Automated Association of Code Changes with Computed Behavioral Differences

Jonas Michel
University of Texas at Austin
Electrical & Computer Engineering
jonasrmichel@mail.utexas.edu

ABSTRACT

Developers often find it necessary to know how source code changes affect the run-time behavior of a program. Currently, identifying behavioral differences between program versions is a tedious manual task that is not exclusively supported by tools or research techniques. The goal of this project is to determine if a mapping from a set of known structural changes to a set of known behavioral changes can be inferred and if so, how reliable the mapping is. This report describes an approach that identifies structural differences between two versions of a program and computes the corresponding attributable dynamic behavioral differences. To enable automation of the approach and experimental evaluation, Bidirectional Structural-Behavioral Change Mapper (BsbCmapper), a prototype Java tool, is presented. Using BsbCmapper, I demonstrate through a small case study on an open-source project that an informative mapping can in fact be automatically inferred between structural and dynamic behavioral differences. The mappings are furthermore demonstrated to be both accurate and reliable.

1. INTRODUCTION

Many practical and analytical scenarios exist where developers face a need to understand run-time behavior differences between program versions. More specifically, there are many cases in which a developer would like to know if and how specific structural changes contribute and relate to dynamic behavior changes. For example, consider a developer performing a refactoring on a set of classes. To verify successful completion of this task, he or she would need to check that the changes made did not induce behavioral changes in the overall system's behavior. Similarly, consider a test engineer improving a regression test suite for a recently updated software system. In order to create tests that effectively assess the new functionality, he or she would be interested in knowing if there were any changes to code covered by the existing test suite. Further scenarios where a developer must understand how specific code changes relate to run-time behavior differences include performance monitoring, bug resolution, software evolution tracking, system maintenance, and program comprehension. The process of determining how particular code differences between two program versions contribute to dynamic behavior differences can often be difficult and tedious, especially in larger more complex software systems.

Current tools and research techniques do not directly enable easy understanding of run-time differences between program versions. This tedious task must be performed manually, likely requires being familiar with a system or program, and requires the use of a handful of tools in tandem to dig through source code and test results, comparing the two by inspection. The efficiency with which this task is performed depends greatly on the developer's prior understanding of and familiarity with the system. Even in the case where a developer does have a good understanding of the system he or she is working on, great amounts of time could be devoted to correlating code changes and dynamic behavior differences. Consider, again, the developer in the refactoring example. If the refactored program did actually exhibit different behavior, the developer would then be faced with the challenge of pinpointing the changes responsible for the unwanted behavioral changes. Similarly, the test engineer improving the test suite would need to manually sort through source code changes and inspect results from, potentially expensive, test executions to successfully complete their job.

In this project, I am interested in determining if a mapping from a set of known structural changes to a set of known behavioral changes can be inferred and if so, how reliable the mapping is. This report presents a methodology and approach for automatically associating structural code changes with detected dynamic behavior differences. To enable automation of the approach and a means of evaluating the methodology, I designed *Bidirectional Structural and Behavioral Change Mapper (BsbCmapper)*, a prototype Java tool that builds on two existing research tools. Using BsbCmapper, a small case study is conducted on various versions of FreeMarker [5], an open-source HTML template engine for Java servlets, to evaluate the utility and effectiveness of the approach. The case study highlights some of the limitations of the tool, but also verifies that an informative mapping from structural to behavioral changes can in fact be inferred automatically.

The goal of this project is not to provide a silver bullet with which a developer can pinpoint the behavioral differences that resulted from a source code change. Rather, this methodology and prototype tool will hopefully be a basis for future research and provide a means of alleviating some of the burden placed on developers needing to know what behavioral differences exist between two programs and why they exist.

The rest of the report is organized as follows: Section 2 describes related work in this area of research, Section 3 contains a high level description of the methodology and implemented approach, Section 4 describes details of the BsbCmapper implementation, Section 5 describes the verification process and presents a case study of BsbCmapper on FreeMarker, and Section 6 finishes with conclusions that can be drawn from the case study's results and discusses areas of future work.

2. RELATED WORK

The approach implemented in this project draws heavily from existing research. However, it does so in such a way to produce original results. Six main bodies of work form the foundation of

this project, two of which are used directly. My project shares a very similar motivation and problem definition with all six of these related pieces of work, however my project's solution is more generally applicable.

The motivation, problem definition, and solution presented in "Scaling Regression Testing to Large Software Systems" by Orso, Shi, and Harrold [1] is, in essence, equivalent to that of this project. Orso et al. describe a technique that identifies behavioral differences between two program versions, but for the purposes of regression test selection. Accordingly, their work infers an association between behavioral differences and test cases. My project follows this same line of thought, but extends the application from tests to structural changes. The tradeoff, however, is precision. While my project's technique is more generally applicable and encapsulates more use case scenarios, it sacrifices a great deal of confidence in accuracy to do so.

"Semantics-Aware Trace Analysis" by Hoffman, Eugster, and Jagannathan [6] presents an approach for representing the dynamic behavior of a program using execution traces to create semantic trace abstractions. "Semantic views" are aggregate collections of events with shared semantic traits found in a program's execution trace. Using a longest common subsequence (LCS) based algorithm, semantic views produced by different program versions can be compared for differences. The purpose making these comparisons is using them to pinpoint exactly the cause of regressions in large programs. The advantage of this approach is that it preserves semantic context. However, it does not take structural context into account. At a high level, my approach differs from this work in three ways. First, Hoffman et al.'s work attempts to identify the semantic root cause of behavior differences. That is, it creates a mapping of behaviors to code changes. I am attempting to make the opposite association (i.e. code changes to behavior differences). Second, Hoffman et al.'s work has a very specific application: regression identification. My project applies to a more general set of uses. Finally, behavior-to-code association is performed by a process of elimination in Hoffman et al.'s approach. Conversely, my project's approach uses a constructive approach to make inferences.

"Automated Behavioral Regression Testing" by Jin, Orso, and Xie [7] proposes a tool called Behavioral REgression Testing (BERT), which identifies behavioral differences between program versions by logging behavioral characteristics during program execution. My approach is very similar to that used in BERT in that it involves linking together a set of existing tools to produce a unique output. However, this work too is primarily focused on behavior differencing for the purpose of fault identification. BERT identifies behavioral differences and additionally determines if any of the differences can be characterized as faults. The limitation of the BERT tool is that it leaves a large portion of the analysis to the user. After BERT is executed on two program versions, the user must manually verify if BERT's fault identifications are indeed valid. My project attempts to go one step further and attempt to perform some of this final portion for the user automatically and for more general use purposes, not strictly for fault detection.

"Automated Inference of Structural Changes for Matching Across Program Versions" by Kim, Notkin, and Grossman [8] introduces a means of automatically inferring API-level changes between two program versions and concisely representing these changes as first-order relational logic rules. I use this work as a building block for my project because of its efficient and extendable representation of structural differences. Furthermore, this alternative representation of API-level changes couples nicely with the PARCS, the second building block of this project's approach.

"Tracking Performance Across Software Revisions" by Mostafa and Krintz [9] presents Performance-Aware Revision Control Support (PARCS), a tool that identifies behavioral differences between program versions and attempts to attribute likely structural changes to behavioral changes. I use the PARCS process as another building block in my project because of its ability to identify topological and performance differences, its extendibility, and because of its intimate similarity to my approach. Despite how similar this work is to my project, it is only addresses half of the problem that motivates my project. Mostafa and Krintz's work attempts to map detected behavioral changes to code changes. My project extends this work by performing inference in the opposite direction as well, by mapping known code changes to detected behavioral changes. Furthermore, the change rules that explain identified code changes in my project can be extended to help explain the behavioral differences identified by PARCS. Any behavioral differences that were not a result of method additions, deletions, or direct modifications, PARCS attributes to indirect method modifications and/or non-deterministic effects and throws away. The application of Kim et al.'s API-level change rules enables even further insight into these particular behavioral differences.

"Chianti: A Tool for Change Impact Analysis of Java Programs" by Ren, Shah, Tip, Ryder, and Chesley [10] presents a tool that is conceptually very similar to PARCS in the sense that it attributes structural differences to behavioral differences, however, again, the intended use of Chianti (test selection and impact analysis) is much more specific and geared towards regression identification.

3. APPROACH

At a high level, my project's approach consists of three components, the first two of which implement tools and/or algorithms contributed by existing research. Given two versions of a program (P and P') and a test suite (T), first, API-Level Code Matching [8] is performed on P and P' to identify method-level changes and transformations that describe change details. Second, a variation of a technique used by PARCS [9] is implemented to detect dynamic behavior differences between P and P' triggered by T and attribute them to categorized code changes. Finally, set comparison is performed to infer a mapping of API-Level Code Change rules to behavioral differences.

3.1 API-Level Code Changes

The first component directly uses an API-Level Code Change tool introduced by Kim et al. in [8]. P and P' are given as inputs to the tool, which automatically infers probable code changes at or above the method level and represents these inferred changes as a set of first-order relational logic rules. Each change rule consists of an applicable scope, exceptions to the rule, and the transformation that occurred between P and P'. The transformation is one of nine predetermined transformations, which includes argument appended, argument deleted, and varying types and granularities of object replacement. The set of rules (R) representing a set of likely structural changes (ΔS) produced for P and P' is saved in XML format for use in the second and third steps. This tool also provides an API that enables easy access to inferred change rules and their applicable scope.

Kim et al.'s API-Level Code Matching tool was chosen as the basis for identifying structural differences because of its concise and descriptive representation of changes. The use of change rules as a change vocabulary, as opposed to a *diff*, allows structural differences to be grouped together by similarity in a human-readable format. Furthermore, the tool is highly extensible. In essence, this component answers the question, "what changed *structurally* and how?".

3.2 The PARCS Process

The second component of the approach implements a process based on a technique used by the Performance-Aware Revision Control Support (PARCS) behavioral differencing tool introduced by Mostafa and Krintz in [9]. Figure 1 illustrates the PARCS process at a high level.

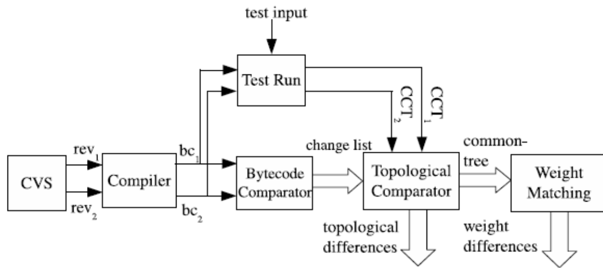

Figure 1. The PARCS process [9]

Given P (rev_1), P' (rev_2), and T (test input), PARCS first compiles P and P' to obtain the bytecode program versions. The bytecode versions of P and P' are executed using T as input to generate their respective calling context trees (CCTs). Each node of the CCT, representing a method call, is annotated with performance metrics, specifically call site, absolute execution time, and relative execution time. The bytecode versions of P and P' are also compared to generate a change list of added, deleted, modified, and renamed methods. The change list and CCTs are used to identify topological behavioral differences (ΔB_T) as a result of added or deleted methods ($\Delta b_{A/D}$), directly modified methods (Δb_M), and indirectly modified methods or non-deterministic effects (Δb_{ND}). After these topological differences are excised, the remaining CCTs are identical in topology, but may vary in performance metrics. PARCS uses a weight matching algorithm to quantify the degree of similarity, or overlap, between the two trees in terms of their annotated performance data. This information represents the behavioral weight difference between P and P' (ΔB_W).

The PARCS process was selected for two reasons. First, PARCS makes clever use of CCTs, a very common structure used for program execution representation. CCTs are advantageous because they can be traversed, sorted, and matched using standard tree algorithms and provide a good tradeoff between size and accuracy compared to other dynamic behavior abstractions [9]. PARCS uses CCTs to identify behavioral differences in terms of topology as well as performance. Second, there is an intimate similarity between the motivation and problem definition of PARCS and that of my project. In essence, this component answers the question, "what changed *behaviorally* and how?".

3.3 Structural and Behavioral Mapping

The final component of the approach performs set comparison of information identified in the first two components to infer a mapping from structural to behavioral changes. There are three cases that are considered.

A. Structural changes are identified that do not result in a behavioral change. More formally expressed, this is the case for methods

$$m: (m \in \Delta S \cap m \notin (\Delta B_t \cup \Delta B_w))$$

B. Structural changes are identified that do result in a behavioral change. More formally expressed, this is the case for methods

$$m: (m \in \Delta S \cap m \in (\Delta B_t \cup \Delta B_w))$$

With this set, a mapping of change rules, R, to behavior differences, ΔB_T ($\Delta b_{A/D}$, Δb_M, Δb_{ND}) and ΔB_W can be constructed.

C. Dynamic behavior changes are identified that are not a result of a structural change. More formally expressed, this is the case for methods

$$m: (m \in (\Delta B_T \cup \Delta B_W) \cap m \notin \Delta S)$$

This third and final component of the approach establishes a bridge between the first two components. In the case of (B), it provides potential answers to the questions, "what behavioral changes occurred as a result of the structural changes and why?".

3.4 Threats to Validity

Validity is a very large concern for this approach and the project in general. There are many threats to internal, construct, and external validity. The largest source of threats to internal and construct validity originate from the fact that test input is limited to the provided test suites. Therefore behavioral changes may actually be the result of the nature of a particular test case and not necessarily because of a structural change. This exemplifies a confounding type of threat to internal validity. Furthermore, a valid question is whether this approach will measure the relation between structural and behavioral changes, or simply the coverage provided by the test suite. This is certainly an important point, and the results presented in the case study (Section 5) demonstrate potential effects of this issue. Finally, threats to external validity are also present. Given the threats to both internal and construct validity in addition to the inability to validate this approach on a large scale, it would be almost impossible to prove that an approach claiming a concrete mapping of structural to behavioral differences was correct for any arbitrary set of programs. However, that is not the goal of this project. I do not claim that my results are concrete, but rather that the mappings are "likely", identify useful information, and are a good starting point for further investigation.

4. Implementation

To enable automation of the approach and a means of evaluating its methodology, I designed *Bidirectional Structural and Behavioral Change Mapper (BsbCmapper)*, a prototype Java tool. The first component of BsbCmapper uses Kim et al.'s API-Level Code Change tool directly to identify structural change details. The second component implements a reverse-engineered version of the PARCS process to identify dynamic behavior differences. The final component aggregates the first two components' results into standardized sets with which it performs set comparison to infer relationships between structural and behavioral changes. Additionally, the third component also compares the structural changes identified by the API-Level Code Change tool for analysis purposes. Due to time restrictions and the sharp learning curves of some of the libraries the tool utilizes, BsbCmapper

suffers from several important limitations. BsbCmapper's limitations are pointed out at the end of this section.

4.1 API-Level Code Change Use

Kim et al.'s API-Level Code Change tool provides a useful API, which enables easy access to the changed methods and change rules it infers between two versions of a program. BsbCmapper makes direct use of this API to query the matching rulebase it constructs. The API-Level Code Change tool also defines a canonical *JavaMethod* data type to uniquely represent methods. For the sake of convenience and consistency, BsbCmapper also internally represents methods using this data type.

4.2 Reverse-Engineered PARCS

Because PARCS is not a publicly available research tool, BsbCmapper implements a reverse-engineered version (RE-PARCS) based on the details in [9]. Mostafa and Krintz's paper leaves out many low-level details, therefore BsbCmapper's RE-PARCS implementation makes some assumptions about how some of these low-level tasks are carried out.

4.2.1 Bytecode Comparison

BsbCmapper's RE-PARCS implementation uses the Apache Byte Code Engineering Library (BCEL) [4] to perform method-level bytecode comparison of two program versions. The PARCS method change sets (i.e. added, deleted, modified, and renamed) are constructed exactly as described in [9]. Following the PARCS bytecode comparison technique, BsbCmapper compares compiled bytecode (.class) files that have identical package and class file names. Uncommon packages and class files are reported, but are not used for any further analysis. An important assumption made in this component is the notion of a "modified method." PARCS defines a modified method as one that is "present in both revisions with everything identical except for the code body." A direct equivalence comparison of BCEL method code cannot be used directly because bytecode contains information beyond the actual compiled code, for example call site. Therefore, two methods' code could be syntactically identical, but exist at different points in their respective program version and would therefore be considered not equivalent by a direct BCEL method code equivalence comparison. BsbCmapper defines code equivalence as the union of several BCEL code attributes. These attributes can be summed to an integer value, which is used to represent a method's code's comparable identity.

4.2.2 CCT Generation

BsbCmapper's RE-PARCS implementation generates CCTs using AspectJ [3], an aspect-oriented extension to Java. The tool uses a simple logging aspect that is woven into a program's source code when the program is built. The CCT-generating aspect logs method invocation attributes (i.e. caller, call-site, and execution time) to an XML file when the program it is woven into is executed (e.g. with test input). BsbCmapper stores CCTs internally in a traversable tree data structure and additionally in a depth-indexed lookup table for fast access to nodes at specific depths. The complementary use of a depth-indexed lookup table in addition to the tree data structure cuts down on complexity and runtime significantly.

4.2.3 Topological Comparison

The BsbCmapper RE-PARCS uses the PARCS Relaxed Common-Tree Matching topological comparison algorithm described in [9] verbatim to identify dynamic behavior differences represented as topological differences in program versions' CCTs. Using the changed methods identified by bytecode comparison, topological differences are iteratively identified, attributed to method additions, deletions, modifications, or non-determinism, and excised from the respective CCT. The nodes (methods) and subtrees excised are saved for use in BsbCmapper's structural and behavioral change mapping component.

4.2.4 Performance Lite-Weight Comparison

PARCS uses an iterative performance weight matching algorithm based on that introduced by Zhuang et al. in [11] to compute the performance overlap of two topologically-identical CCTs. Due to time limitations, BsbCmapper's RE-PARCS computes overlap using the same overlap equation as PARCS, but does not perform any iterative weight adjustments to improve the performance overlap of two CCTs. BsbCmapper's lite-weight performance comparator reports and stores methods with relative performance differences above a user-defined threshold (0-1).

4.3 Bidirectional Change Mapping

This final component of BsbCmapper aggregates differences identified by the first two components and compares their overlap and uniquely identified differences. The set of structural changes identified by the API-Level Code Change tool is taken to be the union of the tool's left-to and right-to domain. The set of behavioral differences identified by the RE-PARCS implementation is taken to be union of the methods identified as differences by the topological comparator and the performance weight comparator. More specifically,

$\Delta S = \Delta s_L \cup \Delta s_R$

$\Delta B = \Delta B_T \cup \Delta B_W$

Where,

$\Delta B_T = \Delta b_A \cup \Delta b_D \cup \Delta b_M \cup \Delta b_{ND}$

$\Delta B_W = m : pweight(m) \geq threshold_W$

The sets $\Delta b_{A/D}$ consist of the roots of subtrees identified as differing due to added or deleted methods and each root's respective caller. The set Δb_M consists of the roots of subtrees identified as differing due to directly modified methods and the chain of modified dominators from a subtree root up to the CCT root. The set Δb_{ND} consists of the roots of subtrees identified as differing due to non-determinism. Finally, ΔB_W consists of the methods with performance weight differences greater than the specified weight threshold.

It is important to note that only the roots of topologically differing subtrees are considered for comparison. This is because we are only interested in the specific points where behavior between two versions of a program *began* to diverge. What happens after this point is not comparable.

BsbCmapper reports statistics regarding each stage of the RE-PARCS implementation (e.g. differing packages and classes, number of methods identified as changed for each bytecode change category, number of subtrees and nodes excised due to types of changes, common CCT size, etc.) and optionally the results of each RE-PARCS stage as well.

BsbCmapper performs two set comparisons: structural-structural change set comparison and structural-behavioral change set comparison. The first comparison is performed between the structural change sets identified by API-Level Code Change matching and bytecode comparison. BsbCmapper identifies structural changes that are exclusively identified by API-Level

Code Change matching and exclusively by bytecode comparison, and reports a detailed mapping breakdown where the two overlap (i.e. for each API-Level Code Change type identified, the aggregate number of bytecode change types it maps to). This is done to provide more insight into the results of the second set comparison. The second comparison is performed in a similar fashion between the API-Level Code Change structural change set (ΔS) and the RE-PARCS behavioral change set (ΔB). BsbCmapper identifies changes that are exclusively identified by API-Level Code Change matching and exclusively by RE-PARCS topological and weight comparison, and reports a detailed mapping breakdown where the two overlap. In addition to the aggregate breakdown of the two sets' commonality, the specific methods that are common between the two sets and their change type in each set are reported.

4.4 Limitations

BsbCmapper suffers from several practical limitations mainly due to decisions that had to be made due to time restrictions. First, the BsbCmapper process is not fully automatic. A user must generate results from the API-Level Code Change tool and CCTs for the program versions being compared prior to using BsbCmapper. In other words, BsbCmapper does not automatically kick off and collect API-Level Code Change version comparison and CCT generation; these steps must be performed manually. Second, BsbCmapper's user interface (UI) is strictly text-based, which makes interpretation and navigation of results difficult. Third, and most importantly, BsbCmapper suffers from memory limitations. It was discovered too late how large the CCT XML files actually become for even a small to medium sized open-source projects (sometimes over a gigabyte). Because BsbCmapper uses RAM to access CCTs and not a relational database, it is limited in the types and sizes of programs it can analyze.

5. EVALUATION RESULTS

To evaluate BsbCmapper's RE-PARCS implementation, small scale verification was performed on a feature-by-feature basis. Second, a case study is conducted on various versions of FreeMarker, an open-source HTML template engine for Java servlets, to empirically evaluate the utility and effectiveness of the implemented approach.

5.1 Verification

Originally, I had planned to verify the RE-PARCS implementation by running it on the same open-source test subjects used by Mostafa and Krintz and comparing RE-PARCS' results to those listed in [9]. However, I ran into many problems due to the memory limitation described in Section 4.4 – the CCTs generated for each of these test subjects were simply too large to be evaluated by BsbCmapper. Therefore, small scale verification was performed on a feature-by-feature basis. Using two versions of a small test program, each of the RE-PARCS features were tested extensively to verify that their behavior was consistent with their counterpart's as described in [9].

5.2 Case Study: FreeMarker

My experimental platform is a dual-core Intel Core 2 Duo machine clocked at 2.0 GHz with 4M of L2 cache and 2GB of main memory running the Windows 7 operating system. The Java virtual machine used is HotSpot version 17.0-b17 within JDK 1.6.0_21.

FreeMarker [5] is an open-source HTML template engine for Java servlets and was chosen because its size is within BsbCmapper's memory capacity and exhibits an overlap of identified structural and behavioral changes. As a side note, BsbCmapper was successfully tested on many different open-source projects for this project; however it was often the case that while there would be many RE-PARCS-identified behavioral differences, the API-Level Code Change tool would not identify any change rule matches. Therefore, it was somewhat of a challenge to find a test subject for which both tools identified changes *and* for which those changes overlapped. The results obtained using FreeMarker are included in this report because they do in fact meet all of these criteria. This section contains results from the FreeMarker case study and one additional experiment.

5.2.1 Test Procedure

Seven versions of FreeMarker were selected whose generated CCTs were compatible with BsbCmapper's memory limitation. Table 1 shows details regarding these versions and their CCTs.

Version	Release Date	Files	Methods	CCT Nodes	CCT Depth
2.3.16	12/7/2009	20	1632	115	27
2.2.8	6/15/2004	15	1568	334	10
1.8.5	11/16/2004	16	1681	2121	49
1.8.2	10/6/2004	16	1641	2066	53
1.8.1	2/17/2004	16	1638	2066	51
1.8	12/6/2003	16	1634	2066	51
1.7.5	6/1/2002	15	1521	268	24

Table 1. FreeMarker version details

The CCT node count and depth originate from the CCTs generated by the test suite included with the respective version. Note that version 2.2.8 was released before versions 1.8.2 and 1.8.5. This is because 2.x versions contain some extra features that 1.x versions do not and are a secondary branch of the FreeMarker open-source project.

This case study is designed to answer the two research questions posed in this project. The first research question being, "*can a mapping from a set of known structural changes to a set of known behavioral changes can be inferred?*" and the second being, "*if a mapping can be established, how reliable is it?*".

To answer the first question, pairs of versions are compared using BsbCmapper. The test suite of the earlier version is used as test input to both program versions for CCT generation. To answer the second question, BsbCmapper's results are compared in terms of "version separation", the number of version releases that separate two FreeMapper versions. For example, the comparison of versions 1.8 and 1.8.1 possess a version separation of 1 whereas a comparison of versions 1.8 and 1.8.2 possess a version separation of 2. This study uses BsbCmapper to compare all of the possible 1.x combinations (10) and the one 2.x combination for a total of 11 combinations. 1.x and 2.x versions are not directly compared against each other because their feature set is not similar enough, however their results are indirectly comparable.

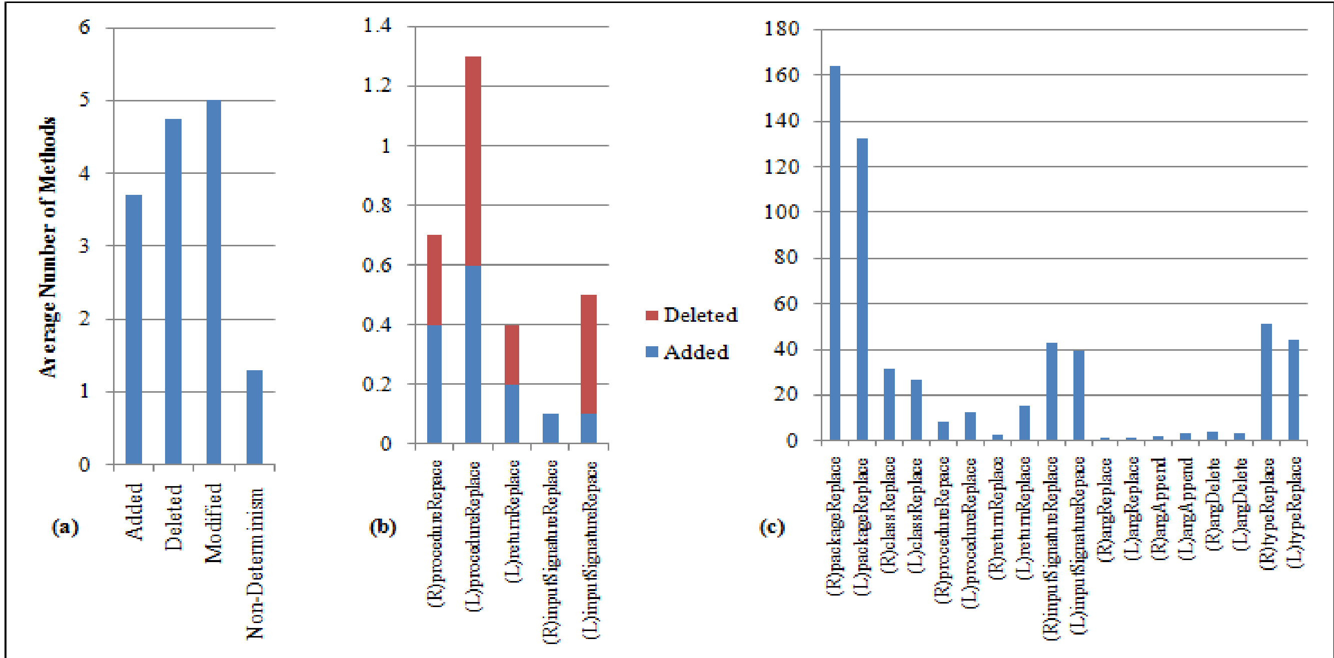

Figure 2. Average number of methods and change types identified exclusively by (a) RE-PARCS and (c) API-Level Code Change matching and (b) their common overlap

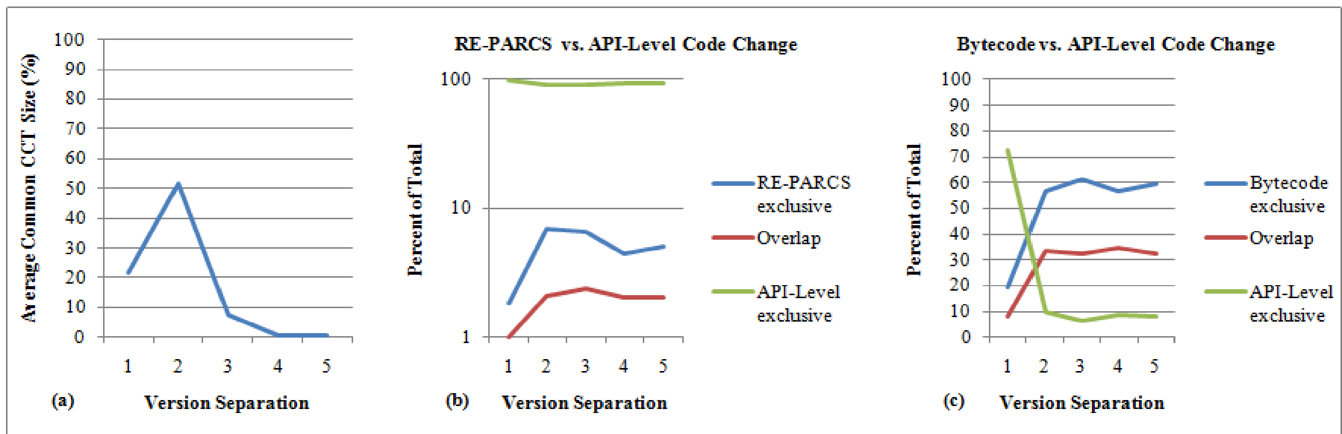

Figure 3. Effects of version separation on (a) common CCT size, (b) RE-PARCS and API-Level Code Change set overlap, and (c) Bytecode and API-Level Code Change set overlap

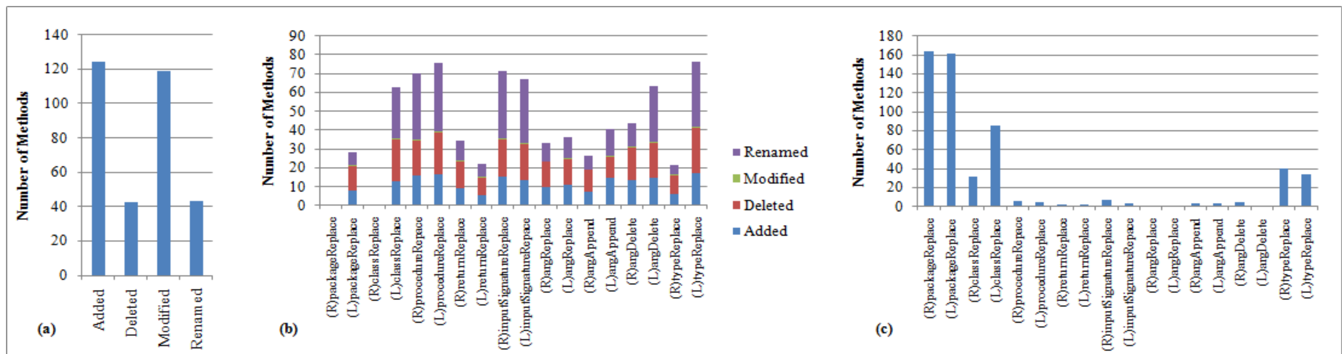

Figure 4. Average number of methods and change types identified exclusively by (a) Bytecode comparison and (c) API-Level Code Change matching and (b) their common overlap

5.2.2 Case Study Results and Discussion

In all version combinations of this study, the API-Level Code Change tool is used with a seed threshold of 0.7 and an exception threshold of 0.3 (see [8] for threshold details). A performance weight threshold of 0.01 is used for the RE-PARCS performance weight comparator in all version combinations of this study.

Figure 2 shows a breakdown of the number of methods identified exclusively by RE-PARCS (Figure 2(a)), exclusively by API-Level Code Change (Figure 2(c)), and their overlap (Figure 2(b)) averaged over the 11 version combinations used. On average, RE-PARCS identifies a comparable amount of differences due to additions, deletions, and direct modification that API-Level Code Change does not capture. On the other hand, API-Level Code Change identifies many changes due to package, type, and input signature replacements that RE-PARCS does not. This behavior is most likely due to the fact that RE-PARCS does not consider methods in packages and classes that do not have identical names. Therefore, even if their package or class was simply renamed, methods in non-identical packages and classes are entirely excluded from the RE-PARCS topological and performance weight comparison components.

In the case of these FreeMarker combinations, the two tools overlap only on differences attributed to additions and deletions by RE-PARCS. API-Level Code Change categorizes these additions and deletions as procedure replacements, return replacements, and input signature replacements. This effect is most likely due to, first the average number of modified methods is exceptionally small and second, RE-PARCS' method modification is at a sub-method level. Therefore, it is likely that the API-Level Code Change tool will not detect RE-PARCS' modified methods. Nevertheless, because of the relatively small amount of overlapping methods and because BsbCmapper explicitly identifies on which methods the two tools overlap, I was able to inspect the overlapping cases by hand. In every case, the API-Level Code Change rule was accurate!

It is important to note that RE-PARCS does not identify any changes due to performance weight differencing. This effect is due to the amount of excising performed by RE-PARCS' topological comparison component. In most of the FreeMarker version comparisons, very little of the CCT are identified as topologically identical, and the performance weight differences in the remaining CCTs are less than the performance weight threshold.

Figure 3 shows three noteworthy trends for FreeMarker version comparison over different version separation distances. In all cases show in this figure, the values used are obtained by averaging the data points at each version separation amount. Figure 3(a) illustrates the effects of version separation on RE-PARCS' topological comparison. The father apart two compared versions are the more behavioral differences they exhibit. Therefore, with more topological differences, RE-PARCS performs more excising. After three versions of separation, the topological comparator almost entirely excises both CCTs. Figure 3(b) shows the relative proportion of RE-PARCS-identified behavioral changes, API-Level Code Change-identified structural changes, and their overlap as the version separation is increased between two versions of FreeMarker. The y-axis is a log scale so that the smaller values are more apparent. In all cases, API-Level Code Change identifies many more structural changes than RE-PARCS identifies behavioral changes. However, their overlap peaks in the case of BsbCmapper comparisons that are three versions apart, after which there is not much variance. This effect is likely due to the fact that in this case after three versions of separation, there are no more topological differences from which to draw; the CCTs are entirely excised! What is important is that, though the overlap between the identified changes of the two methodologies, there is not much variance even as more and more distant versions are compared. This indicates that the reliability of the mapping remains constant.

5.2.3 Additional Experiment and Discussion

An additional experiment is conducted using BsbCmapper results obtained during the FreeMarker case study. To better gauge the quality and more informatively interpret the case study results, a similar change set comparison is performed between the structural changes identified by RE-PARCS' bytecode comparison and the API-Level Code Change tool.

First, Figure 3(c) shows the relative proportion of bytecode-identified structural changes, API-Level Code Change-identified structural changes, and their overlap as the version separation is increased between two versions of FreeMarker. When the version separation is small, API-Level Code Change identifies a majority of the structural differences. However, beyond a separation of one version, bytecode comparison detects a majority of the structural changes, but there is roughly a 30% overlap of the two methods' identified changes. Additionally, beyond a separation of one version, there is not much variance between the two methods' relative amount of identified changes. This effect is likely due to the fact, again, that RE-PARCS excludes packages and classes that do not have identical names. It could be the case that modifications that occur after more than one version separation are predominantly in added or renamed packages and classes. Therefore, the bytecode comparator reaches a ceiling value.

Figure 4 shows a more detailed breakdown of the types of changes identified exclusively by each structural change identification method and their common overlap. Figure 4(a) illustrates that a majority of the bytecode-exclusive identified changes are added and directly modified methods, whereas Figure 4(c) illustrates that a majority of the API-Level Code Change-exclusive identified changes are package and class replacements. This effect emphasizes previous observations; RE-PARCS' bytcode comparison does not detect changes within potentially renamed packages and classes and API-Level Code Change has difficulty detecting sub-method level changes (i.e. bytecode-identified directly modified changes). A clear trend in the overlap illustrated in Figure 4(b) is that bytecode-identified modified methods make up only a very small amount of the commonly identified API-Level Code Change methods. The rest of the bytecode change types are more or less evenly distributed within each API-Level Code Change type.

6. CONCLUSIONS AND FUTURE WORK

The experimental evaluation of this project's approach using BsbCmapper, though limited, verifies that not only can a mapping from structural to behavioral changes be automatically inferred, but additionally, it is accurate and retains reliability! In each of the hand-inspected cases where the API-Level Code Change structural changes overlapped with the RE-PARCS-identified dynamic behavior differences, the API-Level Code Change rule listed was accurate and, within the low-level context of the bytecode change semantics, the RE-PARCS change categorization was accurate as well. Due to time limitations, I was not able to

identify the significance of the identified changes within the context of the FreeMarker program, but the fact that API-Level Code Change rules can successfully be mapped to dynamic behavior differences means that in these few cases, more detailed information is provided as to why this behavior difference exists!

The relatively small amount of behavioral differences detected by RE-PARCS with respect to the number of detected API-Level Code Change differences in the FreeMarker case study could be attributed to two possibilities. First, this effect could be an indication of an insufficient test suite. Second, and more likely, because RE-PARCS does not consider methods in non-identically named packages and classes, modification to identical methods in renamed packages or classes are ignored. The API-Level Code Change tool indicated that there were many package and class replacements within the compared versions of FreeMarker. Any changes made to identical method within these packages would not have been considered by RE-PARCS.

The results observed in the small scale case study, though very dependent on the test subject, do verify that a bidirectional mapping structural and behavioral differences between two versions of a program can in fact be automatically inferred and that the mapping is reliable. Future work will need to directly address the practical limitations of BsbCmapper in order for the utility of this approach to be realized. If BsbCmapper is to actually be used, it will need to be made fully automatic to reduce the effort required to use it. Secondly, the nature of BsbCmapper's results necessitates an interactive graphic user interface (GUI). The tool has the potential for reporting more text-based results than any developer would care to look at no matter how useful they were stated to be. A more useful approach would be to report high-level results initially and provide a UI that enabled a developer to search and/or dig deeper in specific places. Finally, BsbCmapper will need to store and access CCTs via a relational database for it to be used on programs with any amount of test input. The current memory limitations do not enable behavior triggered by large test suites to be captured.

In conclusion, this project demonstrated that a bidirectional mapping can in fact be inferred between structural code changes and dynamic behavior differences. The reliability of this mapping was demonstrated to remain non-varying even as less similar program versions were compared. Though very limited in its utility, the BsbCmapper implementation of this project's approach provides more information than existing tools about why certain behavior differences exist between two program versions in the context of structural code changes.

7. REFERENCES

[1] A. Orso, N. Shi, and M. Harrold. Scaling Regression Testing to Large Software Systems. In SIGSOFT '04/FSE-12: *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on foundations of software engineering* (Newport Beach, CA, USA, 2004), ACM, pp. 241-251.

[2] A. Villazón, W. Binder, P. Moret, and D. Ansaloni. MAJOR: Flexible Tool Development with Aspect-Oriented Programming. In *Software Maintenance, 2009 – IEEE International Conference* (Edmonton, Alberta, Canada, 2009), IEEE, pp. 378-388.

[3] AspectJ Aspect-Oriented extension to Java. http://www.eclipse.org/aspectj

[4] Bytecode Engineering Library. http://jakarta.apache.org/bcel

[5] FreeMarker Java Template Engine Library. http://freemarker.sourceforge.net

[6] Hoffman, K., Eugster, P., and Jagannathan, S. Semantics-Aware Trace Analysis. In PLDI '09: *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (Dublin, Ireland, 2009), ACM, pp. 453-464.

[7] Jin, W., Orso, A., and Xie, T. Automated Behavioral Regression Testing. 2010 Third International Conference on Software Testing, Verification and Validation (Paris, France, 2010), pp. 137-146.

[8] Kim, M., Notkin, D., and Grossman, D. Automatic Inference of Structural Changes for Matching Across Program Versions. ICSE '07: *Proceedings of the 29th international conference on Software Engineering* (Minneapolis, MN, USA, 2007), IEEE, pp. 725-743.

[9] Mostafa, N., and Krintz, C. Tracking Performance Across Software Revisions. In PPPJ '09: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java* (Calgary, Alberta, Canada, 2009), ACM, pp. 162-171.

[10] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A Tool for Change Impact Analysis of Java Programs. In OOPSLA '04: *Proceedings of the 19th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications* (Vancouver, British Columbia, Canada, 2004), ACM, pp. 432-448.

[11] Zhuang, X., Kim, S. IO Serrano, M., and Choi, J.-D. Perdiff: A Framework for Performance Difference Analysis in a Virtual Machine Environment. In CGO '08: *Proceedings of the sixth annual IEEE/ACM international symposium on code generation and optimization* (New York, NY, USA, 2008), ACM, pp. 4-13.