BERT: BEhavioral Regression Testing

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

Tao Xie
Department of Computer Science
North Carolina State University
xie@csc.ncsu.edu

ABSTRACT

During maintenance, it is common to run the new version of a program against its existing test suite to check whether the modifications in the program introduced unforeseen side effects. Although this kind of regression testing can be effective in identifying some change-related faults, it is limited by the quality of the existing test suite. Because generating tests for real programs is expensive, developers build test suites by finding acceptable tradeoffs between cost and thoroughness of the tests. Such test suites necessarily target only a small subset of the program's functionality and may miss many regression faults. To address this issue, we introduce the concept of behavioral regression testing, whose goal is to identify behavioral differences between two versions of a program through dynamic analysis. Intuitively, given a set of changes in the code, behavioral regression testing works by (1) generating a large number of test cases that focus on the changed parts of the code, (2) running the generated test cases on the old and new versions of the code and identifying differences in the tests' outcome, and (3) analyzing the identified differences and presenting them to the developers. By focusing on a subset of the code and leveraging differential behavior, our approach can provide developers with more (and more focused) information than traditional regression testing techniques. This paper presents our approach and performs a preliminary assessment of its feasibility.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Verification.

Keywords: Regression testing, software evolution, dynamic analysis

1. 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 did not affect the unchanged code in unforeseen ways. To this end, developers typically rerun, completely or in part, a set of existing test cases (*i.e.,* a regression test suite) on P'. If one or more of the test cases that executed correctly on P cause an unexpected[1] 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.

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 cases, developers often settle for approximated oracles that perform only partial checks of the outcome of a test [27]. In fact, it is common to consider crashes (or exceptions) as de-facto oracles, even though they capture only a small subset of the possible erroneous behaviors of a program.

In summary, 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 cases 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 techniques. 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 cases targeted at each of the changed classes. In its *second phase*, BERT considers each changed class c and each test case t

WODA – Workshop on Dynamic Analysis, July 21, 2008

[1]Developers would expect some of the existing test cases to fail based on the changes that they performed on the code. These test cases, which are normally called *obsolete test cases*, would either be discarded or modified to run on the new code.

```
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.

created for c, runs t on the old and new versions of c, and compares the outcome of t in the two cases. The technique 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 cases). 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 techniques and enable it to provide developers with more information than such traditional techniques. By focusing on the (typically small) subset of the code that has changed, our approach can address the first limitation of existing techniques: the lack of test cases that can adequately exercise the differences in behavior between P and P'. And by leveraging differential behavior, 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.

We also present a proof-of-concept assessment of BERT performed by applying the approach to an example and examining the feedback provided by BERT to the developer. Although our results are too preliminary to draw any conclusion on the general effectiveness of the approach, they show that BERT has the potential to produce useful information for developers. Such information can either give developers confidence that the changed code behaves as intended or point them to potential issues in the code.

The main contribution of this paper is 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 rest of the paper is organized as follows. Section 2 introduces an example that we use to show possible issues with traditional regression testing techniques and to illustrate our behavioral regression testing approach. Section 3 defines our approach. We present our preliminary assessment of the approach in Section 4 and discuss related work in Section 5. Finally, we conclude and sketch possible future research directions in Section 6.

2. MOTIVATING EXAMPLE

Before presenting the details of our technique, 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 practical effect of this omission fault is that an account that reaches an overdraft state will never leave it.

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

```
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 `BankAccount`.

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, this 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, 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 is commonly the case 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 application does not generate an 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 4, we illustrate how the two key elements of our approach—change-centric automated generation of test cases and focus on differential behavior—dramatically increase the likelihood of our approach to find regression faults such as the one in our example.

3. BEHAVIORAL REGRESSION TESTING

Figure 4 provides a high-level view of our approach compared to traditional regression testing. In traditional regression testing (*e.g.,* [10, 16, 20, 25]), 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 *regression errors*—failures that may indicate the presence of regression faults.

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 cases 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 code. This information would 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, would help developers identify and remove regression faults. The scenario of use that we envision of BERT is one where the technique is integrated into the IDE used by the developers and is activated every time the code is updated and compiled. Therefore, the amount of changes in the code would typically be limited and localized.

BERT consists of three phases: generation of test cases 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 technique affect its definition, we define our technique for Java and assume test cases to be encoded as JUnit [9] test cases (*i.e.,* each test case for a given class c creates an instance of c, invokes one or more methods of c on that instance, and performs some checks on the test outcome). Although we focus on this context in our presentation, BERT should be generally applicable to other languages and types of test cases.

3.1 Phase 1: Generation of Test Cases for Changed Code

In the initial step of this phase, 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.,* [1]), or even a slightly modified version of the Unix `diff` utility. In our current implementation, we use the change information provided by the Eclipse IDE[2] through its API.

BERT then generates a set of test cases 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 cases for Java classes. Because the goal is to generate test cases that cover as many behaviors as possible, the technique can even use multiple generators and just combine the set of generated test cases (possibly after eliminating redundant tests). Our current implementation of BERT relies on Agitar's JUnit Factory [12] and Randoop [21] for the test case generation part. We chose these tools because they are fairly effective in generating test cases for single classes and have the advantage of automatically generating the scaffolding needed for the test cases, such as drivers and stubs (mock objects).

3.2 Phase 2: Behavioral Comparison of Original and Changed Code

In this phase, BERT first runs all of the test cases generated in Phase 1 on their corresponding classes. For each changed class c and each test case t for c, the *test runner* module runs t on the old and new versions of c, c_{v0} and c_{v1},[3] while logging the following information:

[2] `http://www.eclipse.org`

[3] Note that it may not be possible to run all test cases created for c_{v1} on c_{v0} (*e.g.,* due to changes to the class's interface). These cases are

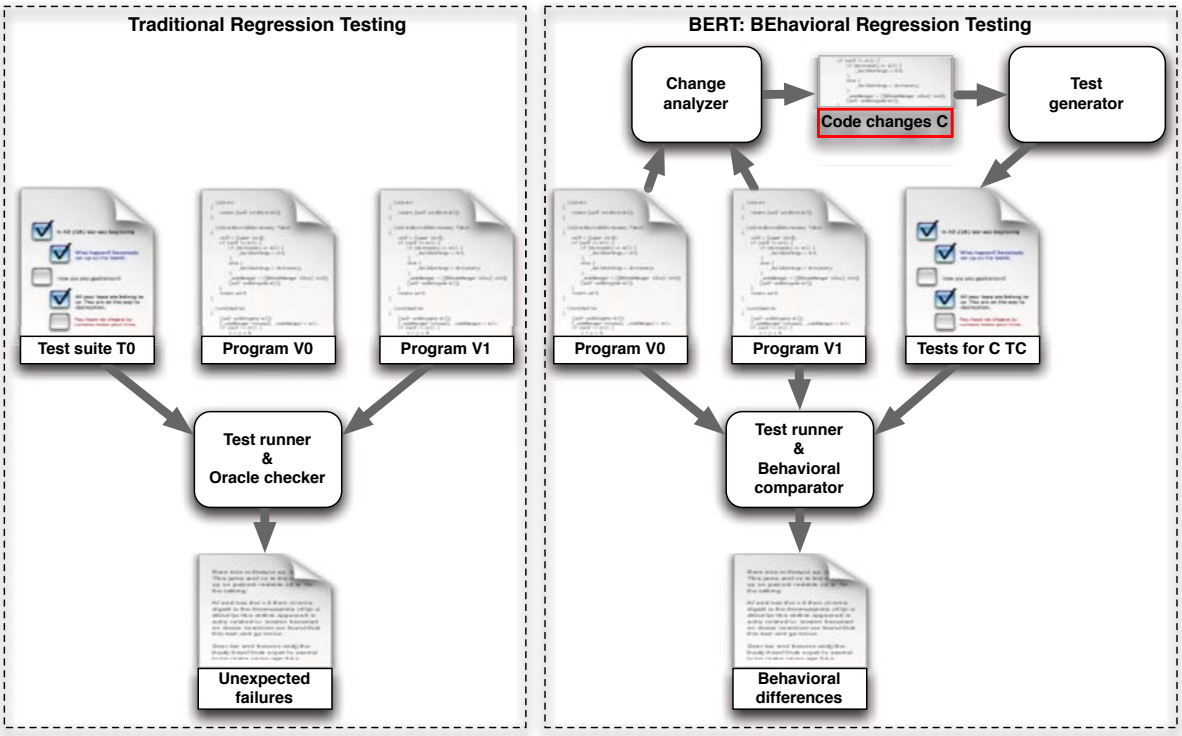

Figure 4: High-level view of our approach.

State: At the end of test t, 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 do this, it retrieves the values of each field f in both $inst\_c_{v0}$ and $inst\_c_{v1}$ and stores them as $< name, value >$ pairs, where $name$ is f's name, and $value$ is the value of f. The values logged are either the actual values of f in $inst\_c_{v0}$ and $inst\_c_{v1}$, if f is scalar, or its hash values, if f is an object reference.

Return values: For each method m of c invoked by t on $inst\_c_{v0}$ and $inst\_c_{v1}$, BERT stores the value returned by m in the two cases as a $< seq\_id, m\_sig, value >$ tuple. In each tuple, seq_id is a unique (per version) id whose value is one for the first call and is increased for each following call; m_sig is m's signature; $value$ is again either an actual value or a hash value, depending on m's return type (scalar or object).

Output: While running t on $inst\_c_{v0}$ and $inst\_c_{v1}$, BERT captures the output produced by execution of the test and stores it in the form $< destination, data >$, where $destination$ 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, concatenated in the common case where multiple output is sent to the same 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 6.

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 each state difference, BERT records which field was different and what were the different values in the old and new versions. Similarly, for each difference in return values, it records the signature of the method involved and the different values returned in the two versions. Finally, for output differences, it records the destination(s) on which different output was produced and the difference between the two outputs. Each of the recorded changes is also 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 test cases 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 above.

3.3 Phase 3: Differential Behavior Analysis and Reporting

This phase analyzes and manipulates the set of differences produced in the previous phase to simplify and refine them and allow developers to better consume the information produced by BERT. To achieve this goal, BERT's *behavioral differences analyzer* tries to abstract away some of the information contained in the raw differences and to reduce redundancy within the set of identified differences. For state-related differences, the analyzer groups all differences that involve the same field as a single behavioral difference involving that field. It also associates such behavioral difference with the set of test cases that reveal each individual difference. Information on the individual pairs of different values for the field in $inst\_c_{v0}$ and $inst\_c_{v1}$ are maintained separately as possible additional information for the developer.

fairly uninteresting because they provide information that could be discovered through static differencing. We therefore discard such test cases.

Analogously, 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 cases that reveal the individual differences. Also in this case, the individual value differences are stored separately for possible further analysis.

The process is different for output-related differences. Because the current incarnation of BERT considers only text-related output, the only grouping performed is for aggregating differences in output directed to graphical widgets. That is, multiple differences in the output directed to the GUI are grouped as a generic group output difference.

The overall results of this phase is therefore 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 cases can cause such differences to manifest; (2) which methods can return different values in c_{v0} and c_{v1} and which test cases 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 cases reveal them.

BERT reports these behavioral differences to the developers, who 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 the developers performed on the code. If the developers identify regression faults, they can then use the test cases associated to the corresponding behavioral differences to investigate and eventually eliminate the fault.

4. EXPERIENCE

To perform an initial assessment of the feasibility of our approach, we applied it to the example that we presented in Section 2. We developed a proof-of-concept prototype of BERT that provides a partial implementation of the technique. Currently, the prototype takes as input two versions of a class, generates test cases for the newest of the two versions, runs the generated test cases on the two versions and collects raw behavioral differences. At this stage of the work, we decided not to implement Phase 3 because the results of Phase 2 are enough to get a preliminary idea of the feasibility of the approach.

We fed the two versions of `BankAccount` to our prototype, and it automatically generated a set of test inputs for version $V1$ of the class. To generate test inputs, the prototype used both Randoop [21] (default configuration) and Agitar's JUnit Factory [12], as we stated in Section 3.1. Overall, 2,569 test inputs were generated, most of which by Randoop (all but 11). Each test input consists of pseudo-random method sequences with pseudo-random method arguments. (It is worth noting that executing the complete set of test inputs on `BankAccount` takes less than a second in this case.)

At this point, our prototype ran each input on both versions of `BankAccount`, while logging state, return value, and output information. To implement the logging, we used reflection and instrumentation of the test scaffolding. The prototype then performed the comparison of the recorded logs and suitably generated the set of raw behavioral differences for the two versions of the classes. The results of the comparison were encouraging: about 60% of the automatically generated test cases (1,557 out of 2,569) were able to reveal the behavioral difference that indicates the regression fault in the example. Figure 5 shows an example of one of such test cases. As the figure shows, the test case exercises the fault-revealing sequence that we discussed in Section 2.

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. To illustrate, consider again the test case in Figure 5. For that test case, 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 only identifies state differences that involve fields that are present in both versions of a class.

```
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 5: An example test input of `BankAccount`.

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 cases for the changed code and the use of automatically identified detailed behavioral differences.

5. RELATED WORK

Regression testing has been a fairly active research area for a number of years, and there is thus a considerable amount of related work. In this section, we review and discuss the approaches that are most closely related to ours.

The Orstra approach [28] augments a set of automatically generated test inputs with extra assertions targeted at regression faults. Orstra first runs the given test-input set and collects the return values and receiver-object states after the execution of each method under test. Based on the collected information, it then synthesizes and inserts new assertions into the existing test-input set to check future runs against the collected method-return values and receiver-object states. Parasoft Jtest [22], Agitar Agitator [3], and JUnit Factory [12] adopt a similar approach to generate test inputs with assertions called *characterization tests*. Our approach does not generate assertions, but captures instead behaviors from data collected via dynamic analysis; such data provides more detailed information—not only return values, but also receiver-object states and output. In addition, our approach generates new test inputs instead of relying only on existing test inputs. As we discussed earlier, existing test inputs may often not be sufficient to expose differential behaviors.

The Diffut approach [29] exploits the preconditions and post-conditions provided by the Java Modeling Language (JML) [15] to enable synchronized execution of two versions ($V0$ and $V1$) of a class. In particular, before the execution of a method in $V1$ with a given set of method arguments, Diffut invokes the corresponding method in $V1$ with the same arguments. After the execution of the two corresponding methods from $V0$ and $V1$, Diffut compares the return values and updated receiver-object states of these two method executions to detect behavioral differences. 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 (JVM), 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 cases 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.

Daniel and colleagues [4] generate bounded-exhaustive test inputs for testing refactoring engines. Their generated test inputs are compilable programs to be used as inputs to the refactoring engines under test. They then compare the refactored programs produced by multiple refactoring engines under test. If these refactored programs are different, they consider that at least one of the refactoring engines is incorrect. This type of testing, called *differential testing* [17], tests several implementations of the same functionality. Differential testing has been previously applied to the testing of C compilers [17], grammar-driven functionality [14], and flash file systems [11]. These previous approaches focus on differential testing of whole systems, whereas our approach focuses on testing the changed parts of two versions of a system and accounts for behavioral changes related to method return values, object states, and program outputs.

Evans and Savoia [8] propose a differential testing approach in which they generate test suites for the two given versions of a software system ($V0$ and $V1$) using JUnit Factory [12]. They then adopt the technique of assertion synthesis described earlier to create assertions in the test suite generated for each version. Assume that the generated test suites for the two versions $V0$ and $V1$ be $T0$ and $T1$, respectively. Their approach then runs test suite $T0$ on $V1$ and test suite $T1$ on $V0$. By using this approach, they were able to find 20-30% more behavioral differences than traditional regression testing approaches. 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 in a limited way. Therefore, our approach is likely to provide better regression-fault detection capability.

Some existing capture and replay techniques [7, 19, 26] capture the inputs and outputs of the unit under test during system-test execution. These techniques then replay the captured inputs for the unit as less expensive unit tests, and can also check the outputs of the unit 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 test cases 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 try to generate new tests to expose such behavioral differences. DeMillo and Offutt [6] developed a constraint-based approach to generate unit tests that can exhibit program-state deviations caused by the execution of a slightly changed program line (in a mutant produced during mutation testing [5]). Korel and Al-Yami [13] propose an approach for differential test generation for Pascal programs. They use a search-based chaining approach to generate inputs for which the two methods under test take a different execution path. They require that there is only a slight change in the methods in the two versions. Although these approaches are related to our work, they target a different problem and are more limited in scope than our approach.

Apiwattanapong and colleagues [2] 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 testers augment an existing regression test suite. Their approach does not automatically generate any test case, but simply provides guidelines for testers 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.

Ren and colleagues [24] propose a change impact analysis tool called Chianti. Chianti categorizes and decomposes the changes between two versions of a program into different atomic types, and assesses the effects of the different changes on the test cases in the program's test suite. Chianti uses only an existing test suite and does not aim to exercise behavioral differences between the two versions of the software system under test. In contrast, the goal of our approach is specifically to expose behavioral differences across versions.

Podgurski and Weyuker [23] propose a technique for re-estimating the reliability of a new version of the software based on the behavioral differences between such version and the previous one. Our approach is complementary to theirs, in that it detects behavioral differences that could be used as inputs for their reliability estimation technique.

6. CONCLUSION AND FUTURE WORK

We have presented a novel regression testing approach, called 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 cases for the changed parts of the code, (2) running the generated test cases on the old and new versions of the code and identifying differences in the outcome of the tests, 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 test cases. Second, it leverages differential behavior, which eliminates the need for developer-provided oracles. Because of these novel aspects, our approach can give developers more (and more detailed) information than traditional regression testing approaches. Our proof-of-concept study provides initial evidence of the feasibility of the approach.

We believe that these initial results, albeit preliminary, are encouraging and motivate further research along several directions. We sketch some of these possible directions for future work. A first, obvious direction is the implementation of a complete prototype of the approach that would allow us to perform an extensive empirical evaluation of the current approach. A second direction is the adoption of finer-grained differencing techniques (*e.g.,* [1]) 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. A third research direction involves investigating the use of test generation techniques that are guided by the characteristics of the identified changes, rather than being based on a mainly random generation. We will also investigate more aggressive ways to cluster, filter, and abstract changes in Phase 3, before presenting them to the user. In this context, we may also be able to leverage bug isolation techniques targeted at specific parts of the code (*e.g.,* [18]) to further reduce the develop-

ers' inspection efforts. We believe that this aspect of the technique will be of key importance when we will apply the technique to real software. Finally, as discussed in Section 3.2, BERT currently handles only a small subset of the possible outputs of a program. We plan to expand the scope of the work by including other common types of outputs, such as network-related output and graphical output in general.

Acknowledgments

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

7. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 2–13, 2004.

[2] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Proc. Testing: Academic & Industrial Conference on Practice And Research Techniques (TAIC-PART 2006)*, pages 137–146, 2006.

[3] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 169–180, 2006.

[4] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proc. joint meeting of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, pages 185–194, 2007.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[6] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[7] S. Elbaum, H. N. Chin, M. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2006)*, pages 253–264, 2006.

[8] R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. In *Proc. joint meeting of European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, pages 549–552, 2007.

[9] E. Gamma and K. Beck. JUNIT: a regression testing framework. `http://www.junit.org`, 2002.

[10] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proc. International Conference on Software Engineering (ICSE 1998)*, pages 188–197, 1998.

[11] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Proc. International Conference on Software Engineering (ICSE 2007)*, pages 621–631, 2007.

[12] JUnit Factory website, 2008. `http://www.junitfactory.com/`.

[13] B. Korel and A. M. Al-Yami. Automated regression test generation. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 1998)*, pages 143–152, 1998.

[14] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *Proc. IFIP International Conference on Testing Communicating Systems (TestCom 2006)*, pages 19–38, 2006.

[15] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, 1998.

[16] H. K. N. Leung and L. White. Insights into regression testing. In *Proc. International Conference on Software Maintenance (ICSM 1989)*, pages 60–69, 1989.

[17] W. M. McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.

[18] A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant component interactions with JINSI. In *Proc. International ICSE Workshop on Dynamic Analysis (WODA 2006)*, pages 3–9, 2006.

[19] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proc. International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, 2005.

[20] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 241–252, 2004.

[21] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA Companion 2007)*, pages 815–816, 2007.

[22] Parasoft Jtest manuals version 4.5. Online manual, 2003. `http://www.parasoft.com/`.

[23] A. Podgurski and E. J. Weyuker. Re-estimation of software reliability after maintenance. In *Proc. International Conference on Software Engineering (ICSE 1997)*, pages 79–85, 1997.

[24] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, 2004.

[25] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[26] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. IEEE International Conference on Automated Software Engineering (ASE 2005)*, pages 114–123, 2005.

[27] E. J. Weyuker. On testing non-testable programs. *Compuer Journal*, 25:465–470, 1982.

[28] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. European Conference on Object-Oriented Programming (ECOOP 2006)*, pages 380–403, 2006.

[29] T. Xie, K. Taneja, S. Kale, and D. Marinov. Towards a framework for differential unit testing of object-oriented programs. In *Proc. International Workshop on Automation of Software Test (AST 2007)*, pages 5–11, 2007.