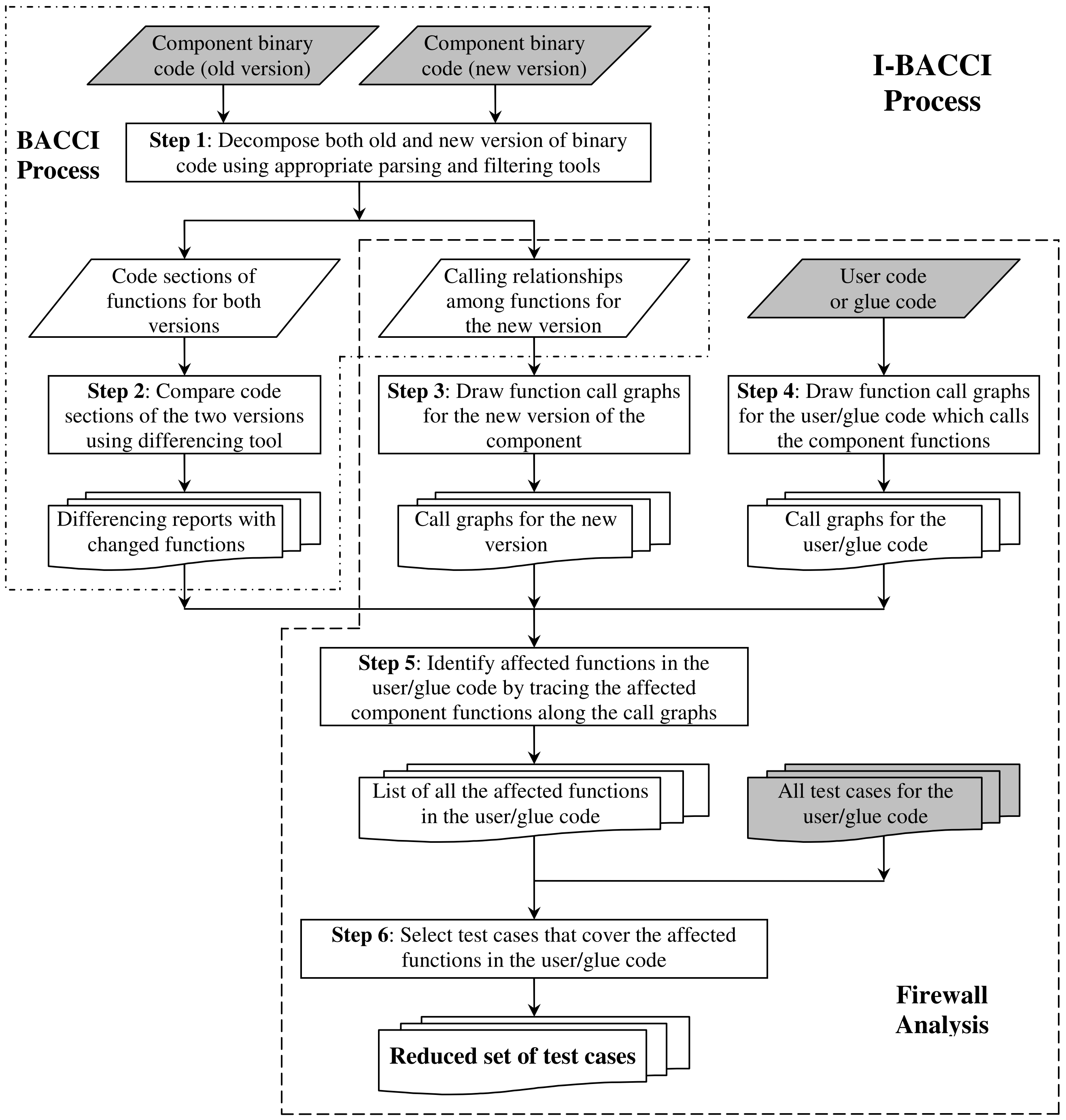
Applying the firewall analysis process for regression test selection within I-BACCI requires the user/glue code, its full test suite, and the output of BACCI. Similar to traditional firewall analysis, the changes determined by the BACCI process are identified and traced one level up (or more than one, depending on data-flow paths). Since we do not have the test cases for any internal functions of the component itself, each identified change is propagated to the roots of the call graph for the component, and all of the functions in the user/glue code that directly or indirectly call this function are identified for retesting.

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 relationship of the source code is stored in some areas of the binary files according to pre-defined formats, so that an external system is able to find and call the functions in the corresponding code sections. For example, Windows NT® uses a special format for the executable (image) files and object files. The format used in these files is referred to as Portable Executable (PE) or Common Object File Format (COFF)[2]. Object files created from C or C++ programs using many compilers conform to COFF, including Microsoft® Visual C++™, GNU® Compiler Collection (GCC™), and Intel® C/C++ Compiler (ICL™). The overall structure of COFF is shown in Table 2.

Table 2: Overall Structure of COFF

Segment	Description
File Header	Stores basic information of the COFF file.
Optional Header	Optional file header; generally does not exist in object files.
Section Header 1 ~ n	Describes section information.
Section Data	Raw data.
Relocation Directives	Describes relocation information for symbols in the COFF file; exists in object files only.
Line Numbers	Maps binary code with line nos. of source code for debugging.
Symbol Table	Describes information for all symbols used in the COFF file.
String Table	Stores long symbol names

[2] MSDN Library - Visual Studio .NET 2003



Component binary code (old version)
Component binary code (new version)
I-BACCI Process
BACCI Process
Step 1: Decompose both old and new version of binary code using appropriate parsing and filtering tools
Code sections of functions for both versions
Calling relationships among functions for the new version
User code or glue code
Step 2: Compare code sections of the two versions using differencing tool
Step 3: Draw function call graphs for the new version of the component
Step 4: Draw function call graphs for the user/glue code which calls the component functions
Differencing reports with changed functions
Call graphs for the new version
Call graphs for the user/glue code
Step 5: Identify affected functions in the user/glue code by tracing the affected component functions along the call graphs
List of all the affected functions in the user/glue code
All test cases for the user/glue code
Step 6: Select test cases that cover the affected functions in the user/glue code
Reduced set of test cases
Firewall Analysis

Figure 1: Proposed I-BACCI Regression Test Selection Process

The compiled functions and data structures are stored in sections in the "Section Data" segment in binary form. In addition, the "Relocation Directives" segment saves the information for the symbol indices which point to corresponding records in the symbol table, and the offsets of the symbols in code sections. Therefore, the calling relationships of the functions can be derived from the "Relocation Directives" segment. We can utilize the information stored in the section headers (such as data offset, real size of section data, the offset of relocation information, and the data in Relocation Directives, Symbol Table, and String Table) to deduce information about the source code of the functions.

There are two sub-steps for **the first step** of the BACCI process: (1a) decomposing the binary file of the component; and (1b) filtering trivial information to facilitate comparisons by differencing tools. Often the first sub-step can be accomplished by parsing tools available for the language/architecture. The second sub-step is frequently necessary because the output may contain trivial information such as timestamps and file pointers, which are "noise" for the change identification. The first output should be formatted conveniently for differencing tools to identify changes in functions between releases, and the second output should be formatted conveniently for a graph generation tool to build call graphs. For example, the 32-bit COFF binary files, such as COFF object files, standard libraries of COFF objects, executable files, and dynamic-link libraries (DLLs), can be examined by the Microsoft COFF Binary File Dumper (DUMPBIN). The output generated by DUMPBIN presents all the information about the 32-bit COFF binary files in a comprehensible manner suitable for use as input to differencing tools.

Generally, the second sub-step cannot be done via existing tools. Therefore, we have created the D-TIZ to perform the decomposition and remove trivial information. Currently D-TIZ can only be used with library 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 the code sections between the two versions. Commercial or open source differencing and merge tools (such as Araxis Merge [3], FolderMatch [4], Beyond Compare [5], and WinMerge [6]), which allow for the comparison of not only plain text files but also binary files, are able to accomplish this step and generate differencing reports showing the changed functions.

The third and fourth steps of the I-BACCI process are necessary to 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/glue source code. Generally, the call graphs generated from Step 3 are more complex than those from Step 4, because in Step 4 only the user functions that directly call the component functions and the component functions being called 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] (an open source tool). For convenience in identifying affected functions in the user/glue code, the call graphs generated from the two steps can be integrated.

In **the fifth step**, the affected functions in the user/glue code are identified. This step can be implemented by algorithms in directed graph theory. If a component function changes, then all its direct callers are considered as affected functions. The direct callers that call these affected functions are also potentially affected by the initial function change. Therefore, we can start from each component function identified as changed and propagate that change along the call graphs until we reach the functions called directly by the user/glue code (which we call the "user" functions). These are the user functions which are affected by the initial changed function in the component, and therefore need to be re-tested.

This method 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 that directly call component functions, but many component function changes. We may start from the user functions and examine the component functions being called by them along the call graphs, until we find a changed component function or we have reached all of the leaves without finding a changed component function. In the former situation, the initial user function is affected by the change in the component, so that it needs to be re-tested, while the latter situation indicates the initial user function does not need to be tested. The output of Step 5 is a list of all the affected functions in the user/glue code.

Generally, there is a set of test cases for each function in the user/glue code. More generically, I-BACCI requires as input a set of test cases which are mapped to the user/glue code functions they cover. In **the sixth step,** we use this information to select test cases that cover only the affected functions in the user/glue code, 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 area in the user/glue code.

3.4. Limitations of I-BACCI

I-BACCI has a legal limitation: the licensing agreement of the COTS component must not preclude the analysis of the binary files. Although licensing agreements are generally intended to prevent someone from creating viruses or competing products or circumventing copy protection, we should avoid violation of any license agreements for each product we are analyzing, including the internal ABB products.

I-BACCI shares an acknowledged technical limitation with all existing source-based firewall methods: it has the potential for reporting false negatives in situations where binary differences are caused by factors other than changes in source code (e.g. changes in the 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 focus of the method on

[3] http://www.araxis.com
[4] http://www.foldermatch.com
[5] http://www.scootersoftware.com
[6] http://winmerge.sourceforge.net
[7] http://www.graphviz.org

decompilation to source for differential analysis precludes identification of such differences.

A third limitation of I-BACCI is its potential for identification of false positives. In tracing the call graphs, we are consciously and conservatively assuming for test selection purposes that any use of a binarily-changed called function will be affected by the change, even though that particular use of a changed function might never exercise the changed logic or data. It might be possible, with further work, to prove this and thus eliminate unneeded tests 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.

4. Case study

An I-BACCI case study was conducted on a 757 thousand lines of code (KLOC) ABB application written in C/C++. This product contains a 67 KLOC internal ABB software component in library (.lib) files written in C. Six incremental releases of the component were analyzed and compared to study the effectiveness of the I-BACCI process at reducing regression test cases. Henceforth, these releases will be referred to as Release 1 through Release 6, respectively. The releases identified as 1 though 4 in our previous BACCI work correspond to Releases 3 through 6 in this study. This software combination was chosen because (1) the numbers of test cases for each function of the application are available; and (2) multiple releases of the component are available.

Source code for the component was available and was used to verify the accuracy of the analysis post hoc. In this case study, the only artifact in each release of the component which was used by the analyzer (the first author of this paper) is a library file in the delivery package; neither the header files nor the documentation were analyzed. The analyzer did not have access to the source code of the component and did not know the changes in source code of the component. The analyzer had access to the source code of the application to analyze the use of the component functions and to draw call graphs for the interface between the user software and the component. The results of the identified changes and call graphs were verified by the second author, who was not involved in the detailed change identification analysis and firewall analysis. The reduction of test cases for each comparison was provided by the second author.

The rest of this section is organized as follows. Section 4.1 describes the process for analyzing library files; Section 4.2 presents the results of the case study.

4.1. I-BACCI processing of library files

The case study involved the analysis of library files. A library file contains the raw binary code of many object files. When we call a function implemented in an object file congregated in a library, a linker program is able to seek the corresponding object file in the library and invoke the function. A library file is organized in segments similar to the COFF file format. Each segment consists of two parts - header and data. The number of Object Sections and offset of each Object Section can be found in the Second Section. The data part of each Object Section is a complete unchanged object file in COFF format, as discussed in Section 3. The names of the object files can be obtained either in the corresponding header or in the Longname Section, according to the offset stored in the header. Also, the relocation table (Relocation Directives segment in Table 2) of a function in the object file stores the names of the functions that are called by that function. 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, the binary files of components were decomposed into code sections of functions. Each library file in each of the six releases was translated into plain text using DUMPBIN. Afterwards, D-TIZ was used to scan the output of DUMPBIN to pick out the code sections of the exported functions, and save them into separate files. The relocation table of each function was obtained by D-TIZ.

The second step was to compare the functions among the six releases and generate differencing reports. The differencing tool selected was Araxis Merge. In the differencing report, every function contained in the two library versions being compared is listed and can be drilled down to see detailed differences between the two releases. The report can be configured to list only those functions with changes.

The 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. It took nine seconds in total for DUMPBIN and D-TIZ to complete the first step of the process. Araxis Merge spent about five seconds on each comparison and about one minute generating the full 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. Currently we have to conduct the firewall analysis steps manually due to the lack of existing tool support. A typical call graph is shown in Appendix A. It took approximately 24 hours for the analyzer to complete the three steps for the six versions compared. In the future, a tool will be developed to generate call graphs and automate the identification of affected functions. 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.

4.2. Results

In this case study, the proposed I-BACCI process was applied five times between six successively-released versions of the internal ABB component. The result is shown in Table 3.

Table 3: Case Study Results

Metrics	Comparisons				
	1 vs. 2	2 vs. 3	3 vs. 4	4 vs. 5	5 vs. 6
Changed component functions	164	668	1	664	2
Added component functions	3	2	0	0	1
Deleted component functions	4	2	0	0	0
Affected exported component functions	331	331	2	331	39
Affected functions in the application	60	60	0	60	0
Total test cases needed	592	592	0	592	0
% of reduced test cases	0 %	0 %	100 %	0 %	100 %

The interface between the application and the internal component was examined to establish a baseline of affected functions in the application. In total, 60 functions (in 50 C++ files) in the application call 89 functions of the component. In the worst case, all of the 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 1 and Release 2 of the component. The BACCI analysis showed that 164 functions were changed out of the 941 functions in Release 2, and three functions were added. Once the changed functions were determined, a source code difference analysis was performed which showed that only 70 out of the identified 164 function differences were really due to changes in the source code from Release 1 to Release 2. The remaining changes are due to the false positives discussed in Section 3.1. Firewall analysis showed that 331 exported functions in the component were affected by the identified changes and all 60 functions in the application were affected. As a result, there was no regression test case reduction.

The second and fourth analyses determined that more than 660 out of 941 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 application functions that call component functions. Unfortunately, all of these component function calls were affected. Similar to the first analysis, approximately 70% of the component functions that were marked as "changed" by D-TIZ were false positives.

The third analysis was conducted between Release 3 and Release 4 of the component. This analysis identified a change in one of 941 functions in the library. The source code difference analysis showed that the BACCI analysis was correct and only the identified function was changed between Releases 3 and 4. Two exported functions in the component were affected by the identified change. However, no function in the application calls the two affected functions in the component. Therefore, we achieved a 100% regression test case reduction.

The final analysis was conducted between Release 5 and Release 6. This analysis showed that two functions were changed out of the 941 functions in Release 6, and one function was added. The source code difference analysis of the two versions verified that two functions changed and one new function was added. These functions were the same functions identified by the BACCI analysis. These changes affect 39 exported functions in the component. The call graph of this case is shown in Appendix A. However, similar to the third case, no functions in the application call the 39 affected functions in the component, and therefore none of the functions in the application need to be retested.

5. Conclusions

In this paper, we proposed the I-BACCI process for regression test selection for user/glue code that uses software components when source code of the components is not available. A large-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 determined from a component without any source code available for analysis. In the above case study, there were times when releases required no retesting, as no changes in the component affected the product using the component.

6. Future work

Accuracy, generalization, and automation are three main goals of work in the future. First, in depth, we need to reduce or even eliminate the false positives identified by the current I-BACCI process and tools. 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. Finally, the whole process should be automated into one tool to save both time and resources.

Acknowledgements

This research was supported by a research grant from ABB Corporate Research. Additionally, we would like to thank Tao Xie for his helpful suggestions.

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] Basili, V. R. and Boehm, B., "COTS-Based systems Top 10 List," *IEEE Computer*, Vol. 24, No. 5, May 2001, pp. 91-93.

[3] 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*, Vol. 10, No. 2, Apr. 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.

[5] 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*, Vol. 10, No. 2, Apr. 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 (ICSM 2001), Florence, Italy, Nov. 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., "Change Impact Identification in Object-Oriented Software Maintenance," International Conference on Software Maintenance, Victoria, B.C., Canada, 1994, pp. 202-211.

[9] 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, May 1995, pp. 51-65.

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

[11] 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.

[12] 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.

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

[14] 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 (ICSE), Edinburgh, Scotland, 2004, pp. 491-500.

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

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

[17] 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.

[18] 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, Jan. 1997, pp. 3-21.

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

[20] 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.

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

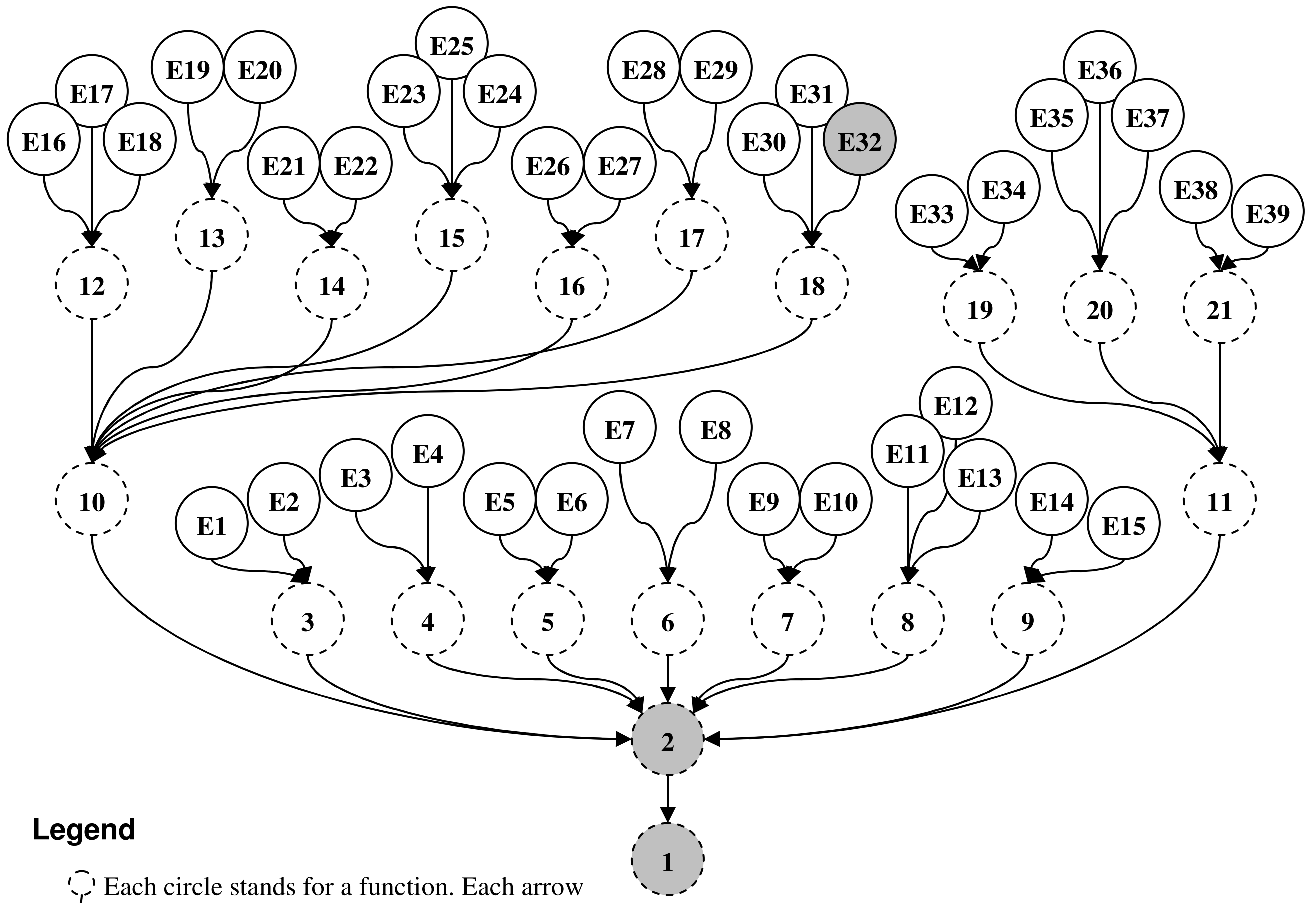
[22] White, L., Almezen, H., and Sastry, S., "Firewall Regression Testing of GUI Sequences and Their Interactions," International Conference on Software Maintenance, Amsterdam, The Netherlands, Sept. 2003, pp. 398-409.

[23] 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.

[24] 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.

[25] 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.

Appendix A. Changed and affected functions in Release 6



Legend

Each circle stands for a function. Each arrow connector stands for a call. The numbers in circles are just sequence numbers but do not imply anything about order.

Gray circles stand for the three changed or added functions.

White circle stands for function affected by the changed functions.

Circle with dotted edge stands for non-exported functions.

Circle with solid edge stands for exported functions. The prefix “E” before the numbers in such circle stands for “Exported”. 39 exported functions are affected in this case.