A Process for Identifying Changes 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, 27695
{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 shown to improve fault detection effectiveness. The majority of these test selection techniques rely on access to source code for change identification. However, when new releases of COTS components are made available for integration and testing, source code is often not available to guide regression test selection. This paper describes a process for identifying changed functions when code is not available. This change information is beneficial for selecting white-box regression tests of customer/glue code. This process is applicable when COTS licensing agreements do not preclude decompilation. A feasibility study of the process was conducted with four releases of a medium-scale internal ABB product. The results of the feasibility study indicate that this process can be effective in identifying changed functions.

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

General Terms
Reliability, Verification.

Keywords
Change Identification, COTS.

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 [5]. To minimize the time and resource cost of regression testing, a variety of regression test selection techniques have been developed [2, 4]. However, regression test selection techniques that rely on source code are not suitable for COTS components.

ICSE-MPEC'05, May 21, 2005, St. Louis, Missouri, USA.

COTS software products typically undergo a new release every eight to nine months on average, with active vendor support for only its latest three releases [1]. Customers of COTS components often do not have access to the source code, only to binary files of the components and a small set of reference documents. Upon receiving the COTS files, customers often desire to run regression tests to determine if a new component or new version of an existing component causes 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 process for change identification for software components for which source code is unavailable.* The purpose of our process is to bridge the gap between code-based regression test selection techniques and components without source code. We call our process Black-box Approach for Component Change Identification (BACCI). Once the changes are identified, existing code-based regression test selection techniques [2, 4] can be applied on customers' systems/glue code interface with COTS components.

A feasibility study of the two-step BACCI process was conducted via collaborative research between North Carolina State University and ABB Inc. The feasibility study involved four releases of a medium-scale internal ABB product. The remainder of this paper is organized as follows. Section 2 discusses the background and related work. The BACCI process is described in Section 3. Section 4 describes a case study of applying this process on library files of the ABB product. Finally, Section 5 presents the conclusions and future work.

2. BACKGROUND AND RELATED WORK
In this section, we discuss the prior work in testing of COTS components, regression testing, and change identification.

2.1. Testing of COTS components
Generally, customer 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 [5]. 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 the access to component source code and

internal artifacts is one of the issues and challenges of component testing [3].

2.2. Regression testing

Regression testing is applied on modified software to provide confidence that changes did not introduce unintended behavior or additional errors into previously-tested code. Re-running all the test cases, the most straightforward regression testing approach, can be prohibitively expensive in both time and resources [4]. Regression test selection concentrates on the tradeoff between the time required to select and run test cases and the fault detection effectiveness of the test cases that are run [4]. The selected regression test suite often focuses on the software components/functions that have been changed or that are most likely to be affected by the change. A variety of regression test selection techniques have been proposed such as minimization techniques, dataflow techniques, and safe techniques [2, 4]. However, no existing regression test selection techniques, except for simple random techniques, deal with component regression testing when source code is not available [2, 4, 9].

2.3. Change identification

Because regression testing is the process of verifying modified software, one of the key steps is to identify changes or change impact between the new release and the previously-tested version with the same source code base. Laski and Szermer [6] proposed a formal method to identify modifications made in a program. Vokolos and Frankl [10, 11], utilized textual differencing technique to perform regression test selection. However, most change identification approaches rely on the analysis of information about the code of the program and the modified version [6, 9-11]. They are not suitable for component testing when source code is not available. Additionally, while a comparison between versions of documentation (such as user manuals, specifications, and samples) is potentially helpful [7, 8], it is possible that not all changes are reflected in the documentation. In some cases, the implementation may change without necessitating any specification changes such as for a code fix. Thus, users of COTS software should perform thorough change identification to identify the code modifications that affect their systems.

3. BACCI Process

The BACCI process involves two steps as shown in the dotted line frame in Figure 1. The larger objective of using this information to feed regression test selection is also shown in this figure, extended outside of the dotted frame. The input to the BACCI process is component binary files, such as .lib, .dll, .ocx, or .class files, as shown in the first data block of Figure 1. Prior to being distributed, COTS source code is compiled into one of these binary code formats. However, 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. 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) files[1]. Object files created from C or C++ programs using many compilers conform to COFF, including Visual C++, GNU Compiler Collection (GCC), and Intel C/C++ Compiler (ICL). The overall structure of COFF is shown as Table 1.

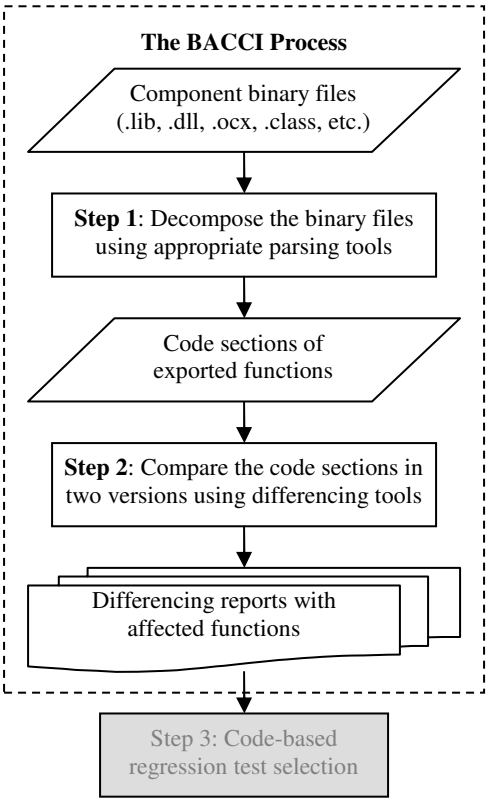

Figure 1: Proposed COTS Regression Test Selection

The compiled functions and structures are stored in sections in the "Section Data" segment in binary. 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.

Table 1: 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 the binary code with line numbers of source code for debugging.
Symbol Table	Describes information for all symbols used in the COFF file.
String Table	Stores long symbol names.

The first step of the BACCI process is to decompose the binary files of the components into code sections of exported functions. There are two necessary sub-steps: (1a) decomposing the binary

[1] MSDN Library - Visual Studio .NET 2003

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 often necessary because the output contains trivial information, such as timestamps and file pointers, which are useless for change identification. The output should be formatted conveniently for differencing tools to identify changes in exported functions between releases. 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)[1]. The output generated by DUMPBIN presents all the information about the 32-bit COFF binary files in a comprehensible manner.

Generally, the second sub-step cannot be done via existing tools. Therefore, we have created a flexible tool to accomplish Step 1. We call this tool the Decomposer and Trivial Information Zapper (D-TIZ)[2]. It can perform the decomposition as well as remove trivial information. Currently D-TIZ can only be used with library files, but will be extended to handle all the COTS types, as discussed in future work.

The second step of the 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.

This change information can be fed into the third step in Figure 1 in which regression test cases are selected. White-box regression test cases which call the changed functions can be selected to be re-run.

4. FEASIBILITY STUDY

In this section, we present a feasibility study in which we applied the BACCI process to four incremental releases of a 67 thousand lines of code (KLOC) internal ABB product written in C. Henceforth, these releases will be referred to as Release 1 through Release 4, respectively. We chose this product because multiple releases and source code are available; source code was needed to verify the accuracy of the analysis post hoc. Our goal was to study the effectiveness of the BACCI process to identify affected code changes to the functions that can be called in the library file in Release 2, Release 3, and Release 4. The feasibility study is an instance of the process introduced in Section 3 in which the ability to identify changes in library (.lib) files was studied. The analyzer (the first author of this paper) did not have access to the source code and did not know the changes in source code. The results of the analysis were verified by the second author, who was not involved in the detailed change identification analysis. In this case study, the only artifacts in each release of the product which were available to the analyzer are two library files in the delivery package; neither the header files nor the documentation were analyzed. However, one of the two libraries was the Unicode version of the other library. Only the one non-Unicode version was analyzed in this study due to time limitations.

The rest of this sub-section is organized as follows. Section 4.1 describes the detailed process for analyzing library file, and Section 4.2 presents the results of the analysis. The limitations of this process are discussed in Section 4.3.

4.1. BACCI processing of library files

The feasibility 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. The basic format of a library file is shown in Table 2.

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 corresponding header.

Table 2: Overall Structure of a Library File

Segment	Description
Signature	An identifier whose value is always "!<arch>\n"
First Section	Includes all the names of symbols in the library and the absolute offsets of the object sections
Second Section	An index table of the content of the first section
Longname Section	Contains a string table including all object file names longer than eight bytes
Object Section 1 ~ n	Raw data of object files

During the first step of the BACCI process in Section 3, the binary files of components were decomposed into code sections of exported functions. Each library file in each of the four 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 second step of this change identification process is to compare the exported functions among the four 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 experiment was conducted on an IBM T42 laptop with one Intel® Pentium® M 1.8GHz processor and one gigabyte RAM. It took eight seconds in total for DUMPBIN and D-TIZ to complete the first step of the process. For the second step, Araxis Merge spent about five seconds on each comparison and about one minute generating the full differencing report.

4.2. Results

In this feasibility study, the proposed BACCI process was applied three times between successive released versions of the internal

[2] http://www4.ncsu.edu/~jzheng4/D-TIZ/index.htm
[3] http://www.araxis.com
[4] http://www.foldermatch.com
[5] http://www.scootersoftware.com
[6] http://winmerge.sourceforge.net

ABB product. The result is shown in Table 3. 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 3: 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 first analysis was conducted between Release 1 and Release 2. This analysis 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 identified function was changed between Releases 1 and 2.

The second analysis was conducted between Release 2 and Release 3. This BACCI analysis determined 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.

The final analysis was conducted between Release 3 and Release 4. This analysis 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 above case study shows the effectiveness of this technique at determining change within third party components when no void function pointers or other static addresses are used. Where these pointers are present, they lead only to false positives and no false negatives are identified.

4.3. Limitations

In addition to the false positives discussed in Section 4.2, the BACCI process has one important limitation. Decompiling must not be against the licensing agreement of the COTS component. 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.

5. CONCLUSION AND FUTURE WORK

In this paper, we proposed the BACCI process for identifying affected areas for COTS components when source code for the components is not available. Several versions of a set of library files were examined as a feasibility study to verify the potential accuracy of this process. The results showed that BACCI can be effective at determining changes in third party software.

Accuracy, generalization, integration, and automation are four main goals of work in the future. First, in depth, we need to reduce or even eliminate the false positives. On the other hand, additional breadth is required to identify changes between releases for all the COTS types. We plan to analyze more components in 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 customer programs act as containers for third party controls. Additionally, the proposed process for change identification should be integrated with code-based regression test selection techniques to achieve the ultimate goal: effective regression testing without code. Finally, the whole process should be automated into one tool to save both time and resources.

6. REFERENCES

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

[2] 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.

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

[4] 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.

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

[6] 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.

[7] 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.

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

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

[10] 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.

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