Automated Behavioral Regression Testing[1]

Wei Jin
Georgia Institute of Technology
weijin@gatech.edu

Alessandro Orso
Georgia Institute of Technology
orso@cc.gatech.edu

Tao Xie
North Carolina State University
xie@csc.ncsu.edu

***Abstract*—When a program is modified during software evolution, developers typically run the new version of the program against its existing test suite to validate that the changes made on the program did not introduce unintended side effects (i.e., regression faults). This kind of regression testing can be effective in identifying some regression faults, but it is limited by the quality of the existing test suite. Due to the cost of testing, developers build test suites by finding acceptable tradeoffs between cost and thoroughness of the tests. As a result, these test suites tend to exercise only a small subset of the program's functionality and may be inadequate for testing the changes in a program. To address this issue, we propose a novel approach called BEhavioral Regression Testing (BERT). Given two versions of a program, BERT identifies behavioral differences between the two versions through dynamical analysis, in three steps. First, it generates a large number of test inputs that focus on the changed parts of the code. Second, it runs the generated test inputs on the old and new versions of the code and identifies differences in the tests' behavior. Third, it analyzes the identified differences and presents them to the developers. By focusing on a subset of the code and leveraging differential behavior, BERT can provide developers with more (and more detailed) information than traditional regression testing approaches. To evaluate BERT, we implemented it as a plug-in for Eclipse, a popular Integrated Development Environment, and used the plug-in to perform a preliminary study on two programs. The results of our study are promising, in that BERT was able to identify true regression faults in the programs.**

I. Introduction

During maintenance, software is modified to enhance its functionality, eliminate faults, and adapt it to changed or new platforms. When a new version P' of a program P is produced, developers must assess whether the changes that they introduced in P' behave as expected and do not affect the unchanged code in unforeseen ways. To this end, developers typically rerun on P', completely or in part, a set of existing test cases (*i.e.,* a regression test suite), consisting of test inputs and oracles. If one or more of the test cases that executed successfully on P cause an unexpected failure when run on P', the developers would know that the changes introduced regression faults and would use these test cases to investigate and eliminate such faults.[2]

[1]We presented an early version of this work at WODA 2008 [1]. In this paper, we extend the definition of the approach, describe the BERT-PLUGIN tool integrated into Eclipse, and present a more extensive evaluation of the approach on a real program and real changes.

[2]Developers would expect some of the existing tests to fail based on the changes made to the code. These tests, normally called *obsolete test cases*, would either be discarded or modified to run on the new version.

Ideally, this traditional approach to regression testing can identify most change-related faults. However, in practice, the approach has a fundamental limitation: it relies exclusively on the quality of the existing test suite for P. If such test suite is inadequate, regression testing is likely to be ineffective. Unfortunately, regression test suites for real, complex programs often target only a small subset of the program behavior, for two main reasons. First, manually generating test cases that achieve high structural coverage of non-trivial programs is difficult and time consuming. Therefore, developers tend to focus on the core functionality of the program and possibly rely on alternative approaches to verify the rest of the program, such as smoke tests, beta testing, and inspection. Second, even in cases where developers manage to build coverage-adequate test suites (*e.g.,* by leveraging some automated test generation technique), they have to account for the oracle problem. Because writing accurate oracles can be as expensive as generating test inputs, developers often settle for approximated oracles that perform only partial checks of the outcome of a test input [2]. In fact, it is common to consider crashes (or uncaught exceptions) as de-facto oracles, even though they capture only a small subset of the possible erroneous behaviors of a program.

In summary, traditional regression testing that relies only on existing test suites can result in limited checking of the changed code because of one of two issues, or both: (1) the lack of test inputs that exercise a changed behavior; (2) the lack of an oracle that can identify such changed behavior. To address these issues, in this paper, we propose *BEhavioral Regression Testing (BERT)*, a novel approach that is meant to complement existing regression testing approaches. The goal of BERT is to accurately and automatically identify behavioral differences between two versions of a program by means of dynamic analysis.

Given information on which parts of the code have changed between P and P', BERT operates in three main phases. (To make the description of the approach more concrete, we describe an instantiation of BERT for the Java language, where the changed parts would consist of a set of classes C.) In the *first phase*, BERT leverages automated test generation techniques to create a large number of test inputs targeted at each of the changed classes. In its *second phase*, BERT considers each changed class c and each test input t created for c, runs t on the old and new versions of c, and compares the outcome of t in the two cases. The approach performs this comparison by checking several aspects of the

test executions: the state of c after the execution of t, the values returned by the methods of c invoked by t, and the various outputs produced by c during the execution of t. Finally, in its *third* phase, BERT analyzes any difference in test outcomes identified in the previous phase to abstract away some of the information and factor together related differences (*e.g.,* differences in the value of a given field observed for multiple test inputs). The result of this phase is a set of behavioral differences that BERT reports to the developers. Developers can then use this information to assess which of these changes may indicate the presence of a regression fault and eliminate the fault.

The characteristics of BERT allow it to overcome the aforementioned limitations of traditional regression testing approaches and enable it to provide developers with more information than such traditional approaches. By focusing on the (typically small) subset of the code that has changed, our approach can address the first limitation of existing approaches: the lack of test inputs that can adequately exercise the differences in behavior between P and P'. And by leveraging differential behaviors, BERT can sidestep the second issue with traditional regression testing and perform an accurate assessment of the changed code without the need for any externally provided oracle.

To evaluate BERT, we implemented it as a plug-in for Eclipse (http://www.eclipse.org), a popular Integrated Development Environment (IDE), and used the plug-in to perform a preliminary study on two programs: a small example and a real program with real changes. In the study, we applied the approach to the programs and their changes, and examined the feedback provided by BERT to the developers. Although our results are still preliminary and do not allow us to draw any definitive conclusion on the general effectiveness of the approach, they are nevertheless promising: BERT was able to identify regression faults in the programs. BERT also produces some potential false positives, but we believe (and provide some evidence that) ranking or filtering can be used to address this issue. Overall, our results provide initial evidence that BERT has the potential to produce useful information for developers by either giving developers confidence that the changed code behaves as intended or point them to potential issues in the code.

This paper makes the following main contributions:

- The definition of the concept of behavioral regression testing, a novel approach to regression testing that can complement existing approaches by addressing two of their major limitations.
- The implementation of the approach in a tool, BERT-PLUGIN, that is integrated into a popular IDE, can be readily used by developers, and is freely available for download (http://www.cc.gatech.edu/~orso/software.html).
- A preliminary study that shows the potential usefulness of the approach when applied on different versions of a program.

The rest of the paper is organized as follows. Section II introduces an example that we use to show possible issues with traditional regression testing approaches and to illustrate our approach. Section III defines our behavioral regression testing approach. We discuss our implementation and our empirical evaluation in Sections IV-A and IV, respectively. Section V discusses related work. Finally, we conclude and sketch possible future research directions in Section VI.

II. Motivating Example

Before presenting the details of our approach, we introduce a small example that we use in the rest of the paper to show the limitations of existing regression testing approaches, motivate behavioral regression testing, and illustrate our approach. The example consists of a single class, `BankAccount`, which implements the main functionality of a bank account and that we assume to be part of a larger bank management system. Figures 1 and 2 show the code of two consecutive versions of the class.

Class `BankAccount` contains two methods: `deposit` and `withdraw`. Method `deposit` is the same in $V0$ and $V1$. It allows for depositing funds in the account. When called, the method first checks whether the deposit `amount` is positive. If so, it adds `amount` to field `balance` and returns `true`; otherwise, it leaves the account balance unchanged, prints an error message, and returns the value `false`.

Method `withdraw` allows for withdrawing funds from the bank account and is different in the two versions. In $V0$, the method first checks whether the withdrawal `amount` is negative. If so, it prints an error message and returns `false`. Otherwise, it checks the value of `balance`. If `balance` is negative, it reports that the account is overdraft and returns `false`. Conversely, if `balance` is positive, the method subtracts the `amount` from the account `balance` and returns a value `true`.

Assume that the developers decide to make the overdraft status of the account explicit. To this end, they make three changes to class `BankAccount`, which are shown in boldface font in Figure 2. First, they add a boolean field, `isOverdraft`, which keeps track of whether the account is in an overdraft state. Then, they modify the conditional at Line 9 of method `withdraw` so that it checks the value of field `isOverdraft` instead of `balance`. Finally, they add to method `withdraw` instructions to set `isOverdraft` to `true` if the balance becomes negative (Lines 13–14).

Although these changes to method `withdraw` are correct, there is a fault in the new version of the code. The developers forgot to reset the value of field `isOverdraft` when a deposit causes the balance to become positive after an overdraft. The effect of this omission fault is that an account that reaches an overdraft state will never leave it.

To identify the regression fault introduced in version $V1$ of `BankAccount`, a regression test suite would need to contain a test case that (1) performs a withdraw that causes the account to enter an overdraft state, (2) performs a deposit that

```
public class BankAccount {
  private double balance;

  public boolean deposit(double amount) {
01  if (amount > 0.00) {
02    balance = balance + amount;
03    return true;
    } else {
04    System.out.println("amount cannot be negative");
05    return false;
    }
  }

  public boolean withdraw(double amount) {
06  if (amount <= 0) {
07    System.out.println("amount cannot be negative");
08    return false;
    }
09  if (balance < 0) {
10    System.out.println("account is overdraft");
11    return false;
    }
12  balance = balance - amount;
13  return true;
  }
}
```

Figure 1. Version $V0$ of the bank account example.

```
public class BankAccount {
  private double balance;
  private boolean isOverdraft;

  public boolean deposit(double amount) {
01  if (amount >= 0.00) {
02    balance = balance + amount;
03    return true;
    } else {
04    System.out.println("amount cannot be negative");
05    return false;
    }
  }

  public boolean withdraw(double amount) {
06  if (amount <= 0) {
07    System.out.println("amount cannot be negative");
08    return false;
    }
09  if (isOverdraft) {
10    System.out.println("account is overdraft");
11    return false;
    }
12  balance = balance - amount;
13  if (balance < 0) {
14    isOverdraft = true;
    }
15  return true;
  }
}
```

Figure 2. Version $V1$ of the bank account example.

```
public void testBehavioralDifference() {
    BankAccount acc = new BankAccount();
    acc.deposit(10.00);
    acc.withdraw(20.00);
    acc.deposit(50.00);
    boolean result = acc.withdraw(20.00);
    assertEquals(result, true);
}
```

Figure 3. A test case that could reveal the regression fault introduced in version $V1$ of the bank account example.

causes the account to exit the overdraft state, (3) performs a withdraw with a positive amount, (4) checks whether the last withdraw was successful. Figure 3 shows a possible test case that would satisfy these requirements.

Although `BankAccount`'s regression test suite may contain such a test case, there is no specific reason why it should. For example, if the test suite was developed with a coverage goal in mind, 100% of `BankAccount`'s code can be covered with a set of simple test cases that do not include the one in Figure 3 (or any other test case that would reveal the fault). Moreover, `BankAccount` is a fairly simple example. The situation is only going to worsen for more realistic code and regression faults. As we discussed in the Introduction section, the test cases in the regression test suite may not exercise the modified behavior. For our example, the test suite may not exercise the specific sequence of method calls and corresponding parameter values required to expose the erroneous behavior of `BankAccount` $V1$. Even in the case where there are test cases in the regression test suite that exercise the erroneous behavior, the oracle associated with such test cases may be inadequate and fail in identifying such behavior. This case commonly occurs when test cases are generated in large quantities automatically, and the only cost-effective way to define an oracle is to use generic, and thus fairly inaccurate, ones. Considering again our example, a generic oracle would likely ignore the semantics of the code and simply check that the program under test does not throw an uncaught exception at runtime. (In the case of object-oriented languages, the oracle problem is further complicated by the presence of encapsulation and information hiding.)

In Section IV, we illustrate how the two key elements of our approach—change-centric automated generation of test inputs and focus on differential behavior—dramatically increase the likelihood of our approach to find regression faults such as the one in our example.

III. Automated Behavioral Regression Testing

Figure 4 provides a high-level view of our approach compared to traditional regression testing. In traditional regression testing (*e.g.,* [3]–[5]), an existing test suite ($T0$) defined for the old version of a program ($V0$) is run on the modified version of a program ($V1$). Non-obsolete test cases that, according to their oracle, fail on $V1$ and did not fail on $V0$ are reported to the developers as warnings that may indicate the presence of regression faults.

Automated *BEhavioral Regression Testing (BERT)* complements the traditional approach that we just discussed by improving regression testing along two main dimensions: (1) it generates a set of test inputs that are specifically targeted at the changed code, and (2) it explicitly leverages both the old and the new versions of the code. The result is a set of *behavioral differences* between the old and the new versions of the code. This information can provide developers with more and finer grained data on how their changes have affected the behavior of the code. Unexpected changes in the behavior, together with the detailed information about these changes, could help developers identify and remove regression faults. The scenario of use that we envision for BERT is one where the approach is integrated into the IDE

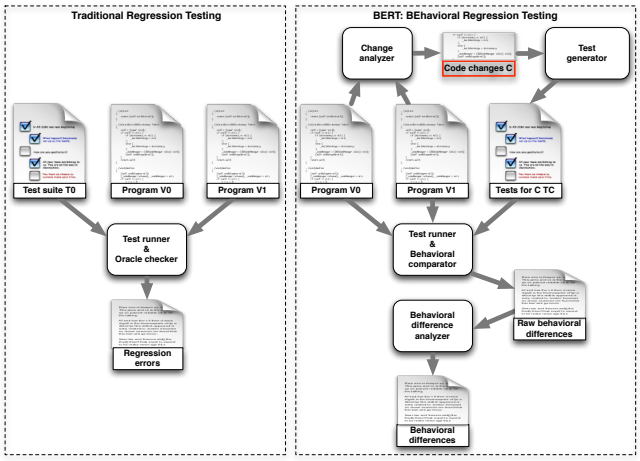

Figure 4. High-level view of our approach.

used by the developers and is activated every time the code is updated and compiled. (To realize this scenario, in our implementation we implemented BERT as a plug-in for Eclipse, a widely used IDE—see Section IV-A.) In such a scenario, the amount of changes in the code would typically be limited and localized.

BERT consists of three phases: generation of test inputs for changed code, behavioral comparison of original and changed code, and differential behavior analysis and reporting. We discuss these three phases in detail by referring to the overview of BERT provided in Figure 4. Because the specific characteristics of the programming language and environment targeted by the approach affect its definition, we define our approach for Java and assume tests to be encoded as JUnit tests. Although we focus on this context in our presentation, BERT should be generally applicable to other languages and types of tests.

A. Phase 1: Generation of Test Inputs for Changed Code

In the initial step of Phase 1, BERT collects change information by leveraging a *change analyzer* that takes as input the two versions of the program considered, $V0$ and $V1$, and produces a list of the classes that differ in the two versions. Because of the generality of this step, BERT can use different kinds of change analyzers, such as the ones typically provided by modern IDEs, specific differencing techniques (*e.g.,* [6]), or even a slightly modified version of the Unix `diff` utility.

BERT then generates a set of test inputs for the changed classes in $V1$ by feeding each of these classes to a *test generator*. As it was the case for the change analyzer, BERT can use any test generator that is able to automatically build test inputs for Java classes. Because the goal is to generate test inputs that cover as many behaviors as possible, the approach can even use multiple generators and just combine the set of generated tests (possibly after eliminating redundant tests). In addition, BERT also leverages any of the existing tests for $V0$ that exercise the changed code. (Such tests can be identified by collecting coverage information when tests are run on $V0$, another task that can be easily automated within an IDE.)

B. Phase 2: Behavioral Comparison of Original and Changed Code

In Phase 2, BERT first runs all of the tests generated in Phase 1 on their corresponding classes. For each changed class c and each test t for c, the *test runner* module runs t on the old and new versions of c, c_{v0} and c_{v1}.[3] After each call to a method m of c performed by t, BERT logs the following information:

State: BERT logs the state of the instances of c_{v0} and c_{v1} created and exercised by t, $inst\_c_{v0}$ and $inst\_c_{v1}$. To log the state, it retrieves the values of each field f in both $inst\_c_{v0}$ and $inst\_c_{v1}$ and stores them as $< seq\_id, m\_sig, name, value >$ tuples: seq_id is a unique (per version) id whose value is one for the first call and is increased for each subsequent call; m_sig is m's signature; $name$ is f's name, and $value$ is the value of f. If f is scalar, the value logged is the actual value of f in $inst\_c_{v0}$ and $inst\_c_{v1}$. If f is a reference to an object o, BERT logs the value of each of o's fields recursively until either a scalar field is encountered or a user-defined depth is reached. In this latter case, BERT stops its recursive descent and logs "null" if the reference is null, and "not_null" otherwise. (Note that we cannot log hash values here because they are not required to remain consistent across executions and may result in false positives.) Therefore, the greater the depth, the more precise—and more expensive—is the information collected and the comparison between different values.

Return values: BERT stores the value returned by m in the two cases as a $< seq\_id, m\_sig, value >$ tuple, where seq_id, m_sig, and $value$ are defined as in the previous case. If the method terminates with an exception, the value of the exception is stored as the return value.

Output: BERT also captures the output produced by the execution of m and stores it in the form $< seq\_id, m\_sig, destination, data >$, where seq_id and m_sig have the usual meaning, $destination$ is the entity where the output is sent (*e.g.,* a textual terminal, a network port, a graphical element), and $data$ is the raw data sent to that entity. In our current definition, for simplicity, we handle output produced only on standard output, on standard error, and on a set of graphical widgets (*i.e.,* text widgets). We propose possible ways to extend the definition in Section VI.

Distance: With every value logged after a call to m, BERT also records the shortest *distance* between m and any changed method in the dynamic call graph induced by t. For

[3]Note that it may not be possible to run all test inputs created for c_{v1} on c_{v0} (*e.g.,* due to changes to the class's interface). These cases are fairly uninteresting because they provide information that could be discovered through static differencing. We therefore discard such tests.

example, if m is a changed method, such distance is zero; if m invokes a changed method directly, the distance is one; if m invokes a method that in turn invokes a changed method, the distance is two; and so on.

When t's execution terminates and the data logs are produced, BERT's *behavioral comparator* accesses the logs for $inst\_c_{v0}$ and $inst\_c_{v1}$ and compares states, return values of corresponding calls, and outputs collected for the two versions of the class. For each difference that it finds, BERT records the fact that there was a difference and a set of relevant data for differences of that type. For all differences, BERT records what were the different field, return, or output values in the old and new versions and what was their distance from a code change. For output differences, it records also the destination(s) on which different output was produced. In addition, each of the recorded changes is tagged with a unique identifier for t, which allows to map individual changes to the test case that revealed them.

After executing all of the tests generated in Phase 1 on all of the changed classes, the result is a set of zero or more *raw behavioral differences* for each class. Each behavioral difference consists of a state, return value, or output difference together with its context information, as discussed earlier.

C. Phase 3: Differential Behavior Analysis and Reporting

Phase 3 analyzes and manipulates the set of differences produced in the previous phase to group and order them, so as to allow developers to better consume the information produced by BERT. To achieve this goal, BERT's *behavioral difference analyzer* first ranks or filters them based on their likelihood to represent a regression fault. It then abstracts away some of the information contained in the raw differences and reduces redundancy within the set of identified differences.

First, BERT divides the set of differences into classes based on their distance value, so that differences with the same distance are in the same class. It then groups changes within a class as follows.

For state-related differences, the analyzer groups all differences that involve the same method and field as a single behavioral difference involving that method and field. It also associates such behavioral difference with the set of test inputs that reveal each individual difference. Information on the individual tuples of different values for the field in $inst\_c_{v0}$ and $inst\_c_{v1}$ are maintained separately as possible additional information for the developers.

Similarly, for differences related to return values, BERT groups all differences involving calls to the same method as a single behavioral difference associated with the set of test inputs that reveal the individual differences. Also in this case, the individual value differences are stored separately for possible further analysis.

Also similarly, BERT groups all output-related differences that involve calls to the same method and are sent to the same destination as a single behavioral difference associated with the set of test inputs that reveal the individual differences. Again, the individual value differences are stored separately for possible further analysis.

The overall result of this phase is a set of behavioral differences between c_{v0} and c_{v1} that includes (1) which fields can have different values in c_{v0} and c_{v1} and which test inputs and method calls can cause such differences to manifest; (2) which methods can return different values in c_{v0} and c_{v1} and which test inputs can cause such differences to manifest; (3) which differences in (textual) output can occur between c_{v0} and c_{v1} on the terminal and graphically, and which test inputs and method calls can reveal them.

BERT reports these behavioral differences to the developers ranked in an order that is inversely proportional to their distance value (*i.e.,* with the differences with greater distance at the top). BERT can also filter out reports below a given distance based on the total number of reports. The intuition and rationale behind this ranking and filtering is that behavioral differences that occur at a greater distance from an actual change are less likely to be intentional than behavioral differences that occur closer to a change. This intuition is confirmed by the results of our empirical evaluation, as discussed in Section IV.

Developers can use this information to assess which of these differences may indicate the presence of a regression fault and which instead are expected given the changes that the developers performed on the code. If the developers identify regression faults, they can then use the test inputs associated to the corresponding behavioral differences to investigate and eventually eliminate the faults.

D. Limitations

Non-deterministic methods in the program under test can cause false positives in the report produced by BERT, as the execution of the same test on two versions of a non-deterministic method can produce different results that are not caused by code changes. (Even running the same test on a single version of a non-deterministic method twice can produce different test outcomes.) For example, methods returning pseudo-random values or methods returning the current time fall into this category. Currently, BERT allows developers to provide a list of methods that should be excluded by our analysis, and this mechanism can be used to exclude knowingly non-deterministic methods. In future work, we plan to extend BERT to automatically exclude at least some of these methods—by running each test multiple times on a method and discarding the information for that method if at least one execution behaves differently.

Another limitation of our approach is that it is currently only applicable to changes that do not involve an interface change, that is, the names and signatures of public methods

remain the same between two subsequent versions. (In the presence of interface changes, the test inputs developed for the new version of the code may not run on the old version.) In this first attempt at behavioral regression testing, we decided for now to limit the applicability of BERT to such cases. In future work, we plan to investigate several ways to extend the approach and be able to handle interface changes. One possible way would be to adapt the tests so that they run on both versions by eliminating and/or adding some parameters, depending on the changes in the interface. (Because we do not need explicit oracles for our tests, these test inputs may still provide relevant information about the differential behavior of two versions.) Another way would be to apply recent interface adaptation techniques (*e.g.,* [7]).

Finally, a possible limitation of our approach is that it is designed to work on localized changes involving one class (or a few classes) and may not be ideal in the case of extensive changes (*e.g.,* changes that add a significant amount of new functionality to a system). Because our approach is applied every time a developer saves a compilable class, however, even large changes would be regression-tested in small increments. We plan to perform further experiments to assess whether our approach needs to specifically handle the case of a large set of changed classes that become compilable only when the last one is saved. If that is the case, we may decide to simply avoid running our approach for such changes and let the developer test the new functionality with traditional testing approaches (which should be conducted anyway, no matter whether our approach is applied or not).

IV. Evaluation of BERT

To assess the feasibility and usefulness of our approach, we implemented it in a prototype tool and used the tool to perform two empirical studies: a proof of concept study on our example described in Section II and a more extensive evaluation on a real subject. More specifically, we investigated the following research question: *Is BERT able to reveal regression faults in a new version of the code automatically and without generating too many false positives?* In the rest of this section, we discuss our implementation of BERT-PLUGIN, present our two studies, and discuss their results.

A. Implementation of BERT-PLUGIN

We have implemented our proposed BERT approach as an Eclipse plug-in, called BERT-PLUGIN, that operates transparently as developers edit and evolve their code. We concisely describe BERT-PLUGIN by discussing how it implements the different parts of our approach, presented in Section III and depicted graphically in Figure 4.

Phase 1's modules: To implement the *change analyzer* module, we leverage two features of Eclipse—the ability to intercept events and the availability of change information between two versions of a project (or parts thereof). More precisely, BERT-PLUGIN intercepts *successful* compilation events to be able to perform its analysis each time some part of a project has been modified, saved, and compiled. When triggered by one such event, BERT-PLUGIN compares the previous and the new versions of the project using the functionality provided by Eclipse through its API. The result of this step is a list of modified classes in the project.

The *test generator* module relies on Randoop [8] and CodePro Server (http://www.instantiations.com/codepro) for test input generation. We chose these tools because they can generate test inputs for single classes and have the advantage of automatically building the scaffolding needed for the tests, such as drivers and stubs (*i.e.,* mock objects) and producing readily usable JUnit tests. (Note that we do not require the JUnit test suite to be equipped with assertions, as we are interested only in differential behavior.) For reusing existing tests, the *test generator* module leverages the functionality provided by Eclipse's JUnit plug-in—for identifying and rerunning tests associated with a project—and EclEmma—for collecting coverage information and associating test inputs with the classes that they exercise.

Phase 2's modules: To collect the information needed for behavioral comparison, BERT-PLUGIN instruments the JUnit tests and the code under test before running the tests. The instrumentation is performed using Javassist (http://www.csg.is.titech.ac.jp/~chiba/javassist), a bytecode rewriting library written in Java. For each test t, BERT-PLUGIN first identifies the set of *relevant objects* for t, that is, instances of changed classes created (and therefore being tested) by t. It then instruments every call performed by t on a method m of a relevant object o by adding probes that store (1) o's state after the execution of m and (2) m's return value, as described in Section III-B. Finally, BERT-PLUGIN instruments m with two kind of probes: (1) probes that intercept all (textual) output produced by m and suitably log such output together with the destination where the output is sent; and (2) probes that keep track of the call stack and measure the distance as defined in Section III-B.

After instrumenting the code and the tests, BERT-PLUGIN's *test runner* module runs both the existing and newly created tests by also leveraging the JUnit plug-in in Eclipse, which allows for rerunning all or part of the JUnit tests for a given project. As the tests run, the probes added through instrumentation collect the information needed by BERT-PLUGIN to perform behavioral comparison and store such information persistently on the file system in XML format. The use of XML allows for easily saving and reloading values even when they consist of hierarchical records (*i.e.,* when collecting non-scalar values using a depth that is greater than one—see Section III-B for details).

The *behavioral comparator* module is implemented in Java and simply (1) reads the XML files produced by the test runner module described earlier, (2) compares the values for corresponding calls, and (3) produces a set of raw behavioral differences. To read and write XML files, BERT-

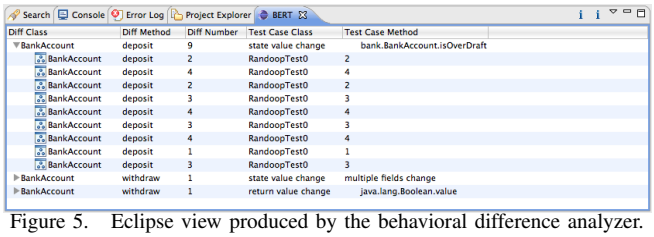

Figure 5. Eclipse view produced by the behavioral difference analyzer.

PLUGIN uses the SAXParser in the standard Java library (`javax.xml.parsers.SAXParser`).

Phase 3's modules: The *behavioral difference analyzer* module is also implemented in Java and performs the grouping and ordering described in Section III-C. The module also provides visualization capabilities for displaying in an Eclipse view the set of grouped and ordered differences. Figure 5 shows the view obtained when using BERT-PLUGIN on our `BankAccount` example. Clicking on any of the behavioral difference groups shows the individual differences in that group, and clicking on an individual difference would show the actual delta between the values involved in the difference, in XML format (see Figure 8 for an example).

B. Empirical Studies

1) Study 1: Proof of Concept:

Setup: In our proof of concept study, we performed an initial assessment of the feasibility of our approach. To this end, we applied BERT-PLUGIN to the example presented in Section II. We loaded the version $V0$ of the `BankAccount` code into Eclipse and saved it, causing the code to be compiled as well, and BERT-PLUGIN to be triggered as a consequence. Since $V0$ was the first version being created, BERT-PLUGIN simply stored the version and did not perform any analysis. We then applied the changes shown in Figure 2 to go from $V0$ to $V1$ and saved the new version $V1$, leading again to the compilation of the code and the invocation of BERT-PLUGIN. This time, however, BERT-PLUGIN realized that $V1$ was a new version and performed the three phases of its analysis. After identifying that `BankAccount` (the only class in the project in this case) had changed, BERT-PLUGIN's test generator generated a set of test inputs for version $V1$ of the class. Overall, 2,569 test inputs were generated. Each test input consisted of pseudo-random method sequences with pseudo-random method arguments.

At this point, BERT-PLUGIN ran each test input on both versions of `BankAccount`, while logging state, return value, and output information. (It is worth noting that executing the complete set of test inputs on `BankAccount` takes less than a second.) BERT-PLUGIN then performed the comparison of the recorded logs and suitably generated the set of differences for the two versions of the class.

Results: The results of the comparison were encouraging: about 60% of the automatically generated test inputs (1,557 out of 2,569) were able to reveal the behavioral difference that indicates the regression fault in the example,

```
public void testclasses3() throws Throwable {
01   BankAccount var0 = new BankAccount();
02   double var1 = (double)1.0;
03   boolean var2 = var0.deposit((double)var1);
04   double var3 = (double)2.0;
05   boolean var4 = var0.withdraw((double)var3);
06   double var5 = (double)1.0;
07   boolean var6 = var0.deposit((double)var5);
08   double var7 = (double)2.0;
09   boolean var8 = var0.withdraw((double)var7);
}
```

Figure 6. An example of test input for the bank account example.

and no false positives were reported. Figure 6 shows one of the automatically generated test inputs that reveal the problem. As the figure shows, the test exercises the fault-revealing sequence discussed in Section II.

In all these cases, the behavioral difference was identified automatically and manifested itself in two ways: some calls to method `withdraw` returned two different values in the two versions and produced some output only in the new version of the code. For illustration, consider again the test in Figure 6. For that test, the last call to `withdraw` would return `true` and produce no output in version $V0$ of `BankAccount`, whereas it would return `false` and produce the output "account is overdraft" in version $V1$. Note that the prototype did not report any state-related behavioral difference because of the presence of the new field `isOverdraft` in $V1$. Since the addition or removal of a field is almost always intentional, BERT-PLUGIN identifies only state differences that involve fields that are present in both versions of a class.

We stress that the successful identification of the erroneous behavior, which would easily reveal the corresponding regression fault, is due to the two key characteristics of BERT: the automatic generation of a large number of test inputs for the changed code and the use of automatically identified detailed behavioral differences.

2) Study 2: Real Program:

Setup: Although Study 1 provides some initial evidence of BERT's usefulness, it targets a small example. Therefore, in our second study, we further investigate the effectiveness and precision of our approach by targeting many versions of a real application: a Java library called Joda-Time (http://joda-time.sourceforge.net/). Joda-Time extends the functionality of date- and time-related classes in the standard Java library. The SVN repository of Joda-Time contains a large number of versions that correspond to the whole history of the library and that is still being actively updated. Because the initial history of the library involves for the most part addition of functionality, rather than evolution and changes to existing code, we focused on relatively mature versions of the code. To identify such versions, we simply checked through sampling the point in the history where the addition of new classes dropped and remained low for several subsequent versions.

Starting from that point in the SVN history, we extracted all versions that satisfy BERT's assumption of not having interface changes between two versions (see Section III-D).

```
//r916:
private transient YearInfo[] iYearInfoCache;
private transient int iYearInfoCacheMask;

//r917:
private static final int CACHE_SIZE = 1;
private static final int CACHE_MASK = CACHE_SIZE - 1;
private final YearInfo[] iYearInfoCache =
                            new YearInfo[CACHE_SIZE];
```

Figure 7. Changes between versions r916 and r917.

```
V1/out-TestSerialization.testSerializedBuddhistChronology
    .2.readObject.state
<className>org.joda...BuddhistChronology</className>
  < <field>
      ...

V2/out-TestSerialization.testSerializedBuddhistChronology
    .2.readObject.state
<className>java.io.NotSerializableException</className>
```

Figure 8. Behavioral difference report for version pair r916–r917.

The final result was a set of 54 pairs of version of Joda-Time. We applied BERT-PLUGIN to each pair of versions, similarly to what we did for the two versions of our example in Study 1: we loaded the first version of the pair into Eclipse, applied the changes to obtain the second version of the pair, and let BERT-PLUGIN perform its three phases. At the end of this process, we obtained a change report, in the form of the one shown in Figure 5, for each version pair. We then manually investigated the reports to determine whether the behavioral differences identified by BERT-PLUGIN actually corresponded to regression faults in the code.

Results: The reports generated by BERT-PLUGIN for the considered 54 version pairs contained 36 behavioral differences. In particular, BERT-PLUGIN detected no behavioral difference for 21 version pairs, 1 behavioral difference each for 30 version pairs, and 2 behavioral differences each for the remaining 3 version pairs. By using the information on bug fixes in the CVS repository, we were able to confirm that one of the behavioral differences was without doubt a true regression fault. We next discuss this fault in detail and then consider the other differences.

The *true regression fault* is caused by the changes made when going from version r916 to version r917 in the SVN repository. Figure 7 shows the relevant part of the change.

The change in class `BaseGJChronology` is meant to enhance the performance of some critical methods. The culprit that causes the regression fault is the modification of `YearInfo`'s declaration from `transient` to `final`. Because `YearInfo` is a non-serializable class (*i.e.,* it does not implement the `Serializable` interface), this modification affects the ability to serialize the whole `BaseGJChronology` class: objects of type `BaseGJChronology` can be serialized in version r916, but can no longer be serialized in version r917. (Note that transient fields are excluded from serialization, whereas final fields are not.) Figure 8 shows parts of the XML content of the behavioral difference report for this change.

In the report, `SerializedBuddhistChronology` is a subclass of `BaseGJChronology`, thereby inheriting the fields declared

```
public long set(long instant, int value) {
   int min = getMinimumValue(instant);
//r1216:
   if (value >= min && value < getMaximumValue(instant)) {
//r1217:
   if (value >= min && value <= getMaximumValue(instant)) {
       return super.set(instant, value);
   }
   return add(super.set(instant, min), value - min);
}
```

Figure 9. Changes between versions r1216 and r1217.

```
V1/out-LenientDateTimeField.set.retval.testNearDstTrans.10
<f_value>1162188000000</f_value>
V2/out-LenientDateTimeField.set.retval.testNearDstTrans.10
<f_value>1162191600000</f_value>
```

Figure 10. Behavioral difference report for version pair r1216–r1217.

in `BaseGJChronology`. This report shows a difference in behavior between the two versions of the class when we execute the second call (indicated by ID ".2.") to method `SerializedBuddhistChronology.readObject` within test `testSerializedBuddhistChronology`. In the second version, the call results in a `NotSerializableException` exception, whereas the execution of the same test throws no exception in the first version. The group of differences that include the one listed in Figure 8 also include similar behavioral differences for classes `SerializedCopticChronology`, `SerializedGJChronology`, `SerializedGreorianChronology`, and `SerializedJulianChronology`, which are also subclasses of `BaseGJChronology`.

Based on inspection of the report(s) produced by BERT and the code, it looks clear that (1) the behavioral difference is indeed a regression fault and (2) it would be fairly straightforward to go from the difference to the actual fault in the code and fix it.

From the comments on bug fixes in the SVN repository, we found that the true regression fault detected by BERT on version r917 was indeed fixed in a subsequent version, and the commit-time difference between the two versions was three days. This is one case where the use of BERT would have likely allowed the developers to find and fix the problem before even committing their changes.

For the remaining behavioral differences reported by BERT-PLUGIN on Joda-Time, we were not able to identify information in the SVN logs that clearly indicated whether the reported differences correspond to true regression faults. We therefore inspected all of the reported differences manually, which is a difficult task; in many cases, it is not clear what the developers' intention was in introducing a change, and it is therefore not possible to classify a behavioral difference as either a true or a false positive. Here, we discuss one of the cases for which we were able to classify a reported difference as a false positive. The behavioral difference is related to the changes between versions r1216 and r1217, shown in Figure 9.

In this example, developers changed the `set` method in the `LenientDateTimeField` class. By looking at the SVN logs,

we discovered that the reason for this change is to fix a fault in class `LenientChronology`, which might incorrectly adjust a valid hour field near a daylight-saving transition. The behavioral difference reported by BERT-PLUGIN, shown in Figure 10, indicates that BERT detects a difference in the return value of method `LenientDateTimeField.set`. This difference is caused by the bug fix and is thus not a regression fault.

Although this example indicates that BERT-PLUGIN can produce false positives, two things can be noted about the report. First, the developer who just fixed the fault would likely recognize immediately that the behavioral difference can be ignored and would not waste time on it. Second, the distance for the difference is zero in this case, that is, the behavioral difference manifests itself in the method that was changed. As we discussed in Section III-B, we expect reports with distance zero to be considerably more likely to represent expected behavioral changes.

Considering all of the 36 behavioral differences reported by BERT-PLUGIN, 22 are at distance 0, 10 at distance 1, and the remaining 4 at distances greater than 1. We analyzed these four behavioral differences by looking at the code and the SVN logs: one corresponds to the true regression fault discussed earlier (versions r916–r917); one is a false positive; and the remaining two could not be classified either way with the information available to us. In other words, if we filtered reports with distance lower than 2, the developers would be presented with only four reports, of which one represents a true regression fault, which we believe to be an encouraging result. Moreover, ranking could further help focusing the developers' attention on potentially more useful reports—the difference corresponding to the true regression fault is ranked second based on distance among all reports.

To get further evidence of the usefulness of BERT, we also examined the 21 changes that did not result in any behavioral difference. After inspecting the code locations of these changes and the SVN logs, we found that most of these code changes are related to refactoring for improving the design of the project, and the corresponding code was not further changed in subsequent versions. This is also encouraging, as it shows that BERT-PLUGIN did not report differences for cases where there should have not been.

V. Related Work

The Diffut approach [9] exploits method preconditions and postconditions to enable synchronized execution of two versions ($V0$ and $V1$) of a class and compare the behavior of the two versions. Diffut suffers from a number of limitations. For example, corresponding classes from $V0$ and $V1$ with the same package and class name cannot be executed in the same Java Virtual Machine, as required by Diffut, and system outputs from the two versions cannot be captured or compared by Diffut. In contrast, our approach does not suffer from these limitations because it runs the same test inputs separately on two versions and compares the captured data from two versions offline. In addition, our approach is also flexible enough to allow for incorporating heuristics for filtering out intended behavioral differences, such as addition, deletion, or renaming of object fields.

Evans and Savoia [10] proposed a differential testing approach in which they generate a test suite for each of the two given versions of a software system ($V0$ and $V1$); assertions are synthesized in the generated test suite to assert the captured behavior of the version [11]. Assume that the generated test suites for the two versions $V0$ and $V1$ are $T0$ and $T1$, respectively. Their approach then runs test suite $T0$ on $V1$ and test suite $T1$ on $V0$. Our approach does not synthesize or add assertions in the generated test inputs, but runs the same test inputs on both versions while capturing and comparing data related to the execution of each version. The behaviors being compared by our approach are more detailed than the ones targeted by the approach proposed by Evans and Savoia, which does not compare program outputs and compares receiver-object states only in a limited way. Therefore, our approach is likely to provide better regression-fault detection capability.

Some existing capture and replay approaches [12]–[14] capture the inputs and outputs of the unit under test during the execution of system tests. These approaches then replay the captured inputs for the unit as less expensive unit tests, where the outputs of the unit are checked against the captured outputs. Different from these existing approaches, our new approach captures runtime behavior of the execution of automatically generated new unit tests, which exercise behaviors that are not necessarily exercised by system tests. However, our approach can also be used in combination with these previous approaches by comparing the behaviors of the unit tests generated by such approaches.

Sometimes the quality of the existing tests might not be good enough to cause the outputs of two program versions to be different and expose behavioral differences between them. Some previous regression test generation approaches [15]–[17] try to generate new tests to expose such behavioral differences. These approaches complement our approach and can be integrated with our approach as third-party test-generation tools.

Santelices and colleagues [18] use data and control dependence information, along with state information gathered through partial symbolic execution of the old and new version of a program, to help developers augment an existing regression test suite. Their approach does not automatically generate any test input, but simply provides guidelines for developers on how to improve an existing regression test suite. Our approach not only generates unit tests, but also compares the behaviors captured while running the generated unit tests on multiple versions.

VI. Conclusion and Future Work

We have presented a novel regression testing approach—BEhavioral Regression Testing (BERT)—that is based on automatically identifying behavioral differences between two versions of a program through dynamic analysis. BERT consists of three main phases: (1) generating a large number of test inputs for the changed parts of the code, (2) running the generated test inputs on the old and new versions of the code and identifying differences in the tests' behavior, and (3) analyzing the identified differences and presenting them to the developers. Our approach has two key aspects that distinguish it from traditional regression testing. First, it focuses on a small subset of the code, which allows it to generate a more thorough set of tests. Second, it leverages differential behavior, which eliminates the need for developer-provided oracles. Because of these novel aspects, BERT can give developers more (and more detailed) information than traditional regression testing approaches. Our preliminary evaluation of BERT provides initial evidence of its usefulness: for the cases considered, BERT was able to identify true regression faults while generating false positives that could be filtered out or ranked at low priority.

We believe that these initial results, albeit still preliminary, are encouraging and motivate further research along several directions. First, we plan to perform a more extensive empirical evaluation of BERT. Second, we plan to investigate the use of finer-grained differencing techniques (*e.g.,* [6]) to further reduce the scope of the test generation portion of BERT. Moving to the method or even code-fragment level might allow for an increasingly thorough testing of the changes. Third, we plan to explore the use of test generation techniques that are guided by the characteristics of the identified changes, rather than being based on mainly random generation. Fourth, we plan to investigate different ways to handle interface changes, as described in Section III-D. Finally, we plan to explore more aggressive ways to cluster, filter, and abstract changes before presenting them to the developers. In this context, we may also be able to leverage bug isolation techniques targeted at specific parts of the code (*e.g.,* [19]) to further reduce the developers' inspection efforts. We also plan to investigate filtering of the behavioral differences based on diversity metrics other than distance of dynamic call graph (*e.g.,* [20]), which may provide more accurate feedback to the developers.

Acknowledgments

This work was supported in part by NSF awards CCF-0725202 and CCF-0916605 to Georgia Tech and CCF-0725190 to NC State University.

References

[1] A. Orso and T. Xie, "BERT: BEhavioral Regression Testing," in *Proc. WODA*, 2008, pp. 36–42.

[2] E. J. Weyuker, "On testing non-testable programs," *Compuer Journal*, vol. 25, pp. 465–470, 1982.

[3] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," in *Proc. ICSE*, 1998, pp. 188–197.

[4] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test case prioritization," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, October 2001.

[5] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *Proc. FSE*, 2004, pp. 241–252.

[6] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *Proc. ASE*, 2004, pp. 2–13.

[7] D. Dig, S. Negara, V. Mohindra, and R. Johnson, "ReBA: Refactoring-aware binary adaptation of evolving libraries," in *Proc. ICSE*, 2008, pp. 441–450.

[8] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *OOPSLA Companion*, 2007, pp. 815–816.

[9] T. Xie, K. Taneja, S. Kale, and D. Marinov, "Towards a framework for differential unit testing of object-oriented programs." in *Proc. AST*, 2007, pp. 5–11.

[10] R. B. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *Proc. ESEC/FSE*, 2007, pp. 549–552.

[11] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *Proc. ECOOP*, 2006, pp. 380–403.

[12] A. Orso and B. Kennedy, "Selective capture and replay of program executions," in *Proc. WODA*, 2005, pp. 29–35.

[13] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for Java," in *Proc. ASE*, 2005, pp. 114–123.

[14] S. Elbaum, H. N. Chin, M. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proc. FSE*, 2006, pp. 253–264.

[15] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.

[16] K. Taneja and T. Xie, "DiffGen: Automated regression unit-test generation," in *Proc. ASE*, 2008, pp. 407–410.

[17] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Guided path exploration for regression test generation," in *Companion ICSE, NIER*, 2009, pp. 311–314.

[18] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *Proc. ASE*, 2008, pp. 218–227.

[19] A. Orso, S. Joshi, M. Burger, and A. Zeller, "Isolating relevant component interactions with JINSI," in *Proc. WODA*, 2006, pp. 3–9.

[20] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, "Searching for cognitively diverse tests: Towards universal test diversity metrics," in *Proc. ICST Workshop*, 2008, pp. 178–186.