BERT: A Tool for Behavioral Regression Testing*

Wei Jin
Georgia Tech
weijin@gatech.edu

Alessandro Orso
Georgia Tech
orso@gatech.edu

Tao Xie
NC State University
xie@csc.ncsu.edu

ABSTRACT

During maintenance, software is modified and evolved to enhance its functionality, eliminate faults, and adapt it to changed or new platforms. In this demo, we present BERT, a tool for helping developers identify regression faults that they may have introduced when modifying their code. BERT is based on the concept of behavioral regression testing: given two versions of a program, BERT identifies behavioral differences between the two versions through dynamic 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 detailed information than traditional regression testing approaches—approaches that rely exclusively on existing test suites, which may be limited in scope and may not adequately test the changes in a program. BERT is implemented as a plug-in for Eclipse, a popular Integrated Development Environment, and is freely available.[1] This demo presents BERT, its underlying technology, and examples of its usage.

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

General Terms: Algorithms, Experimentation, Reliability

Keywords: Regression testing, differential testing

1. TECHNOLOGY AND TOOL

Figure 1 provides a high-level view of our approach compared to traditional regression testing. In traditional regression testing (*e.g.,* [6]), 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* complements the aforementioned traditional approach 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

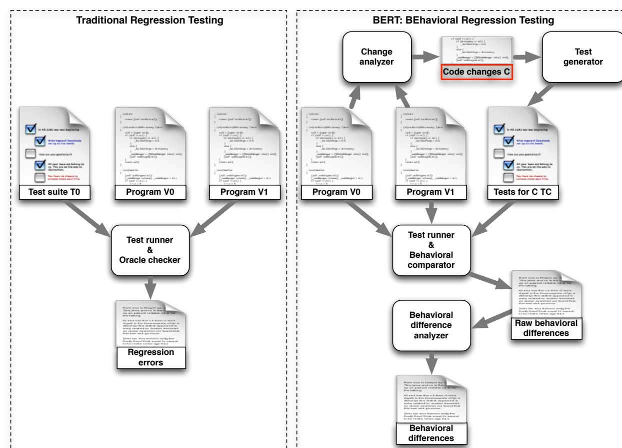

Figure 1: High-level view of our approach.

set of *behavioral differences* between these two versions. This information can provide developers with more and finer grained data on how their code 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 for our technique is one where the approach is integrated into the IDE used by the developers and is activated every time the code is updated and compiled successfully. In such a scenario, the amount of changes in the code would typically be limited and localized.

BERT is currently implemented for the Java language, integrated into Eclipse, and 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 next provide an overview of these phases. Complete details on these phases can be found in the paper that describes the approach [2].

Phase 1: Generation of 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. To do so, BERT leverages two features of Eclipse: the ability to intercept events and to produce change information between two versions of a project. More precisely, BERT 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 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. BERT then generates a set of test inputs for the changed classes in $V1$ by feeding each of these classes to a *test generator*. In the current version of the tool, we use Randoop [4] and JPF [8] as test input generators. We chose these tools because they can generate test inputs for individual classes and, with minimal changes, can build the scaffolding needed for the tests, such as drivers and stubs (*i.e.,* mock objects) and produce readily runnable JUnit tests.

*This demo illustrates the implementation of a technique presented at WODA 2008 [3] and, more extensively, at ICST 2010 [2].

[1]See `http://www.cc.gatech.edu/~orso/software.html`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-60558-791-2/10/11 ...$10.00.

Phase 2: Behavioral Comparison of 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}. After each call to a method m of c performed by t, BERT logs the following information. *First*, it 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 so, 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 logged value 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. *Second*, BERT stores the **return 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. *Third*, BERT 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. *Finally*, 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, which is later used to rank differences. To log this information, BERT instruments the tests and the code under test using Javassist (`http://www.csg.is.titech.ac.jp/~chiba/javassist`), a bytecode rewriting library written in Java.

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 found, BERT records the fact that there was a difference and a set of relevant data for differences of that type. In particular, each of the recorded changes is tagged with a unique identifier for t, which allows to map individual changes to the test 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.

Phase 3: Differential Behavior Analysis/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 that are within a class and involve the same entities. For example, for state-related differences, the analyzer groups all differences that involve the same method and field as a single behavioral difference, associates it with the set of test inputs that reveal each individual difference, and stores the individual value differences 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, (2) which methods can return different values, and (3) which differences in output can occur in c_{v0} and c_{v1} and which test inputs can cause such differences to manifest. BERT reports these differences to the developers in an Eclipse custom view, 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 are 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—an intuition confirmed by the results of our empirical evaluation [2].

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

2. RELATION WITH EXISTING TOOLS

To the best of our knowledge, our behavioral regression testing approach is novel, and BERT is the first tool that implements it. Most regression testing techniques and tools would be complementary, rather than alternative to BERT. Particularly, test input generation and maintenance tools, such as Parasoft Jtest [5], could be integrated with BERT to improve its test generation and execution phase. Similarly, capture and replay tools that can generate unit test cases from system tests (*e.g.,* [1, 7]) could be used in combination with BERT to extend the set of unit tests used to test changes.

3. BENEFITS OF BERT

BERT has two key aspects that distinguish it from traditional regression testing. First, it focuses on a small subset of the code, which lets it 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 evaluation of BERT provides initial evidence of such usefulness: for the cases considered, BERT was able to identify true regression faults while generating false positives that could be reduced through ranking [2]. This demo presents BERT and shows such results and benefits in practice.

Acknowledgments

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

4. REFERENCES

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

[2] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *Proc. of ICST*, pages 137–146, 2010.

[3] A. Orso and T. Xie. BERT: BEhavioral Regression Testing. In *Proc. WODA*, pages 36–42, 2008.

[4] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for Java. In *OOPSLA Companion*, pages 815–816, 2007.

[5] Parasoft Jtest 8.4.20, 2009. `http://www.parasoft.com/`.

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

[7] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. ASE*, pages 114–123, 2005.

[8] W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proc. ASE*, pages 3–12, 2000.