Pallino: Automation to Support Regression Test Selection for COTS-based Applications

Jiang Zheng
North Carolina State University
Room 3231, Engineering Building II
Raleigh, NC 27695-8206
jzheng4@ncsu.edu

Laurie Williams
North Carolina State University
Room 3272, Engineering Building II
Raleigh, NC 27695-8206
williams@csc.ncsu.edu

Brian Robinson
ABB Inc., US Corporate Research
940 Main Campus Drive
Raleigh, NC 27606
brian.p.robinson@us.abb.com

ABSTRACT
Software products are often built from commercial-off-the-shelf (COTS) components. When new releases of these components are made available for integration and testing, source code is usually not provided by the vendors. Various regression test selection techniques have been developed and have been shown to be cost effective. However, the majority of these test selection techniques rely on source code for change identification and impact analysis. In our research, we have evolved a regression test selection (RTS) process called Integrated - Black-box Approach for Component Change Identification (I-BACCI) for COTS-based applications. I-BACCI reduces the test suite based upon changes in the binary code of the COTS component using the firewall regression test selection method. In this paper, we present the Pallino tool. Pallino statically analyzes binary code to identify the code change and the impact of these changes. Based on the output of Pallino and the original test suit, testers can determine the regression test cases needed that cover the application glue code which is affected by the changed areas in the new version of the COTS component. Three case studies, examining a total of fifteen component releases, were conducted on ABB internal products. With the help of Pallino, RTS via the I-BACCI process can be completed in about one to two person hours for each release of the case studies. The total size of application and component for each release is about 340~830 KLOC. Pallino is extensible and can be modified to support other RTS methods for COTS components. Currently, Pallino works on components in Common Object File Format or Portable Executable formats.

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

General Terms
Algorithms, Reliability, Verification.

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

1. INTRODUCTION
Companies increasingly incorporate a variety of commercial-off-the-shelf (COTS) components in their products. Upon receiving a new release of a COTS component, users of the component often conduct regression testing to determine if a new version of a component will cause problems with their existing software and/or hardware system. Regression testing involves selective re-testing of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [12]. Rerunning all of the test cases for the application can be prohibitively expensive in both time and resources [10]. Therefore, a variety of regression test selection (RTS) techniques have been developed (for example, [5, 10, 22]) to reduce the number of tests that need to be executed without significant risk of excluding important failure-revealing test cases. However, most existing RTS techniques rely on source code, and therefore are not suitable when source code is not available for analysis, such as when an application incorporates COTS components.

In our research, we have evolved a RTS process called the *Integrated - Black-box Approach for Component Change Identification (I-BACCI[1])* for COTS-based applications. I-BACCI uses static change identification of binary code and the firewall analysis RTS technique [31-33]. In this paper, we present the new comprehensive automation for the I-BACCI process, henceforth called *Pallino*[2]. Pallino performs binary change identification and impact analysis and can be modified to support other RTS methods for COTS components. The input artifacts to Pallino are the binary code of the components (old and new versions), which is generally available to users of COTS components. Pallino outputs a list of affected exported component functions. *Affected exported component functions* are functions within the COTS component that interface with the application, and are either changed or are in the call chain of other changed functions.

Based on the list of affected exported component functions and the original test suite, testers can identify glue code functions that call affected exported component functions. Then the subset of the regression test cases that cover the glue code which is affected by the changed areas in the new COTS components can be identified. *Glue code* is application code that interfaces with the COTS components, integrating the component with the application. The final output of the I-BACCI process is a reduced suite of regression test cases. Currently Pallino works on components in Common Object File Format (COFF) or Portable Executable (PE) formats written in C/C++. COFF libraries usually have the extension `.lib`. Typical PE files have the extensions `.exe`, `.dll`, `.ocx`, `.sys`, `.cpl` and `.scr`. This paper also reports the results of applying Pallino to identify a reduced test suite for three industrial case studies.

---

[1] We pronounce BACCI the way the *bocce* is pronounced when referring to the Italian ball game: [bah-chee].

[2] http://www4.ncsu.edu/~jzheng4/pallino.htm
A *pallino* is the small ball used in the bocce ball game.

The rest of this paper is organized as follows. Section 2 outlines the background and related work. Section 3 introduces the I-BACCI process. Section 4 illustrates two examples of how Pallino works. Section 5 and Section 6 describe the architecture and algorithms of Pallino, respectively. Section 7 illustrates how to use Pallino. Section 8 presents the results of case studies. Finally, Section 9 presents the conclusions and future work.

2. BACKGROUND AND RELATED WORK

This section provides prior work in testing of software components, regression test selection, binary code analysis and its application for change identification and impact analysis and RTS, I-BACCI process, and legal issues.

2.1 Testing of Software Components

Poor testability, due to the lack of access to the component's source code and other artifacts, is one of the challenges in user-oriented component testing [8, 9, 28]. When only binary code is available, binary reverse engineering is a technically-feasible approach for automatically deriving information that can inform the RTS. The derived information can be a program structure of a component from its binary code, such as call graphs [18].

Harrold et al. [11] presented techniques that use component metadata for RTS 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 [11]. 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 [11]. However, the component vendors may not provide the metadata information. In our research, we focus on using information that is typically available to a COTS component user.

Recently Mariani et al. addressed both the problem of quickly identifying components that are syntactically compatible with the interface specifications, but badly integrate in target systems, and the problem of automatically generating regression test suites [17]. Their technique relies on automatic inference of IO and interaction models. IO models are boolean expressions over the values exchanged during the computation, describing properties of data values exchanged between components. Interaction models describe sequences of invocations by finite state machines labeled with method invocations. [17] As a dynamic analysis technique that automatically synthesizes behavioral models from execution traces, their technique requires runtime setup and reliable sets of test suites. Our approach based on static binary code analysis which will be discussed in Section 2.3.

2.2 RTS and Firewall Analysis

The purpose of RTS techniques is to reduce the high cost of retest-all regression testing by selecting a subset of possible test cases [10]. A variety of RTS techniques (for example, [5, 10, 22]) have been proposed, such as methods based upon path analysis techniques or dataflow techniques. Leung and White [16, 29] developed firewall analysis for regression testing with integration test cases in the presence of small changes in functionally-designed software. Firewall analysis restricts regression testing to potentially-affected system elements directly dependent upon changed system elements [29, 30]. 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]. Via empirical studies of industrial real-time systems, firewall analysis was shown to be effective [30]. The firewall analysis allowed an average savings of 36% of the testing time and 42% of the tests run. No additional errors were detected by the customer on the studied software releases that were due to the changes in these releases to date. [30] Our I-BACCI approach extends the traditional concept and scope of firewall analysis for use with binary code.

2.3 Binary Code Analysis

Binary code analysis (BCA) approaches and tools have been developed and utilized in many software development-related activities, including program comprehension, software maintenance and software security, even by software developers that have access to the source code [24]. For example, malicious code or patterns in executable can be detected via BCA to enhance software security [4].

Balakrishnan et al. presented the "What You See Is Not What You eXecute" (WYSINWYX) phenomenon: There can be a mismatch between what a programmer intends and what is actually executed by the processor, e.g., presence or absence of procedure calls by the optimizing compiler [3]. Therefore, analyses performed on source code can fail to detect certain bugs and vulnerabilities. Also, analyzing executables has other advantages, such as, revealing more accurate information about the behaviors that might occur during execution, because an executable contains the actual instructions that will be executed [3].

Additionally, BCA can be dynamic or static. Dynamic BCA monitors the execution of programs. In contrast, static BCA provides a way to obtain information about the possible states that a program reaches during execution without actually running the program on specific inputs. Static techniques explore the program's behavior for all possible inputs and all possible states that the program can reach. [3] Srivastava et al. discussed the advantages of comparing software at the binary level rather than the source code level: binary comparison is (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 [24]. Our tool utilizes static BCA to identify changes and change impact within the COTS components where source code is not available.

2.4 BCA for Change Identification, Impact Analysis and RTS

A key step in choosing regression tests is applying impact analysis [19] to identify changes between the new release and the previously-tested version with the same source code base. However, similar to RTS, most change identification approaches utilize the source code of the old and modified programs [2, 14, 21, 22, 25, 26]. These approaches are not suitable for component testing when source code is not available.

Srivastava and Thiagarajan at Microsoft developed Echelon [24], a test prioritization system. Echelon is used to prioritize tests based upon changes between two versions identified by a binary code comparison. 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). Wang et al. [27] 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 [23] to create an intermediate representation of x86 binaries. Vulcan separates code from data and identifying program symbols. The tool enables good matching even with shifted addresses, different register allocations, and small program modifications [27]. BMAT underlies Echelon [24] to match blocks in the two binaries.

However, Echelon and BMAT are large proprietary Microsoft internal products with a significant infrastructure and an underlying binary code manipulation engine, and therefore cannot be used by the community at large. Also, Echelon prioritizes, but does not eliminate tests [24]. Our goal is to provide information about which test cases are not necessary to rerun.

3. I-BACCI

We have evolved the I-BACCI process for RTS for COTS-based applications [31-33]. For previous case studies of I-BACCI, the process was supported by three separate tools, D-TIZ, TID-BITZ, and CAAFI [31-33] which required the expertise and manual intervention of the first author of this paper to understand the form of the input and outputs from/for each tool to obtain the desired output. Pallino integrates these three tools and provides a user interface to enable it to be used and modified by others to support RTS of COTS components in the general case.

The steps of I-BACCI are shown in Figure 1. The I-BACCI process is an integration of (1) a static binary code change identification process; and (2) firewall analysis RTS technique. Our uniqueness is the combination of the two parts to identify and localize change with the goal of reducing the regression test suite.

The I-BACCI process has been evolved to Version 4 through the application of the process on both COFF and PE components written in C/C++. The I-BACCI Version 4 involves seven steps. The first four steps are completed via a BCA process (in dash-dotted line frame) using the Pallino tool. The remaining three RTS steps are completed via firewall analysis (in dashed line frame). The input artifacts to the process are the binary code of the COTS components (old and new versions); the source code and test suite of the development application; and all test cases which are mapped to the glue code functions they cover. These input artifacts are generally available to users of COTS components. The output of the I-BACCI process is a reduced suite of regression test cases necessary to exercise the changed areas in the new COTS components.

Lawyer and software engineering professor Dr. Cem Kaner deems that we are reverse engineering [13]. The definition of "reverse engineering" he provided is: "to study or analyze (a device, as a microchip for computers) in order to learn details of design, construction, and operation, perhaps to produce a copy or an improved version." [1, p. 1326] His opinion is: if a license indicates "no reverse engineering" then use of Pallino could constitute a breach of contract. However, many software components may not have this clause, for example, open source and free software products, and many other scientific/educational products distributed in binary. The interaction of patent law and mass market license terms, as it affects interoperability, is being actively debated within the legal profession [15]. The intent of I-BACCI is to enable COTS purchases to gather change information to inform their RTS. Kaner advises that purchasers of COT software should contact their vendors and request waivers that allow them to reverse engineer COTS components for the purpose of managing their maintenance costs. He furthers that vendors of these components would serve their customers well by revising their licenses to specifically permit this kind of analysis.

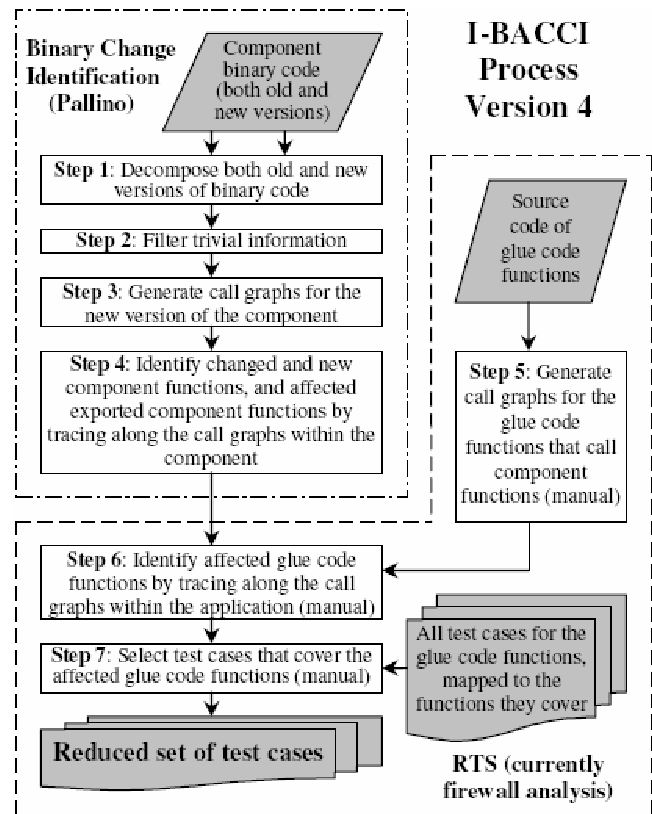

Figure 1: I-BACCI Version 4

4. PALLINO

Pallino utilizes static binary code analysis to compare two versions of a component, identifying semantic changes and their impact within the new version of the component. Pallino first decomposes the binary files of the component, i.e., breaking up the binary code down into constituent elements, such as code sections and relocation tables. Prior to distribution, component source code is compiled into binary code, such as `.lib` or `.dll` files. Information on the data structure, functions, and function calling relationships of the source code is stored in 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. Often the first step can be accomplished by parsing tools available for the specified language/architecture. Pallino uses Microsoft COFF Binary File Dumper (DUMPBIN)[3] to decompose COFF and PE binary files.

The second step, filtering trivial information, is frequently necessary because the output from the first step may contain trivial information such as timestamps and file pointers that are irrelevant to the change identification. Pallino removes the trivial information and extracts the raw code section of each function/data, and function/data calling relationships for the new version of the component.

In the third step, Pallino identifies true changes in the raw binary code of functions and data by removing the false positives caused by differences due to trivial changes, such as shifted addresses and register reallocations. Pallino also represents, generates and analyzes call graphs for the new version of the component.

In the last step, Pallino identifies changed and new added component functions according to the results of prior steps, and then identifies affected exported component functions by tracing along the call graphs within the component using directed graph theory algorithms. Analysis starts from each component function identified as changed, and that change is propagated along the call graphs until the exported functions are reached.

The remainder of this section presents two examples to illustrate how affected exported component functions are identified from binary code. We will discuss the examples for COFF component and PE component in Section 4.1 and Section 4.2, respectively.

4.1 COFF Component

Windows NT® uses a special format for the executable (image) files and object files. The format used in these files is referred to as COFF files[3]. Object files created from C or C++ programs using many compilers conform to COFF, including the Visual C++ and the GNU Compiler Collection (GCC).

In the example to follow, Release 4 and Release 5 of an ABB component are compared. We refer to these as the "old" and the "new" releases. The input to Pallino is `.lib` binary files of the two releases. Binary code fragments in the two releases are shown in Figure 2. At first glance, we can not see any relationship between the two binary code fragments because both the binary code and the address ranges are different. We use the DUMPBIN tool to translate the binary files into plain text. The counterparts in the output of DUMPBIN for the two binary code fragments are shown in Figure 3. The sections can be located and identified by function signatures, e.g., `function1` in this example. Directive information, such as size of raw data and function signature, is shown in the "`SECTION HEADER`" subsection. The "`RAW DATA`" subsection displays the binary code that represents `function1` for each release, i.e. the code in boldface in Figure 2. The "`RELOCATIONS`" subsection lists which other functions and data are called by `function1`. Non-trivial difference for `function1` between the two releases is underscored in Figure 2 and 3. Thirty six bytes of code (three lines of source code), with three functions and data calls were deleted in the new release.

```
00003830: ......                                   /* Old Release */
00003840: 00 07 00 51 56 8B 74 24 0C 57 C6 44 24 0B 01 83
00003850: 7E 04 01 7D 07 5F 83 C8 FF 5E 59 C3 56 E8 00 00
00003860: 00 00 8B F8 83 C4 04 85 FF 74 15 6A FF 68 00 00
00003870: 00 00 E8 00 00 00 00 83 C4 08 8B C7 5F 5E 59 C3
00003880: 56 E8 00 00 00 00 8B F8 83 C4 04 85 FF 74 18 68
00003890: 80 00 00 00 68 00 00 00 00 E8 00 00 00 00 83 C4
000038a0: 08 8B C7 5F 5E 59 C3 56 E8 00 00 00 00 8B F8 83
000038b0: C4 04 85 FF 74 15 6A FF 68 00 00 00 00 E8 00 00
000038c0: 00 00 83 C4 08 8B C7 5F 5E 59 C3 8B 4E 04 8D 44
000038d0: 24 0B 6A 01 50 6A 04 68 FF FF 00 00 51 FF 15 00
000038e0: 00 00 00 85 C0 74 1B 6A 04 68 00 00 00 00 E8 00
000038f0: 00 00 00 6A 04 68 00 00 00 00 E8 00 00 00 00 83
00003900: C4 10 56 E8 00 00 00 00 83 C4 04 5F 5E 59 C3 90
00003910: 90 90 90 1B 00 00 00 96 00 00 00 14 00 2B 00 00
00003920: ......
00003a60: ......                                   /* New Release */
00003a70: 00 01 00 00 14 00 00 00 00 A5 00 00 00 07 00 51
00003a80: 56 8B 74 24 0C 57 C6 44 24 0B 01 83 7E 04 01 7D
00003a90: 07 5F 83 C8 FF 5E 59 C3 56 E8 00 00 00 00 8B F8
00003aa0: 83 C4 04 85 FF 74 15 6A FF 68 00 00 00 00 E8 00
00003ab0: 00 00 00 83 C4 08 8B C7 5F 5E 59 C3 56 E8 00 00
00003ac0: 00 00 8B F8 83 C4 04 85 FF 74 18 68 80 00 00 00
00003ad0: 68 00 00 00 00 E8 00 00 00 00 83 C4 08 8B C7 5F
00003ae0: 5E 59 C3 8B 4E 04 8D 44 24 0B 6A 01 50 6A 04 68
00003af0: FF FF 00 00 51 FF 15 00 00 00 00 85 C0 74 1B 6A
00003b00: 04 68 00 00 00 00 E8 00 00 00 00 6A 04 68 00 00
00003b10: 00 00 E8 00 00 00 00 83 C4 10 56 E8 00 00 00 00
00003b20: 83 C4 04 5F 5E 59 C3 90 90 90 90 90 90 90 90 1B
00003b30: ......
```

Figure 2: Binary code fragments

```
SECTION HEADER #31                                /* Old Release */
   .text name                    // section name
         ......                  // directive information
         Communal; sym= _function1          // signature
         ......                  // directive information
RAW DATA #31
00000000: 51 56 8B 74 24 0C 57 C6 44 24 0B 01 83 7E 04 01
00000010: ......       // more binary code for function1
RELOCATIONS #31
Offset   Type  Index Symbol Name
-------- ----- ----- -----------
0000001B REL32    96 _function2
0000002B DIR32    B8 string_data1
00000030 REL32   152 _function3
0000003F REL32    9D _function4
00000052 DIR32    B5 string_data2
00000057 REL32   152 _function3
00000066 REL32    A4 _function5
00000076 DIR32    B2 string_data3
0000007B REL32   152 _function3
0000009C DIR32    6C _function6
000000A7 DIR32    AF string_data4
000000AC REL32   152 _function3
000000B3 DIR32    AC string_data5
000000B8 REL32   152 _function3
000000C1 REL32    BD _function7
SECTION HEADER #31                                /* New Release */
   .text name                    // section name
         ......                  // directive information
         Communal; sym= _function1          // signature
         ......                  // directive information
RAW DATA #31
00000000: 51 56 8B 74 24 0C 57 C6 44 24 0B 01 83 7E 04 01
00000010: ......       // more binary code for function1
RELOCATIONS #31
Offset   Type  Index Symbol Name
-------- ----- ----- -----------
0000001B REL32    97 _function2
0000002B DIR32    B6 string_data1
00000030 REL32   14F _function3
0000003F REL32    9E _function4
00000052 DIR32    B3 string_data2
00000057 REL32   14F _function3
00000078 DIR32     C _function6
00000083 DIR32    B0 string_data4
00000088 REL32   14F _function3
0000008F DIR32    AD string_data5
00000094 REL32   14F _function3
0000009D REL32    BB _function7
```

Figure 3: DUMPBIN output

[3] MSDN Library - Visual Studio .NET 2003

Using the calling relationship information in the "RELOCATIONS" subsection, Pallino then generates call graphs and identifies the affected exported component functions in the new release. Figure 4 shows how `function1` (in black) affects glue code. Although three exported component functions are affected by `function1`, only one glue code function (`Glue_code_function1`) calls one of the affected exported component functions. Therefore, RTS will only need to select test cases for `Glue_code_function1` from the initial test suite.

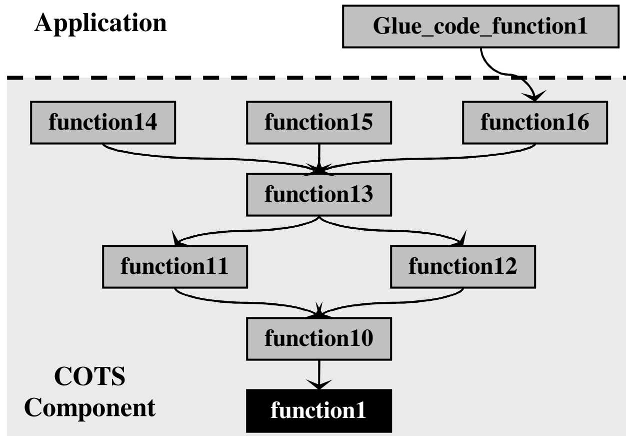

Figure 4: Call graph: how component change affects glue code

4.2 PE Component

Many executables, such as `.exe` files, Object Linking and Embedding Control Extension (OCX) controls, and Control Panel applets (`.cpl` files) are in PE format. When loaded into main memory by the Windows loader, PE files can be mapped directly into memory, such that the data structures on disk are the same as those Windows uses at runtime. If one knows how to find something in a PE file, he can almost certainly find the same information when the file is loaded in memory. [20] This aspect facilitates static BCA. However, some characteristics of the PE format make the change identification and call graph generation more complex than analyzing COFF files. For example, only the names of exported component functions can be obtained in the binary code, such that functions have to be mapped between two releases after generating the call graphs for exported component functions. Also, the DUMPBIN output for PE files contains only one *.text* section where the raw code fragments for all the functions are consecutively arranged. Relocation information is stored in the only *.reloc* section, as shown in Figure 5. Whereas the information for each function, such as raw code fragment and relocation table, locates in separated sections in the DUMPBIN output for COFF files. Pallino needs to parse these sections and integrate information to achieve the goal of change identification and impact analysis.

This example illustrates how the exported function `foo` is affected by changed data. Information related to `foo` in the DUMPBIN output for the two `.dll` files are shown in Figure 5. The *Relative Virtual Address (RVA)* of the raw code segment of `foo` can be found in the exports table in the *.rdata* section, e.g., `00003BC0` in the old release and `000024D0` in the new release. Therefore, the *start virtual addresses* of the raw code segment of `foo` can be calculated by adding the image base address (`0x10000000` in this example) to the RVA, i.e., `0x10003BC0` in the old release. The binary code that represents `foo` for each release is in boldface in the "`RAW DATA #1`" subsections. Differences for the raw code segment of `foo` between the two releases are gray highlighted in Figure 5.

```
SECTION HEADER #1                                  /* Old Release */
   .text name                    // code section
         ......                  // directive information
RAW DATA #1
......
10003BC0: 8B 44 24 04 85 C0 74 13 8B 4C 24 08 51 68 34 82
10003BD0: 01 10 50 E8 88 6D 00 00 83 C4 0C 8B 44 24 0C 85
10003BE0: C0 74 13 8B 54 24 10 52 68 2C 82 01 10 50 E8 6D
10003BF0: 6D 00 00 83 C4 0C 8B 44 24 14 85 C0 74 13 8B 4C
10003C00: 24 18 51 68 24 82 01 10 50 E8 52 6D 00 00 83 C4
10003C10: 0C B8 01 00 00 00 C2 18 00 90 90 90 90 90 90 90
......
SECTION HEADER #2
  .rdata name                     // read only data section
         ......                   // directive information
  ordinal hint RVA      name      // exports table
        ......
        6    A 00003BC0 foo
        ......
SECTION HEADER #3
   .data name                     // read/write data section
         ......                   // directive information
RAW DATA #3
......
10018220: 00 00 00 00 41 53 43 49 49 00 00 00 31 2E 31 2E
10018230: 38 00 00 00 45 42 50 41 5F 4C 49 43 45 4E 53 49
......
SECTION HEADER #4
  .reloc name                     // relocation section
         ......                   // directive information
BASE RELOCATIONS #4
     ......
    3000 RVA
     BCE  HIGHLOW      10018234            // call data3
     BE9  HIGHLOW      1001822C            // call data2
     C04  HIGHLOW      10018224            // call data1
     ......
SECTION HEADER #1                                  /* New Release */
   .text name                    // code section
         ......                  // directive information
RAW DATA #1
......
100024D0: 8B 44 24 04 85 C0 74 13 8B 4C 24 08 51 68 08 7B
100024E0: 01 10 50 E8 78 84 00 00 83 C4 0C 8B 44 24 0C 85
100024F0: C0 74 13 8B 54 24 10 52 68 00 7B 01 10 50 E8 5D
10002500: 84 00 00 83 C4 0C 8B 44 24 14 85 C0 74 13 8B 4C
10002510: 24 18 51 68 F8 7A 01 10 50 E8 42 84 00 00 83 C4
10002520: 0C B8 01 00 00 00 C2 18 00 90 90 90 90 90 90 90
......
SECTION HEADER #2
  .rdata name                     // read only data section
         ......                   // directive information
  ordinal hint RVA      name      // exports table
        ......
        6    A 000024D0 foo
        ......
SECTION HEADER #3
   .data name                     // read/write data section
         ......                   // directive information
RAW DATA #3
......
10017AF0: 25 30 38 6C 78 00 00 00 41 53 43 49 49 00 00 00
10017B00: 31 2E 31 2E 39 00 00 00 45 42 50 41 5F 4C 49 43
......
SECTION HEADER #4
  .reloc name                     // relocation section
         ......                   // directive information
BASE RELOCATIONS #4
     ......
    2000 RVA
     4DE  HIGHLOW      10017B08            // call data3
     4F9  HIGHLOW      10017B00            // call data2
     514  HIGHLOW      10017AF8            // call data1
     ......
```

Figure 5: DUMPBIN output

They are all trivial shifted addresses to be ignored in semantic differencing. However, according to the information in the relocation sections, `foo` calls `data2`, which changes from `0x312E312E38` (ASCII string "`1.1.8`") to `0x312E312E39` (ASCII string "`1.1.9`"), as shown in underscored code in the "`RAW DATA #3`" subsections. The one byte change exactly reflected the modification in source code: the value of macro definition `VERSION` changed from "`1.1.8`" to "`1.1.9`" in the new release. Pallino then generates call graphs and identifies how changed `data2` affects `foo`, as shown in Figure 6.

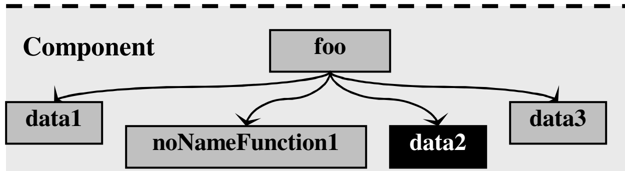

Figure 6: Call graph: How changed data affects exported component function

5. ARCHITECTURE

The overall architecture of Pallino conforms to the model-view-controller (MVC) model, as shown in Figure 7. The solid lines represent direct associations, and the dashed lines represent indirect associations. An MVC architecture separates data (model) and user interface (view) concerns, so that changes to the user interface do not affect the data handling, and that the data can be reorganized without changing the user interface.

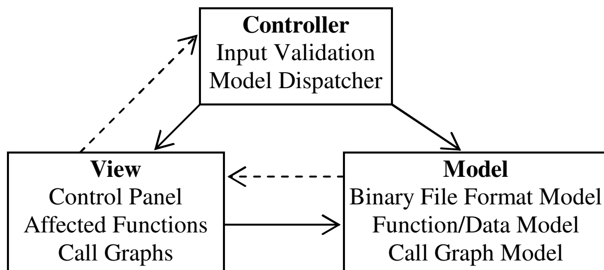

Figure 7: Overall architecture

According to the object-oriented software design principle of "program to an interface, not an implementation"[7], a generic interface `BinaryFileFormatModel` was created to represent the abstract concept of the binary file format model. The concrete types of binary file format model, including `COFFFormat` and `PEFormat`, implement the generic interface and represent the data structure of COFF and PE binary file formats, respectively. The client code accesses objects of a concrete binary file format model only through their abstract interface. This pattern allows for new derived types of binary file format model to be introduced with no change to the code that uses the base object, increasing the extensibility of the tool. Functionality that processes other binary file format, such as Executable and Linking Format (ELF)[4], can be easily added into the system without affecting many other modules, as shown in Figure 8.

[4] A common standard file format for executables, object code, shared libraries, and core dumps, which was chosen as the standard binary file format for Unix and Unix-like systems.

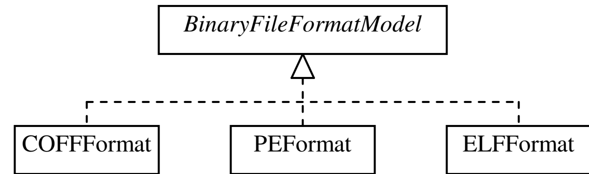

Figure 8: Binary File Format Model

Additionally, we adapted the data structures to the functionality of change identification and impact analysis. Not all information in the binary file formats are needed to be described in the models used in Pallino, for example, time date stamps and number of symbols. The data structure of `PEFormat` is shown in Figure 9.

```
Class PEFormat {
  PEFileHeader      fileHeader;    //File header
  PEOptionalHeader optHeader;      //Optional header
  PESectionHeader  textHeader;     //.text section header
  PESectionHeader  rdataHeader;    //.rdata section header
  String           rdataRaw;       //.rdata section data
  PESectionHeader  dataHeader;     //.data section header
  String           dataRaw;        //.data section data
  PESectionHeader  idataHeader;    //.idata section header
  String           idataRaw;       //.idata section dara
  PESectionHeader  relocHeader;    //.reloc section header
  PEExportTable    exportsTable;   //Exports table
  PEImportTable    importsTable;   //Imports table
}
```

Figure 9: PE format data structure

In the function/data model, we abstract function and data as the same class (`PEFunctionData`) with the following main attributes: signature, start virtual addresses, end virtual address, raw binary code, and relocation list. Functions without explicit signatures (e.g. non-exported functions in PE files) and all data use start virtual addresses as their signatures.

The view module of the architecture includes the control panel, and result representation and displaying, which will be described in Section 7 with the illustration of use.

The controller module is responsible for processing and responding the input event from the user interface. First, the controller validates the input and recognizes the type of the input binary file by the magic number to decide to which algorithm it will pass the request. A *magic number* is a pre-defined constant, typically located at the first few bytes of a binary file, used to identify the file type. For example, PE files start with the ASCII string '`MZ`' (`0x4D5A`), and the magic number of a COFF file is the ASCII string "`!<arch>\n`" (`0x213C617263683E0A`). The controller then executes the algorithm and accesses the corresponding model. Finally, the results are produced and returned to the view module and user interface.

6. ALGORITHMS

In this section, the algorithms that were developed for components in COFF and PE formats are described, respectively.

6.1 Algorithms for COFF components

At first, DUMPBIN was invoked to convert the binary library code into plain text. An example of the DUMPBIN output is shown in the Figure 10. Function names, binary code representation of the functions, and relocation tables are all clearly described in the output text of DUMPBIN. The algorithm 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 next step to perform differencing. The relocation table set is utilized to generate and analyze call graphs of the components in later steps.

```
int ClassA::functionA(int s) {          /* Source Code */
   return state==s;
}
SECTION HEADER #78                      /* Old Release */
         ......
         Communal;  sym=  "public:  virtual  int  __thiscall
ClassA::functionA(int)"
         ......
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
SECTION HEADER #79                      /* New Release */
         ......
         Communal;  sym=  "public:  virtual  int  __thiscall
ClassA::FunctionA(int)"
         ......
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
```

Figure 10: Source code and DUMPBIN output for functionA in ClassA

In the next step, it is necessary to reduce the number of false positive changes identified due to trivial changes, such as shifted addresses and register reallocations. A large number of false positives were observed in the initial case study of the I-BACCI Version 1 [31], 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 the Figure 10, 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[5]. 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 one of the header files included in the source file of the new release. 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. The binary code like `8B89A0060000` is called 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[6]. The algorithm did reduce the false positive rate to less than 8% in the case studies [33].

The algorithm then builds and analyzes the call graphs of components of COFF type automatically using the relocation table set generated in the first step, and changed functions identified in the second step. Due to the large number of functions in the components, it is time-consuming to identify affected functions given changed functions in the components according to the calling relationships produced in Step 3 and 4 of the I-BACCI Version 4 process. The relocation table set of a component is converted into an adjacency-matrix [6] to represent call graphs of the functions in the component. For each changed function, the algorithm then backtracks the call graphs to identify all functions that directly or indirectly call the changed function.

The outputs of Pallino for analyzing COFF components include: (1) the call graph of each exported component function; (2) a differencing report on the two releases; (3) a list of all affected exported component functions in the new release.

6.2 Algorithms for PE components

The algorithm examines the binaries from coarse to fine granularity step by step. First, the tool invokes DUMPBIN to translate the illegible binary library files into readable plain text files. This file-level granularity step assumes that file names do not change between releases. Then a file reader automatically scans the DUMPBIN output and loads useful information, such as instructive information in file header, section headers, exports table and imports table, into the predefined data structure `PEFormat` which is constructed according to the PE file format specification. File and section information is ready to facilitate future lookup after this section-level step.

The next finer granularity is in function/data-level. Binary code of functions and data are stored consecutively in *.text* section and data sections (*.rdata*, *.data*, *.idata*, etc.), respectively. However, only names of exported component functions are available. Other functions and all data have to be labeled by their start virtual addresses. The data structure `PEFunctionData`, as discussed in Section 5, is used to represent the functions and data. The relocation table is read from the *.reloc* section and then converted into a `Hashtable` called *relocation index*. For each key-value pair in the relocation index, the key is a *calling virtual address* where the control flow jumps to another function or data, and the start virtual address of the function or data being called (a.k.a. *target virtual address*) is stored as the value. Function calling virtual address and target virtual address can also be calculated according to the position of each call instruction and the address offset following each call instruction, respectively. Because only binary code is available instead of assembly code, the tool searches opcode `E8` and `E9` which represent "call near" in the Intel instruction set[6] to locate the position of each function call. After finding all functions and data start virtual addresses, an array in `PEFunctionData` type is constructed and the raw code of the *.text* and data sections is decomposed into separate functions or data.

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

Further instruction-level comparisons can be conducted after the function/data level if the full code representations of an exported component function in two releases are still different. For example, false positives may be caused by register allocation changes from build to build. After all of the above steps, a report on differencing

---

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

[6] http://www4.ncsu.edu/~jzheng4/895/tools.htm

of exported component functions will be generated. We can use this report to identify affected application code and then select proper regression test cases.

The outputs of Pallino for analyzing PE components include: (1) the call graph of each exported component function; (2) a full binary code representation of each exported component function, including all sub-functions and data that might be called by that exported function; (3) a differencing report on the two releases; (4) a list of all affected exported component functions in the new release.

7. ILLUSTRATION OF USE

Although developed in Java, Pallino is transformed to a Windows executable file (.exe) by exe4j[7] to facilitate the use in the Windows operating environment. An illustrative screen shot of the main console of Pallino is shown in Figure 11.

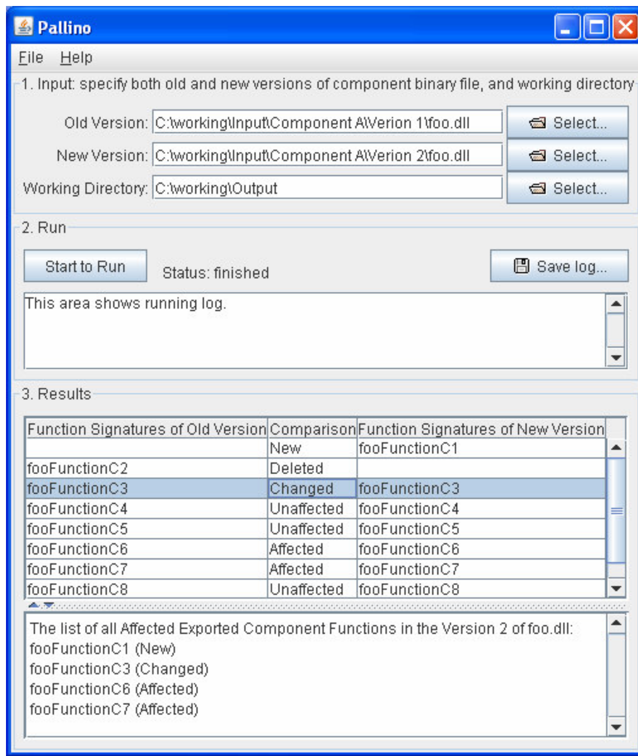

Figure 11: Pallino screen shot

There are three panes on the main console: input pane, run pane, and results pane. The user of Pallino first specifies the binary files of both old and new versions of a component, and a working directory in the input pane. The specified working directory is for the purpose of saving results and running log. Once the input is specified and the "Start to Run" button in the run pane is clicked, Pallino accepts and validates the input, executes the corresponding algorithms according to the file format, and upon completion, refreshes the results in the results pane.

The exported component functions for both versions of the component are shown in a table in the results pane, matched by the function signatures, i.e., functions with the same signatures are shown in the same row. The user can clearly see which functions are new added, changed, affected, or unaffected in the new version of the component in the middle column of the result table. Further explanation, including a list of all affected exported component functions for the new version of the component, is shown in a text area in the results pane. The results are also saved into files for the RTS analysis of the I-BACCI process. The running log can also be saved to a file by clicking on the "Save log..." button.

[7] http://www.ej-technologies.com/products/exe4j/overview.html

8. CASE STUDIES

The subjects examined in our case studies are summarized in Table 1. These software combinations were chosen for these case studies because (1) the numbers of test cases for each function of the applications were available; (2) multiple releases of the components were available; (3) the high cost of executing the retest-all strategy demonstrates the potential value of achieving regression test reductions.

The first author was the analyzer and the third author was the verifier. The analyzer conducted the first six steps of the I-BACCI Version 4. 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 verifier 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 verifier also confirmed the efficacy of the RTS process by examining the failure records of retest-all black-box testing.

Table 1. Summary of case study subjects

Case	Application	Component	Releases
1	757 KLOC	one 67 KLOC .lib file in C	1 ~ 6
2	40 KLOC	eight .lib files in C, totally 300 KLOC	1 ~ 5
3	757 KLOC	one 3 KLOC .dll file in C	1 ~ 4

8.1 Results

The results of applying the Pallino on the three case studies were the same as when we used the original separated tools (D-TIZ, TID-BITZ, and CAAFI) [33, 34]. Generally, the higher percentage of affected exported component functions, the lower the percentage of test cases reduction, as shown in Figure 12.

In the best case, as much as 100% regression test case reduction can be achieved by the I-BACCI process if our analysis indicates the changes to the COTS component are not called by the glue code. This fact would not be known to the users of COTS component without I-BACCI analysis, such that they would still be tempted to do retest-all. When there are a lot of changes in the new release of the component, the I-BACCI process suggests a retest-all regression test strategy, similar to other RTS techniques. Also, the I-BACCI process is more effective when there are small incremental changes between revisions, as is true with all RTS techniques.

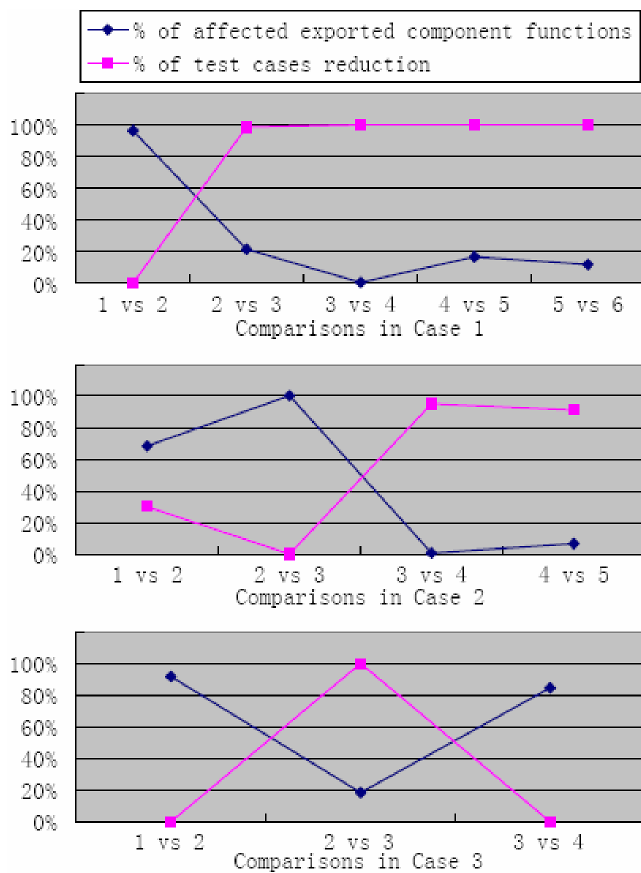

Figure 12: Relationship between the percentage of affected exported component functions and the percentage of test cases reduction for each case study

8.2 Running Costs

Pallino were run on an IBM T42 laptop with one Intel® Pentium® M 1.8 GHz processor and one gigabyte RAM. The comparisons of total time costs among different RTS strategies for each release of the three case studies are shown in Table 2. One assumption is that the mapping of all test cases with the glue code functions they cover is ready. Also, another limitation is that we only have rough estimation on time costs of test execution. Although there is no time costs in BCA and RTS, retest-all strategy takes a lot of time in test execution. Conducting the I-BACCI process without automation can be time consuming as well, especially for analyzing PE components. With the help of Pallino, the I-BACCI process can be completed in about one to two person hours for each release of the case studies. Depends on the percentage of test cases reduction determined by the I-BACCI process, the total time cost of the whole regression testing process can be reduced from five person months by retest-all strategy to one person hours in the best case.

8.3 Limitations

Pallino works only when the releases of components are built by the same compiler. If two compared releases are built by different compilers or linkers, Pallino will yield a significant number of false positives.

Table 2. Rough total time costs

Case	Approach	Time Costs (for each release)			
		BCA	RTS	Test Execution	Total
1	Retest-all	0	0	1 months	1 months
	I-BACCI (manually)	5 days	2 hrs	0 ~ 1 months	5 days ~ 1.1 months
	I-BACCI (w/ Pallino)	2 mins	2 hrs	0 ~ 1 months	2 hrs ~ 1 months
2	Retest-all	0	0	5 months	5 months
	I-BACCI (manually)	15 days	1 hr	0 ~ 5 months	15 days ~ 5.5 months
	I-BACCI (w/ Pallino)	5 mins	1 hr	0 ~ 5 months	1 hr ~ 5 months
3	Retest-all	0	0	4 days	4 days
	I-BACCI (manually)	>> 4 days	2 hrs	0 ~ 4 days	>> 4 days
	I-BACCI (w/ Pallino)	15~19 mins	2 hrs	0 ~ 4 days	2.5 hrs ~ 4 days

9. CONCLUSIONS AND FUTURE WORK

In this paper, we present Pallino, a tool that statically identifies binary code changes and their impact to support regression test selection for COTS-based applications when source code of components is not available. Based on the output of Pallino and the original test suit, testers can determine the regression test cases needed that cover the application glue code which is affected by the changed areas in the new COTS components. Pallino was designed to support the I-BACCI process but could be extended and/or modified to support other RTS methods for COTS components when source code is not available. Pallino can be applied to binary files of components in either COFF or PE format written in C/C++ at this stage. Three case studies, examining a total of fifteen component releases, were conducted at ABB on products written in C/C++. The results indicate Pallino can efficiently identify affected exported component functions, and therefore facilitate reducing the required number of regression test. With the help of Pallino, the I-BACCI process can be completed in about one to two person hours for each case study. Depends on the percentage of test cases reduction determined by the I-BACCI process, the total time cost of the regression testing process can be reduced from five person months to one person hours in the best case.

Besides expanding Pallino to adapt to more programming language and more of the COTS file types, such as components of Component Object Model (COM)[3] type and in ELF format, we plan to reduce the false positives caused by factors other than source code (e.g. build tools and target platforms). Additionally, extensive validation of both Pallino and I-BACCI RTS process will require more industrial case studies and data collection.

10. ACKNOWLEDGMENTS

This research was supported by a research grant from ABB Corporate Research. We would like to thank Dr. Cem Kaner for his help on the legal aspects of this paper.

11. REFERENCES

[1] "Bowers v. Baystate Technologies," in *Federal Reporter 3d*. vol. 320: United States Court of Appeals for the Federal Circuit, 2003, p. 1317.

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

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

[4] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs," in *The International Symposium on Requirements Engineering for Information Security*, 2001.

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

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

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley Professional, 1995.

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

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

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

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

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

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

[14] J. Laski and W. Szermer, "Identification of program modifications and its applications in software maintenance," in *International Conference on Software Maintenance*, 1992.

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

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

[17] L. Mariani, S. Papagiannakis, and M. Pezze, "Compatibility and regression testing of COTS-component-based software," in *29th International Conference on Software Engineering*, Minneapolis, MN, 2007, pp. 85-95.

[18] A. M. Memon, "A process and role-based taxonomy of techniques to make testable COTS components," in *Testing Commercial-off-the-shelf Components and Systems*, S. Beydeda and V. Gruhn, Eds. Berlin, Germany: Springer-Verlag, 2005, pp. 109-140.

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

[20] M. Pietrek, "An In-Depth Look into the Win32 Portable Executable File Format," in *MSDN Magazine*, March 2002.

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

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

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

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

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

[26] F. Vokolos and P. Frankl, "Empirical evaluation of the textual differencing regression testing technique," in *International Conference on Software Maintenance*, 1998.

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

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

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

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

[31] J. Zheng, B. Robinson, L. Williams, and K. Smiley, "An Initial Study of a Lightweight Process for Change Identification and Regression Test Selection When Source Code is Not Available," in *16th IEEE International Symposium on Software Reliability Engineering*, Chicago, IL, USA, 2005, pp. 225-234.

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

[33] J. Zheng, B. Robinson, L. Williams, and K. Smiley, "Applying Regression Test Selection for COTS-based Applications," in *28th IEEE International Conference on Software Engineering*, Shanghai, China, 2006, pp. 512-521.

[34] J. Zheng, L. Williams, B. Robinson, and K. Smiley, "Regression Test Selection for Black-box Dynamic Link Library Components," 2nd International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques, 2007.