

Improving Impact Analysis and Regression Testing Using Field Data*

Alessandro Orso, Taweessup Apiwattanapong, and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
{orso|term|harrold}@cc.gatech.edu

ABSTRACT

Software products are often released with missing functionality, errors, or incompatibilities that may result in failures, inferior performances, or, more generally, user dissatisfaction. In previous work, we presented the *Gamma* approach, which enables analyses that (1) rely on actual field data instead of synthetic in-house data and (2) leverage the vast and heterogeneous resources of an entire user community.

In this paper, we investigate the use of the Gamma approach to support and improve two fundamental tasks performed by software engineers during maintenance: impact analysis and regression testing. We propose a new approach that leverages field data to perform these two tasks. We also discuss ongoing empirical studies, performed on a real subject and on a real user population, to assess the feasibility of the approach.

1. INTRODUCTION

In recent years, the number of computers and software systems has increased dramatically. Moreover, the continuous growth of the Internet has significantly increased the connectivity of computing devices and led to a situation in which most of these systems and computers are interconnected.

Although these changes create new software development challenges, such as shortened development cycles and increased frequency of software updates, they also represent new opportunities that, if suitably exploited, may provide solutions to both new and existing software quality problems.

For example, consider quality assurance tasks such as testing and analysis. If performed solely in-house, using synthetic data, such tasks may waste considerable resources in exercising configurations that do not occur in the field and code entities that are not exercised by actual users. Conversely, those tasks can miss exercising configurations and behaviors that actually occur in the field, thus lowering our confidence in the quality of the software.

We believe, and our previous research suggests, that software development can greatly benefit by augmenting analysis and measurement tasks performed in-house with analysis and measurement tasks performed on the software deployed in the field [2, 5]. We call this the *Gamma* [6] approach.

*This work was supported in part by National Science Foundation awards CCR-9988294, CCR-0096321, CCR-0205422, SBE-0123532 and EIA-0196145 to Georgia Tech, and by the State of Georgia to Georgia Tech under the Yamacraw Mission.

There are two main advantages of the Gamma approach: (1) analyses rely on actual field data, rather than on synthetic user data, and (2) analyses leverage the vast and heterogeneous resources of an entire user community.

In this paper, we investigate the use of the Gamma approach to support and improve the tasks of maintaining evolving software. We address two fundamental tasks that are routinely performed by software engineers in maintaining evolving software: impact analysis and regression testing.

To the best of our knowledge, all existing techniques for performing these tasks (e.g., [1, 4, 7]) rely on in-house data only. This paper presents new techniques that leverage field data to perform impact analysis and regression testing. We also present ongoing empirical studies, performed on a real subject and on a real user population, to assess the feasibility of our approach.

2. SCENARIO

D is the developer of a software product P , whose latest version has been released to a number of users. During maintenance, D needs to modify P to add new features and to fix bugs. Before performing the changes, D assesses the impact of such changes on the software and, indirectly, on the users. Based on the assessment, D selects the changes to be applied on P and, after performing the changes, obtains a new version of the software product, P' . At this point, D retests P' by selecting a subset of the test suite used to test P and by adding new test cases to test the added features. After testing is complete, D releases P' to the users.

This scenario involves two main tasks on the developer's side: predictive impact analysis and regression testing.

Because we want to use field data to support these tasks, we pose two requirements on D .

The first requirement is that D provides a method for instrumenting the software. There are a number of ways in which this can be done, such as including performing instrumentation before releasing the software or dynamically updating the software to insert and remove instrumented parts. Moreover, instrumentation can be added at different levels and to different extents based on the context (for example, beta testers may tolerate a considerably higher overhead than users of the final product). In the rest of the paper, we assume that D instruments P to gather coverage information at the method level.

The second requirement is that D collects execution-related data from the users. The way information is actually collected may vary based on the type of the deployed software. In the rest of the paper, we assume that D collects, over time, a set of execution data per user. The data for each

execution contain the coverage information related to that execution. Therefore, for method coverage, each execution data contains the set of methods invoked during that execution.

3. IMPACT ANALYSIS

Software impact analysis is the task of estimating the parts of the software and related documentation that can be affected if a proposed software change is made. Impact-analysis information can be used when planning changes, making changes, and tracking the effect of changes (e.g., in the case of regression errors).

Traditionally, the way to assess the impact of possible changes in the software is to use static information about the code and estimate the parts of the software that could be affected by a given change (e.g., performing forward slicing at the change points to identify a set of statements that may be affected by the changes or computing the transitive closure on the call graph starting from the method(s) where the change occurs [1]). These techniques are either imprecise, unsafe, or both, and tend to overestimate the effect of changes.

More recently, researchers started to consider more precise ways to assess the impact of changes. In particular, Law and Rothermel defined a technique for impact analysis based on dynamic analysis [4]. Although techniques based on such dynamic execution traces can improve the results of impact analysis, they may be too expensive to be used in the field. (The traces generated, even with optimized algorithms that compress the data, can easily approach hundreds of megabytes.)

To use real field data, we need to constrain both the amount of the instrumentation required to collect the data and the amount of data to be collected for each execution. The approach for impact analysis that we present requires only lightweight instrumentation and collects an amount of data on the order of a few kilobytes per execution.

3.1 Impact Analysis Using Field Data

According to the scenario described in Section 2, in considering a set of candidate changes for program P , developer D analyzes the expected impact of each of those changes on P to make an informed decision about the changes that should actually be implemented and those that should be postponed.

In the following, we use C to represent the change, or set of changes, to be performed on program P , expressed as a set of modified methods in P .

Figure 1 gives a high-level algorithm, `ImpactAnalysis`, that performs impact analysis using execution-related data. The algorithm takes three inputs: (1) a program P , (2) a set of executions of P , each of which is a set of methods, and (3) a set of changes C . `ImpactAnalysis` consists of a loop that performs two main steps for each change c in C . In the first step (lines 1–7), the algorithm identifies those executions that are affected by the changes—that is, executions that traverse at least one change. For each c in C , the algorithm identifies the set of users’ executions that execute c and stores them in AE , the set of affected executions. In the second step (lines 8–10), the algorithm identifies the remainder of the impacted methods. It does this by first computing a forward static slice SM using the appropriate variables in c as the slicing criterion, sc . Next, the algorithm computes the intersection of SM and AE , and adds the in-

algorithm `ImpactAnalysis`

input: Program P ,
Set of execution data E ; $e \in E$ is a set of methods
Set of changes C
output: Set of impacted methods IM , initially empty
declare: Set of affected executions AE ,
Set of methods SM

```

begin ImpactAnalysis
/* identify users’ executions that go through  $C$  */
1: for each change  $c \in C$  do
2:    $AE = \emptyset$ 
3:   for each execution-data set  $e \in E$  do
4:     if  $e \cap \{c\} \neq \emptyset$  then
5:        $AE = AE \cup e$ 
6:     end if
7:   end for
/* identify methods in users’ executions affected by  $C$  */
8:    $sc =$  set of variables in  $c$ 
9:    $SM =$  forward slice of  $P$  starting from  $sc$ 
10:   $IM = IM \cup (SM \cap AE)$ 
11: end for
12: return  $IM$ 
end ImpactAnalysis

```

Figure 1: Algorithm for performing impact analysis using field data.

tersection to the current set IM . After each c is processed, the resulting set, IM is the set of methods impacted by c according to the way the program is actually used in the field. If C contains more than one change, the result is the union of the sets for each change.

3.2 User-Sensitive Impact Analysis

An interesting byproduct of using field data for impact analysis is that it lets the developer assess the impact on the user of a given change or set of changes. To the best of our knowledge, this kind of impact analysis is new and has not been performed traditionally.

By knowing how users use the software, we can estimate how and to what extent various changes can affect different users.¹ This kind of analysis provides the developer with another piece of information that can be leveraged to further support the decision about which changes to integrate into the system.

There are a number of ways in which we can use field data to compute the impact on the user population. We present three such computations, and, for clarity, we express each computation as an equation.

Collective Percentage (CP). First, we can compute the percentage of users’ executions affected by C . To this end, we simply compute what we call *Collective Percentage* or CP, which is the ratio of the number of executions e in E that traverse at least one change c in C to the total number of executions E .

$$CP = \frac{\|e \in E \text{ traversing at least one } c \in C\|}{\|E\|} \quad (1)$$

For example, if C consists of two changes and the coverage data collected shows that 1,500 of the 10,000 executions ex-

¹Note that the term *user* is used here to refer to a role more than to an actual entity. In the case of a big user population, we do not expect to be able to address single users. In those cases, we may aggregate the field information at different levels (e.g., per deployment site or per company).

ercise at least one of the two basic blocks, CP is 15%. This measure can give the developer D a general idea of the overall impact of the changes on the user population, based on the way the program is actually being used.

Percentage Per User (PPU). Second, we can compute the percentage of users’ executions affected by C per user. We call this measure *Percentage Per User i* or PPU_i . To compute PPU_i , we use Equation 1 except that, instead of applying it to all executions in E , we apply it to the execution data for each user, U_i , separately. The result is, for each user, the percentage of executions of the user, E_i , that may be affected by the changes.

$$PPU_i = \frac{\|e \in E_i \text{ traversing at least one } c \in C\|}{\|E_i\|} \quad (2)$$

PPU_i provides finer grained information about the effects of changes than CP . CP , being cumulative over all the executions, can underestimate or overestimate the impact of a change on particular users. For example, a value of 50% for CP for a change C may underestimate the impact of C on some users—those whose executions traverse C very frequently. Similarly, CP can overestimate the impact of a change on users. In such a case, finer-grained information provided by PPU_i may lead to a more informed decision about of whether to implement C .

Percentage of Affected Users (PAU). Third, we can compute the percentage of all users (U) affected by C . This measure can be obtained by simply considering as affected users, those users for which the percentage of affected executions is greater than zero—that is, those U_i for which PPU_i is greater than zero.

$$PAU = \frac{\|U_i : PPU_i > 0\|}{\|U\|} \quad (3)$$

This third measure lets developer D reason about the possible effects of changes in terms of the number of users or deployed instances of the software.

Developer D is in general interested in all three kinds of information. For example, D could decide to postpone some change(s) because the impact on the users would be significant. For another example, D may decide to release the new version of P only to a subset of the users—the ones least affected by the changes.

4. REGRESSION TESTING

As software evolves during development and maintenance, regression testing is applied to modified software to provide confidence that the changed parts behave as intended and that the unchanged parts have not been adversely affected by the modifications.

Let P' be a modified version of program P , and T be the test suite used to test P . During regression testing of P' , test cases are selected from T to reduce the cost of regression testing. In attempting to reuse T for testing P' , two problems arise. First, which test cases in T should be used to test P' (the *regression test selection* problem). Second, which new test cases must be developed to test parts of P' such as new functionality (the *test suite augmentation* problem).

There are various techniques for performing regression testing that differ in the kind of static and dynamic analysis

algorithm CriticalMethods

```

input: Program  $P$ ,
        Modified version of the program  $P'$ ,
        Set of execution data  $E$ ,
        Set of changes  $C$ 
output: For each change  $c \in C$ , set of critical methods  $CM_c$ ,
        initially empty

begin CriticalMethods
  /* identify users' executions that go through each change  $c$  */
  1: for each change  $c \in C$  do
  2:    $CM_c = \emptyset$ 
  3:   for each execution-data set  $e \in E$  do
  4:     if  $e \cap \{c\} \neq \emptyset$  then
  5:        $CM_c = CM_c \cup e$ 
  6:     end if
  7:   end for
  8: end for
  9: return  $CM_c$ 
end CriticalMethods

```

Figure 2: Algorithm for identifying critical methods.

performed and in precision and efficiency (e.g., [3, 7]). To the best of our knowledge, none of the existing techniques leverages field data to perform regression testing. In the following, we present our technique for regression testing, which is based on the use of such data.

The intuition behind the technique is to consider users’ executions as test cases that cannot be rerun, but that may be representative of the use of P' in the field. Our technique identifies, for each change c , a set of *critical* methods CM_c in the program—methods that are likely to be exercised in the field by executions that traverse c .

Figure 2 gives a high-level algorithm, **CriticalMethods**, that computes the set of critical methods. Algorithm **CriticalMethods** is analogous to the first part of algorithm **ImpactAnalysis**, presented in Section 3: At line 5, CM_c contains the set of all methods traversed by users’ executions that traverse change c .

The set of critical methods can be used to assess whether, according to the field data, the regression testing of P' (i.e., the set of test cases T' rerun on P') is adequate. For each c in C , we identify the set of test cases in T' going through c , T'_c , and verify whether T'_c covered the methods in CM_c . Each method m not covered is an additional test requirement that should be satisfied before releasing P' . The requirement can be expressed as the coverage of m by a test case that also traverses c .

In the context of a traditional regression testing process, the technique would be used after performing regression test selection, to assess the adequacy of T' with respect to the test requirements derived from the field data.

It is worth noting that the presented technique is not safe. First, we do not consider the sequence of method calls, thus missing the distinction between executions that traverse a method m before and after traversing a change c . Second, we group all executions traversing a given change c , instead of considering the methods covered by each execution that traverse c . Finally, we consider coverage at the method level: a test case may cover a method m differently than the field execution(s). However, we expect such approximations to lower considerably the overhead of the technique, so making it practical and still effective. Empirical studies are needed to assess whether our hypotheses hold in practice.

5. ONGOING EMPIRICAL STUDIES

To validate the presented techniques and to assess the usefulness of using field data for impact analysis and regression testing, we are performing a set of empirical studies on a real subject and on a real user population.

For the set of empirical studies, we used JABA (Java Architecture for Bytecode Analysis), a framework for analyzing Java programs developed in Java within our research group; JABA consists of 550 classes, 2,650 methods, and approximately 60KLOC. We instrumented JABA for different kinds of coverage and released it to a set of users who agreed to have information collected during execution and sent back to our server.

We distributed the first release of the instrumented JABA to nine users, who used it for two months. This first release helped us tune the approach in terms of instrumentation, data collection, and interaction with the user's platform. Using the information we obtained from this first release, we created a second instrumented version of JABA, and distributed it to seven users.

Five of the seven users involved in the studies are working in our lab: three are part of our group and use JABA for their research; another one is a student working in our department who uses JABA for a graduate-level project; the last one is a Ph.D. student who is using a regression testing tool built on top of JABA. The remaining two users are a researcher and a student working in two different universities, one of which is abroad.

To instrument and collect the data, we used our GAMMATELLA tool [5]. The tool lets us instrument at different levels of granularity and for different kinds of information. When instrumenting, the tool also includes in the program the network-communication code that is used to send data back to our central server. On the server side, the tool performs both the data-collection and the data-storage tasks. The data for the different executions are stored in a database and can be retrieved at different levels of granularity and aggregation (e.g., per-user, per day, or per-execution).

Using GAMMATELLA, we have been gathering data for 3 weeks, during which we collected around 400 executions for the latest version of JABA. In the next months, we will continue to collect data and perform the following studies.

Study 1. The goal of this study is to assess whether the use of field data, rather than synthetic data, can yield different analysis results. To this end, we will compare the results of our technique performed using field and in-house data. As in-house data, we will use coverage data for the internal regression test suite, developed for JABA over the years (63% method coverage).

Study 2. The goal of this study is to assess the effectiveness of our impact-analysis technique compared to traditional approaches. To this end, we will compare the results of performing predictive impact analysis (i.e., the size of the computed impact sets) using three techniques: our technique, transitive closure on the call graph, and static slicing.

Study 3. The goal of this study is to assess the degree of variation among the different users in the way they interact with the program. The presence of different profiles among the users is an important indicator of the usefulness of using field data: it is hard (if at all possible) to recreate the variety in the user population with synthetic data produced in house, and such variety may affect considerably the result

of dynamic analysis. To reach our goal, we will compute the impact of a set of changes on the user population, in terms of CP, PPU, and PAU (presented in Section 3.2).

For all three studies, we will consider two sets of changes: (1) a set of 2,650 independent changes consisting of one change per method, and (2) sets of multiple correlated changes, identified using real changes for the subject program.

Although we need more data before performing the studies and having statistically relevant results, the data collected so far are promising in that they show a considerable variety in the users' behavior and evident differences in the results obtained with in-house and field data.

6. CONCLUSION

We strongly believe that quality assurance tasks can greatly benefit by augmenting analysis and measurement tasks performed in-house with field data because (1) real users are different from simulated users, and (2) real users are different from one another. Both aspects are generally not captured by in-house data and the use of such data can thus hamper the effectiveness of dynamic analyses. In this paper, we have presented two techniques that leverage field data for impact analysis and regression testing and discussed a set of ongoing empirical studies.

After performing the studies described in this paper, we will continue our work in several directions.

First, we will investigate efficient mechanisms to record users' actions and inputs, so as to be able to recreate, at least partially, users' execution in-house. This is necessary to perform studies on the effectiveness of the regression testing technique presented in this paper.

Second, we will perform studies on the stability of users' population and behavior, to assess to what degree historical field data provide useful information for the future.

Finally, we will study whether the impact sets computed by our technique reflect the actual impact of changes in the field; we will estimate the impact of future changes to JABA and use the data gathered from the field after the new releases to assess how good is our estimate.

7. REFERENCES

- [1] S. Bohner and R. Arnold. *IEEE Software Change Impact Analysis*. Computer Society Press, Los Alamitos, CA, USA, 1996.
- [2] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering*, Nov 2002.
- [3] Y. F. Chen, D. S. Rosenblum, and K. P. Vo. TestTube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, May 1994.
- [4] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, May 2003.
- [5] A. Orso, J. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proceedings of the ACM symposium on Software Visualization*, June 2003.
- [6] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proceedings of the International Symposium on Software Testing and Analysis*, Jul 2002.
- [7] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.