Leveraging Field Data for Impact Analysis and Regression Testing

Alessandro Orso, Taweesup 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 user dissatisfaction. In previous work, we presented the *Gamma* approach, which facilitates remote analysis and measurement of deployed software and permits gathering of program-execution data from the field. 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 present a new approach that leverages field data to perform these two tasks. The approach is efficient in that the kind of field data that we consider require limited space and little instrumentation. We also present a set of empirical studies that we performed, on a real subject and on a real user population, to evaluate the approach. The results of the studies show that the use of field data is effective and, for the cases considered, can considerably affect the results of dynamic analyses.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—Monitors, Testing tools, Tracing

General Terms

Algorithms, Experimentation, Reliability, Verification

Keywords

Software engineering, Gamma technology, impact analysis, regression testing

1. INTRODUCTION

In recent years, we have witnessed a fundamental shift in the way in which software is developed and deployed. Decades ago, relatively few software systems existed, and these systems were often custom-built and run on a limited number of mostly disconnected computers. After the PC revolution, 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 and performance problems. For example, consider quality assurance tasks such as testing and analysis. With rare exceptions, these activities are currently performed in-house, on developer platforms, using developer-provided inputs. The underlying assumption is that the software is tested and analyzed in the same way that it will be used in the field. Unfortunately, this assumption is rarely satisfied. Consequently, considerable resources can be wasted on exercising configurations that do not occur in the field and code entities that are not exercised by actual users. Conversely, in-house testing can miss exercising configurations and behaviors that actually occur in the field, thus lowering our confidence in the testing.

Our previous research suggests that software development can significantly benefit by augmenting analysis and measurement tasks performed in-house with analysis and measurement tasks performed on the software deployed in the field. We call this the *Gamma* [11] approach. 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 varied resources of a user community, not just the limited, and often homogeneous, resources found in a typical development environment.

There are many scenarios in which the Gamma approach can be exploited, and numerous tasks that can benefit from using the Gamma approach. In previous work, we investigated the use of the Gamma approach for collecting coverage information from deployed instances of the software for use in creating user profiles, determining classes of users of the software, and assessing the costs and identifying the issues associated with collecting field data [1, 4]. We also investigated the use of Gamma for visualization of program-execution data collected from the field for use in investigating aspects of the behavior of software after deployment [10].

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., [2, 3, 9, 15, 17]) rely on in-house data only.

We present a technique for performing these tasks that uses field data collected from actual users of the deployed software. To use the techniques in realistic settings, we must constrain both the instrumentation required to collect the field data and the amount of field data collected. Our techniques require only lightweight instrumentation and collect data on the order of a few kilobytes per execution.

We also present a set of empirical studies that evaluate our approach. For the studies, we used a real program and a real setting: we instrumented a version of a program-analysis tool developed within our research group, released it to a set of users, and collected dynamic data while the users used the tool for their work. Our studies show that, for this system, field data can be effective in improving the quality of the dynamic analyses we considered. The studies also show that, for the same system, traditional techniques would compute results that do not reflect the actual use of the system.

The main contributions of this paper are:

- a technique to perform impact analysis that leverages data collected from the field;
- a technique to support regression testing that leverages data collected from the field; and
- a set of empirical studies that show how those dynamic analyses can benefit from the use of real field data.

2. SCENARIO

In this section, we sketch the scenario that we investigate. We refer to the scenario throughout the paper to illustrate our techniques.

D is the developer of a software product P, whose latest version has been released to a number of users. During maintenance, D is considering modifying P. Before performing the changes, D assesses the impact of such changes on P and, indirectly, on the users. Based on the assessment, D selects the changes to be applied to 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 affected and 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 impose 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, including performing instrumentation before releasing the software, providing an instrumenter on-site that performs the instrumentation on demand, or dynamically updating the software while it is running [12]. For this work, we assume that developer D adds lightweight instrumentation to program P before releasing it.

D can add instrumentation at different levels and to different extents based on the context in which P is used. 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 information about the coverage of either basic blocks[1] (hereafter referred to as blocks) or methods.

The second requirement is that, when users execute P, coverage information is collected and sent back to D. The way information is collected may vary. For example, for a platform that is always connected, the data may be sent back as soon as they are produced, whereas for a mostly disconnected platform, the data could be stored locally and sent back when a connection is available. 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. For example, if the execution information collected is method coverage, the data for each execution consist of the set of methods covered by that execution.

3. IMPACT ANALYSIS

Software impact analysis is the task of estimating the parts of the software 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 effects of changes (e.g., in the case of regression errors).

One traditional way to estimate the impact of one or more changes is to use static information about the code (i.e., dependences) to identify the parts of the software that may be affected by such changes. For example, inserting changes in the code and performing forward slicing[2] from the change points yield a set of statements that may be affected by the changes. For another example, transitive closure on the call graph[3] starting from the method(s) where the changes occur results in an (unsafe) estimate of the set of methods affected by the changes [3]. These techniques are generally imprecise and tend to overestimate the effects of changes.

Recently, researchers have considered more precise ways to assess the impact of changes, using information gathered during execution. For example, Law and Rothermel defined a technique for impact analysis [9] that uses *whole path profiling* [8] to estimate the effects of changes.

Although techniques based on such execution traces can achieve better results than traditional impact-analysis approaches [9], they are constrained by the quality of the data used: if these techniques are applied in-house, the use of synthetic inputs may limit their effectiveness; if the set of inputs is inadequate, so will be the results of the analysis. However, using real data gathered from the field may not be an option for these techniques: (1) the size of the execution traces generated during execution can easily approach thousands of megabytes, and (2) algorithms that compress the traces to a reasonable size can be computationally expensive and cannot be straightforwardly used on-line, while the execution traces are produced.

To use field data, a technique must constrain both the instrumentation required to collect the data and the data collected for each execution. Our approach for impact analysis requires only lightweight instrumentation and collects data on the order of a few kilobytes per execution.

[1]A *basic block* is a sequence of statements in which control enters at the first statement and exits at the last statement without halt or possibility of branching except at the end.

[2]*Forward slicing* determines, for a point p in program P and a set of variables V, those statements in P that may be affected by the values of variables in V at p [19].

[3]A *call graph* is a directed graph in which nodes represent functions (or methods) and an edge between two nodes A and B means that A may call B.

```
algorithm ChangeImpact
input: Program $P$,
         Set of execution data $E$; $e \in E$ is a set of entities,
         Changed entity $c$
output: Set $IE$ of entities impacted by $c$
declare: Set of affected executions $AEx$, initially empty,
         Set of entities $SL$
begin ChangeImpact
 /* identify users' executions that go through $c$ */
 1: for each execution-data set $e \in E$ do
 2:    if $e \cap \{c\} \neq \emptyset$ then
 3:       $AEx = AEx \cup e$
 4:    end if
 5: end for
 /* identify entities in users' executions affected by $c$ */
 6: $SL$ = forward slice of $P$ starting at $c$ with variables in $c$
 7: $IE = SL \cap AEx$
 8: return $IE$
end ChangeImpact
```

Figure 1: Algorithm for performing impact analysis for an entity-level change.

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 change on P to make an informed decision about which changes should be implemented and which changes should be postponed.

In illustrating the technique, we use the term *entities* to indicate either blocks or methods, depending on the level of instrumentation and detail chosen by developer D. We use C to represent a change to be performed on program P. C consists of one or more *entity-level changes* c, which are changes to single entities in P. In the general case, developer D will have a set of candidate changes C_1, C_2, $\ldots$, C_n to perform on the system, each one consisting of one or more entity-level changes. We use E to represent a set of execution data e, where each e is expressed as the set of entities traversed by an execution.

Given a change C, our technique for impact analysis identifies an *impact set*—a set of entities potentially impacted by change C. To this end, our technique computes, for each entity-level change c in C, an approximate dynamic slice based on the execution data for executions that traverse c. The impact set is the union of the slices computed for each c in C.

The algorithms to compute the impact set, ChangeImpact and ImpactAnalysis, are shown in Figures 1 and 2, respectively. ChangeImpact performs change-impact analysis for an entity-level change c, whereas ImpactAnalysis uses the results of ChangeImpact to compute the impact set for a set of entity-level changes C.

Algorithm ChangeImpact takes three inputs: (1) a program P, (2) a set of execution data for P, E, and (3) a changed entity c. The algorithm consists of two main steps.

In the first step (lines 1–5), the algorithm identifies those executions that are affected by c. To this end, the algorithm identifies the set of users' executions that traverse c and stores them in AEx, the set of affected executions.

In the second step (lines 6–8), ChangeImpact identifies the entities impacted by c. First, the algorithm computes a forward static slice SL using c and the variables in c as the slicing criterion. Then, the algorithm computes IE as the intersection of SL and AEx, and returns it. IE is the set of entities impacted by c.

Algorithm ImpactAnalysis uses ChangeImpact and computes the impact set for change C, which may consist of multiple entity-level changes. This algorithm also takes three inputs: (1) a program P, (2) a set of execution data for P, E, and (3) a change C.

```
algorithm ImpactAnalysis
input: Program $P$,
         Set of execution data $E$; $e \in E$ is a set of entities,
         Change $C$; $c \in C$ is an entity-level change
output: $IEC$, the impact set for $C$, initially empty
use: $ChangeImpact(P, E, c)$ returns the affected entities set $IE$
       for program $P$, executions $E$, and change $c$.
begin ImpactAnalysis
 1: for each change $c \in C$ do
 2:    $IEC = IEC \cup ChangeImpact(P, E, c)$
 3: end for
 4: return $IEC$
end ImpactAnalysis
```

Figure 2: Algorithm for performing impact analysis for a change C.

The algorithm consists of a loop (lines 1–3) that simply invokes ChangeImpact for each change c in C and computes the union of the resulting IE sets. After all c's are processed, the resulting set, IEC, is returned (line 4). IEC provides developer D with an estimate of the parts of P that would be impacted by C, according to the way P is actually used in the field.

3.2 User-Sensitive Impact Analysis

A useful byproduct of using field data for impact analysis is that it lets developer D assess the impact on the users of a given change or set of changes. By knowing how users use the software, D can estimate how and to what extent various changes can affect different users.[4]

To the best of our knowledge, this is a new kind of impact analysis that provides the developer with another piece of information 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.

Collective Percentage (CP) is the percentage of users' executions affected by C. CP is the ratio of the number of executions in E that traverse at least one change c in C to the total number of executions E.

$$\mathrm{CP} = \frac{\|e \in E \text{ traversing at least one } c \in C\|}{\|E\|} \tag{1}$$

For example, if C consists of two entity-level changes, and the execution data collected show that 1,500 of the 10,000 executions exercise at least one of the two changes, CP is 15%. This measure can give 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$_i$) is the percentage of users' executions affected by C per user. To compute PPU_i, we use Equation 1 applied to the execution data for user U_i, rather than to all executions in E. For each U_i, the result is the percentage of executions of U_i, E_i, that may be affected by the changes.

$$\mathrm{PPU}_i = \frac{\|e \in E_i \text{ traversing at least one } c \in C\|}{\|E_i\|} \tag{2}$$

[4]The term *user* refers to a role instead of an actual entity. For a large user population, we may need to aggregate the field information at different levels (e.g., per deployment site or per company).

PPU_i provides finer-grained information about the effects of changes than CP. Because CP is cumulative over all executions, it 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 for which most executions traverse C. In such cases, finer-grained information provided by PPU_i may lead to a more informed decision on whether or not to implement C.

Percentage of Affected Users (PAU) is the percentage of all users U affected by C. This measure is obtained by considering as affected those users for which the percentage of affected executions is greater than zero.

$$\text{PAU} = \frac{\|U_i : PPU_i > 0\|}{\|U\|} \tag{3}$$

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

Developer D can use all three kinds of information to make decisions during maintenance. 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.

For P', a modified version of program P, let C be the set of entity-level changes between P and P', T be the test suite used to test P and T' the test suite to regression test P'. When developing T', three problems arise: (1) which test cases in T should be used to test P' (*regression test selection*); (2) which new test cases must be developed (*test suite augmentation*); and (3) which order should be used to run the test cases (*test-suite prioritization*).

There are a number of ways to obtain T' that differ in the kind of analysis performed and in precision and efficiency (e.g., [5, 6, 14, 15, 16, 18, 20]). Our technique leverages field data to support regression testing. We use the results of impact analysis to help selecting, augmenting, and prioritizing T'.

First, based on coverage information for P, our technique selects an initial set T' that consists of all test cases in T that traverse at least one change in C.

Then, we use the set of affected entities identified by impact analysis to assess whether, according to the field data, T' is adequate for P'. This step is based on the intuition that, in the field, executions that traverse changed parts of the code are more likely to traverse affected entities than other entities in the program. Therefore, we consider T' adequate for P' if, for each entity-level change c and each affected entity ae in the IE set for c, there exists at least one test case in T' that traverses ae after traversing c. We call the affected entities for which no such test case exists *critical entities.* If T' is not adequate for P', T' will need to be augmented with additional test cases to cover the critical entities.

```
algorithm RegressionTesting
input: Program $P$,
       Set of test cases $T$ for $P$,
       Set of execution data $E$; $e \in E$ is a set of entities,
       Change $C$; $c \in C$ is an entity-level change
output: Set $T'$ of test cases in $T$ that traverse at least
          one critical entity, initially empty,
        Sets of critical entities $CE[]$, one for each entity-
          level change
use: $cov(t)$ returns the entities in $P$ covered by test case $t$,
     $ChangeImpact(P, E, c)$ returns the impact set $IE$
       for program $P$, executions $E$, and change $c$.
declare: Impact set $IE$,
         Set of entities exercised by a test case $EXIE$,

begin RegressionTesting
 1: for each change $c \in C$ do
 2:   $IE = ChangeImpact(P, E, c)$
 3:   $CE[c] = IE$
 4:   for each test cases $t \in T$ do
 /* If the test case does not traverse $c$, skip it */
 5:     if $cov(t) \cap \{c\} == \emptyset$ then
 6:       continue
 7:     end if
 8:     $T' = T' \cup \{t\}$
 /* Identify affected entities covered by $t$ */
 9:     $EXIE = cov(t) \cap IE$
10:     if $EXIE \neq \emptyset$ then
11:       $CE[c] = CE[c] - EXIE$
12:     end if
13:   end for
14: end for
15: return $T'$, $CE[]$
```

Figure 3: Algorithm for identifying the initial T' and the critical entities.

Additionally, we can use the information on the affected entities to prioritize the test cases in T', by giving a higher priority to test cases that cover a higher number of affected entities. By doing so, we prioritize according to the way in which we expect the program to be used in the field. This way of prioritizing can be combined with other existing prioritization techniques (e.g., [16, 18]). For example, the number of affected entities covered can be an additional parameter of the prioritization.

Figure 3 shows algorithm `RegressionTesting`, which selects the initial set T' and computes the set of critical entities. The algorithm takes four inputs: (1) a program P, (2) a set of test cases for P, T, (3) a set of execution data for P, E, and (4) a change C.

For each entity-level change c in C (loop 1–14), the algorithm first identifies the entities impacted by c, IE, using algorithm `ChangeImpact` (see Section 3.1) and initializes the set of critical entities for change c, $CE[c]$, to IE.

Then, for each test case t in T (loop 4–13), the algorithm checks whether t traverses change c and, if not, continues to the next test case (lines 5–7). If t does traverse c, the algorithm inserts t in T' (line 8) and inserts the affected entities exercised by t in set $EXIE$ (line 9). If $EXIE$ is not empty, the algorithm removes the entities in $EXIE$ from the set of critical entities for c (lines 10 and 11).

When all entity-level changes have been processed, T' contains all test cases that traverse at least one change, and $CE[]$ contains one set of critical entities for each change c in C. At this point, `RegressionTesting` returns T' and $CE[]$ (line 15).

As stated above, each critical entity is an additional test requirement that should be satisfied before releasing P'—the requirement can be expressed as the coverage of that entity by a test case that also traverses c. Developer D can thus use the sets of critical entities to augment T'. D can also use the information about the number of affected entities covered by the test cases in T' to prioritize the test cases.

Our technique is unsafe for two reasons. First, it does not consider the sequence in which entities are executed, thus missing the distinction between executions that traverse an entity before and after traversing a change c. Second, it uses information about coverage on version P to approximate information about P', although executions with the same inputs may cover different entities in P and P'. However, we expect such approximations to lower the overhead of the technique, so making it practical and still effective.

5. EMPIRICAL EVALUATION

To validate the techniques that we presented and to assess the usefulness of using field data for impact analysis and regression testing, we performed a set of empirical studies.

5.1 Experimental Infrastructure

For the studies, we used a real system: Java Architecture for Bytecode Analysis (JABA),[5] which is a framework for analyzing Java programs developed in Java within our research group; JABA consists of 550 classes, 2,800 methods, and approximately 60KLOC. JABA provides components that read bytecode from Java class files and perform analyses such as control flow and data flow.

We instrumented JABA for different kinds of coverage and released it to a set of users who agreed to have information collected during execution.

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 [10].

Using the information that we obtained from this first release, we created a second instrumented version of JABA, and distributed it to 11 users. The studies reported in this paper are based on the data collected using the second release of our tool. Five of the 11 users had already used JABA for their work (and were part of the first data collection experiment), whereas the other six users had just started projects that involved the use of JABA.

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

When the users run JABA, dynamic information is collected and sent back to a server in our lab that continuously collects and stores this information. The information includes profiling and coverage data at block and method levels. The data for the different executions are stored in a database and can be retrieved at various levels of granularity and aggregation (e.g., per-user, per day, or per-execution).

To instrument and collect the data, we used our GAMMATELLA tool [10]. 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.

[5] `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`

Using GAMMATELLA, we gathered data for 18 weeks, during which we collected approximately 2,000 executions for both versions of JABA. The data set that we use for the studies consists of the 1100 executions collected during the ten weeks after the release of the second version of JABA.

Although we collected coverage information at both block and method levels, for the empirical studies we used only method-level data. We made this decision for two main reasons: (1) instrumentation to collect coverage information at the method level has lower overhead than instrumentation for block coverage and is more likely to be used in practice, and (2) results at the method level are less precise than results at the block level (i.e., good results at the method level imply as good, if not better, results at the block level).

5.2 Study 1: Impact Analysis Using Field Data

The goal of the first study is twofold: (1) to assess whether using field data, instead of synthetic data, can yield different analysis results, and (2) to assess the effectiveness of our technique over traditional approaches to impact analysis.

To achieve our goal, we performed an experiment in which we compared the results of performing predictive impact analysis using four techniques:

Dynamic impact analysis using field data. Developer D applies our technique for predictive impact analysis (Section 3.1) and leverages field data; the field data used are described in Section 5.1. We refer to this technique as the FIELD technique.

Dynamic impact analysis using in-house data. D uses our technique for predictive impact analysis, except that D uses in-house data (i.e., coverage data for the internal regression test suite). As a regression test suite, we used the test suite that has been developed for JABA over the years. We refer to this technique as the IN-HOUSE technique.

Transitive closure on the call graph. Given C, expressed in terms of changed methods, D estimates the impact set by computing a transitive closure on the call graph starting from nodes that correspond to changed methods. We refer to this technique as the CALL-GRAPH technique.

Static slicing. D estimates the impact set by performing forward slicing from the change points. In the study, we approximate a forward slice by computing simple reachability on the interprocedural control-flow graph. We refer to this technique as the SLICING technique.

To compare these techniques, we use the size of the impact sets they compute; a similar approach was used by Law and Rothermel [9]. Comparing the impact sets computed by techniques FIELD and IN-HOUSE lets us assess the effect of using field data on the results of the analysis in a controlled way: the only parameter that varies between the two techniques is the set of execution data considered. Comparing the impact sets computed by technique FIELD and techniques CALL-GRAPH and SLICING lets us assess the results of our technique compared to traditional impact-analysis approaches.

In the experiment, the impact analysis techniques considered are the independent variables, whereas the resulting impact sets computed by each technique are the dependent variables. A parameter of the experiment is the set of changes C considered. We performed two instances of the experiment, using different sets of changes.

Table 1: Summary data for the comparison of techniques FIELD (*FL*), IN-HOUSE (*IH*), CALL-GRAPH (*CG*), and SLICING (*SL*) performed considering one change per method at a time.

	FL	*IH*	*CG*	*SL*	*FL/IH*	*FL-IH*	*IH-FL*	*FL/CG*	*FL/SL*
AVG	257.39	330.75	71.42	1974.91	0.88	34.95	108.31	70.08	0.11
STD	341.39	378.93	272.85	1036.38	0.3	78.56	204	156.78	0.17
MAX	802	806	1763	2531	2.09	676	761	802	1

5.2.1 *Experiment with Single Changes*

In the first experiment, we considered a set of 2,800 changes, each consisting of one change per method.[6] For each change, we computed the impact sets for the program using the four techniques and compared the results. Because of the size of the results, we cannot represent them in tabular form. Thus, we provide a summary of the results in Table 1.

In the table, we report a number of measures. *FL*, *IH*, *CG*, and *SL* are the sizes of the impact sets computed by the FIELD, IN-HOUSE, CALL-GRAPH, and SLICING techniques, respectively. *FL/IH* is the ratio of the size of the impact set computed by technique FIELD to the size of the impact set computed by technique IN-HOUSE; *FL/CG* and *FL/SL* are defined analogously. *FL-IH* is the set difference between the impact set computed by technique FIELD and the impact set computed by technique IN-HOUSE (i.e., the number of methods that are considered impacted by technique FIELD, but are not considered impacted by technique IN-HOUSE). *IH-FL* is defined analogously.

For each measure, we report the following information: *AVG*, the average value of the measure computed for each of the 2,800 changes considered; *STD*, its standard deviation; and *MAX*, the maximum value assumed. For example, row *AVG* for column *FL-IH* shows the average number of methods that are in the impact set produced by technique FIELD, but not in the impact set produced by technique IN-HOUSE, computed over the 2,800 changes considered.

This experiment is significant because it considers a complete distribution of changes within a program. However, the experiment is representative only of situations in which all changes to the program consist of only one entity-level change and are independent from one another.

In most cases, changes consist of multiple correlated entity-level changes—a change in method m_1 may require a change in method m_2 to be implemented, and so on. In those cases, the developer must estimate the joint impact of multiple entity-level changes at once. To address those cases, we performed a second experiment.

5.2.2 *Experiment with Real Changes*

In this experiment, we address the case of multiple correlated entity-level changes that must be performed together. To ensure the meaningfulness of the considered changes, instead of randomly aggregating method-level changes, we used a set of real changes for the subject program. To this end, we extracted the last 21 versions of JABA from our CVS repository. For each (version, subsequent-version) pair (v_i,v_{i+1}) of JABA, we identified the changes between the two versions and, for each change, (1) mapped it to the method m containing the change, and (2) added m to the set of changes C_i. The resulting sets of 20 changes, C_1 to C_{20}, are the sets that we used for our experiment.

[6]Because the experiment was conducted at the method level, the location of the change within the method is irrelevant.

Table 2 shows the number of methods changed for each of the 20 sets. As the table shows, the number of methods changed ranges from a minimum of 2, for change sets C_6 and C_{16}, to a maximum of 178, for change set C_8.

The results of computing the impact sets using the four considered techniques are shown in Table 3. The measures reported in each column of Table 3 are the same as those reported in Table 1, except that the first column, *C*, shows the set of changes considered. Moreover, because for this experiment we consider only 20 (instead of 2,800) sets of changes, in the table we show both the results per set of changes and the summary results.

5.2.3 *Discussion*

We discuss the results of this first study by addressing its two goals separately.

The first goal was to assess whether the use of field data, rather than synthetic data, can yield different analysis results. To this end, we consider the differences between the results of techniques FIELD and IN-HOUSE. The data presented in Tables 1 and 3 clearly show that the results of the analysis are affected significantly by the kind of data considered.

Consider, for example, the results in Table 3: for 18 of the 20 changes (all but C_5 and C_{16}), a significant number of methods (68–139) are in the impact set computed by FIELD but not in the impact set computed by IN-HOUSE and vice-versa (97–122). Similar results can be observed in Table 1. Also in this case, techniques FIELD and IN-HOUSE compute on average fairly dissimilar impact sets.

Considering Table 3, note that, for all change sets considered, the sizes of the impact sets computed by techniques FIELD and IN-HOUSE are almost identical—what differs is the composition of those sets. For the results in Table 1, the situation is not as extreme: the average value for *FL/IH* is 0.88, and its standard deviation is 0.3.

Note also that the above results are not due to the fact that the sets of entities covered by the in-house test suite and by the field executions are mostly disjoint. In fact, the internal test suite and the field executions both cover approximately 65% of the code and their coverage sets have an 85% overlap.

The second goal of the study was to assess the effectiveness of our technique compared to traditional approaches to impact analysis. The results in the tables show that our technique FIELD computes impact sets that are considerably and consistently smaller (11% and 30% on average, for the two experiments) than technique SLICING. Because, unlike technique SLICING, technique FIELD is unsafe, we cannot generally claim that FIELD produces better results than SLICING. However, in all cases in which the user population we considered is a good representative of the actual user population, which should be case when using stable field data, technique FIELD provides, by definition, accurate information on the actual impact of the changes.

Table 2: Number of methods changed in the sets of real versions considered.

C sets	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}	C_{13}	C_{14}	C_{15}	C_{16}	C_{17}	C_{18}	C_{19}	C_{20}
# methods	15	3	15	6	3	2	3	178	95	12	87	28	6	61	22	2	61	5	6	89

Table 3: Results for the comparison of techniques FIELD (FL), IN-HOUSE (IH), CALL-GRAPH (CG), and SLICING (SL)performed considering real changes.

C	*FL*	*IH*	*CG*	*SL*	*FL/IH*	*FL-IH*	*IH-FL*	*FL/CG*	*FL/SL*
C_1	776	784	1608	2519	0.99	96	104	0.4876	0.31
C_2	778	771	1688	2519	1.01	116	109	0.4568	0.31
C_3	778	784	1546	2519	0.99	110	116	0.5071	0.31
C_4	780	778	56	2522	1	112	110	13.8929	0.31
C_5	617	617	10	2521	1	0	0	61.7000	0.24
C_6	750	765	1546	2519	0.98	86	101	0.4948	0.30
C_7	791	796	7	2519	0.99	97	102	113.7143	0.31
C_8	806	794	1896	2617	1.02	126	114	0.4188	0.31
C_9	822	785	1751	2551	1.05	139	102	0.4483	0.32
C_{10}	789	800	1547	2519	0.99	111	122	0.5171	0.31
C_{11}	737	766	1833	2528	0.96	68	97	0.4179	0.29
C_{12}	802	797	1593	2522	1.01	113	108	0.5003	0.32
C_{13}	805	788	1575	2519	1.02	120	103	0.5003	0.32
C_{14}	797	784	1724	2524	1.02	122	109	0.4548	0.32
C_{15}	773	751	35	2525	1.03	127	105	21.4571	0.30
C_{16}	0	0	3	2519	1	0	0	0.0000	0.00
C_{17}	790	767	1625	2523	1.03	130	107	0.4720	0.31
C_{18}	753	759	1546	2519	0.99	98	104	0.4909	0.30
C_{19}	763	761	322	2520	1	99	97	2.3634	0.30
C_{20}	819	793	1561	2522	1.03	131	105	0.5080	0.32
AVG	736.3	732	1173.6	2527.3	1.01	100.05	95.75	10.99	0.29
STD	174.11	172.14	1896	21.73	0.02	37.22	32.46	27.37	0.07
MAX	822	800	729.68	2617	1.05	139	122	113.71	0.32

As far as the comparison with technique CALL-GRAPH is concerned, the results in Table 3 show that, in five of 20 cases—for change sets C_4, C_5, C_7, C_{15}, C_{19}—technique FIELD selected impact sets that are larger (in some cases significantly larger) than the corresponding sets computed by CALL-GRAPH. These results indicate that the impact sets computed with the CALL-GRAPH technique can be highly inaccurate and have little or nothing to do with the behavior of the program. In the remaining 15 cases, technique FIELD selected impact sets of about half the size of the corresponding sets computed by CALL-GRAPH. The results in Table 1 are similar in nature and show an even larger difference between the results for the two techniques.

Also in this case, we cannot draw a general conclusion that technique FIELD computes more accurate results than technique CALL-GRAPH. Nevertheless, we can conclude that technique CALL-GRAPH is not likely to provide a good estimate when we want to assess the effect of changes based on the program usage.

Our findings that static-analysis based techniques, such as SLICING and CALL-GRAPH, can produce overestimates and, in the case of CALL-GRAPH, unsafe estimates, are consistent with the results obtained by Law and Rothermel [9].

5.3 Study 2: User Sensitive Impact Analysis

The goal of the second study is to assess the degree of variation among the different users in the way they use the program. The presence of different profiles among the users is an important indicator of the usefulness of using field data: it is difficult (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 conducted the following experiment: we computed the impact of each change set considered for the first experiment of Study 1 (i.e., 2,800 independent changes, one per method), on the user population, in terms of CP, PPU, and PAU (Section 3.2). For space reasons, we show the results only for CP and PAU.

Figures 4 and 5 contain scatter plots that show the distribution of CP and PAU, respectively, when different methods are modified. The horizontal axes represent the identifier of the changed method, ranging from 1 to 2,800. The vertical axes represent the values of CP and PAU, respectively.

As the figures show, both CP and PAU vary dramatically depending on the location of the change, and both measures appear evenly distributed in the data space. This result indicates a considerable variety in users' behaviors, which can make it difficult to create synthetic data to exercise the program in the same way in which users exercise it.

To gather evidence of this difficulty, we performed an additional study: we measured the differences in the value of CP computed using our in-house data (i.e., coverage data for the internal regression test suite) and the field data. For each of the 2,800 changes considered, we compared the value of CP. The difference between CP computed using in-house and field data was 15% on average, with a standard deviation of 10 and a maximum of 52%. These results also show that in-house data may provide a poor approximation of field data. For example, if we use the number of executions affected by a change as an estimate of the impact of such a change, we compute results that can differ significantly for field and in-house data.

Note that, for the study, we considered changes at the method level, and thus we computed an overestimate of the actual impact. If the real change is in a part of the method that is not in the main path within the method (i.e., the

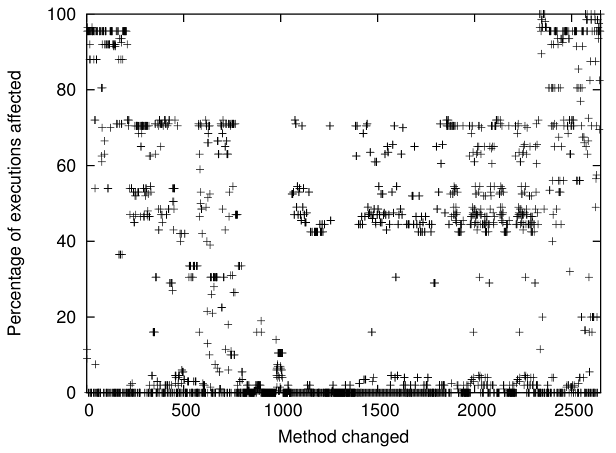

Figure 4: Distribution of CP based on the location of the changes.

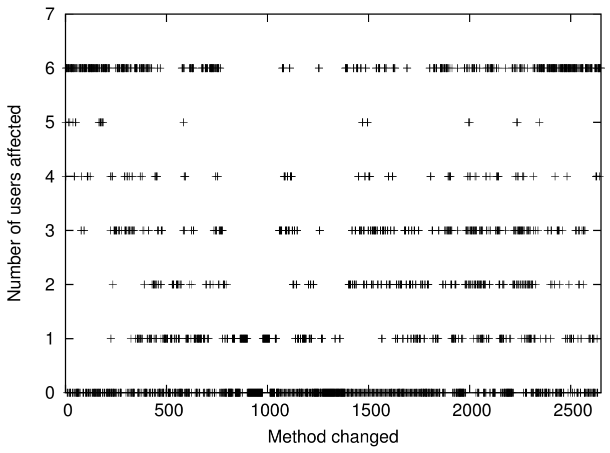

Figure 5: Distribution of PAU based on the location of the changes.

Table 4: Number of critical methods for 20 real changes (the total number of methods is 2,800).

Change	$C1$	$C2$	$C3$	$C4$	$C5$	$C6$	$C7$	$C8$	$C9$	$C10$
Avg #CM	361.43	48.33	223.13	272.5	331	145	72	19	301	84.58

Change	$C11$	$C12$	$C13$	$C14$	$C15$	$C16$	$C17$	$C18$	$C19$	$C20$
Avg #CM	0.95	204.93	232.83	127.19	304.59	0	215.26	74.4	99.17	96.14

change is not dominated by the method's entry), then the number of affected executions, and possibly users, would in general be lower than what we have found, and the distribution of the impact information even more varied.

Although we performed this study only to assess the variety in the users' behavior, the results provided us with some interesting insights. For example, we found that CP, PPU, and PAU are complemental measures: in JABA, our experimental subject, there are several methods for which a change in the method affects less than 15% of the executions (low CP), but affects 100% of the users (high PAU). For another example, we realized that there are many cases in which almost no users are affected at all by the changes.

5.4 Study 3: Regression Testing Using Field Data

Our third study is divided in two parts with distinct goals.

The goal of the first part of the study is to assess whether the use of field data instead of in-house data actually yields different requirements for regression testing. To this end, we computed the set of critical entities (methods, in this case) for the 20 real changes considered in Study 1 and for our internal regression test suite.

Table 4 shows the results of this study. In the table we report, for each change C, the average number of critical methods for that change (*Avg #CM*). We compute *Avg #CM* by averaging the number of critical methods in the $CE[]$ sets C, obtained using the algorithm presented in Section 4.

The table provides additional evidence of the importance of using field data. For most changes, the number of critical methods is fairly high and for some changes it is extremely high. Consider, for example, changes $C1$, $C5$, $C9$, and $C15$, for which the number of critical methods is greater than 300. For those changes, there are more than 300 methods that may be executed in the field by executions that traverse at least one change and that our regression test suite does not adequately exercise. The data show that there are only a few cases in which the in-house regression testing adequately exercises the program with respect to its use in the field.

In most cases, the complexity of the program under test makes it impractical to develop a generally adequate set of tests. Because the number of critical entities is a measure of the adequacy of the (regression) testing performed in-house, the use of field data can help testing by providing precise directives on where to improve existing test suites. In fact, based on the collected field data, we are extending our internal test suite to (re)test JABA similarly to the way it is actually used.

The goal of the second part of the study is to assess the imprecision introduced by the use of coverage information for P to estimate information on P' (see Section 4). To achieve our goal, we studied how the coverage at the method level changed among a set of versions of JABA. We first selected 11 versions of JABA and ran all versions against a set of 200 test cases, while collecting coverage information at the method level. Then, for each (version, consecutive-version) pair (v_i, v_{i+1}) of JABA and for each test case, we compared the coverage for the two versions; to compare the coverage, we assumed that methods with the same fully-qualified name and signature corresponded to each other. Finally, we averaged the number of methods covered in v_i but not in v_{i+1} over all test cases.

The differences in coverage between versions provide an indicator of the imprecision that we introduce by estimating P''s coverage using P's coverage.

Table 5 shows the results of this study. For each version v_i of the program, the table reports the number of corresponding methods changed from v_i to v_{i+1} (*#MC*), the number of test cases affected by the changes (*#TCA*), the average number of corresponding methods covered in v_i but not in v_{i+1} (*Avg #CB*), and the average number of corresponding methods covered in v_{i+1} but not in v_i (*Avg #CA*).

Table 5: Results for the estimate of coverage across 10 different versions of a program.

Ver	*#MC*	*#TCA*	*Avg #CB (%)*	*Avg #CA (%)*
v_1	13	198	5 (0.31%)	0
v_2	64	78	0	0
v_3	11	198	0	0
v_4	4	198	0	0
v_5	4	198	1 (0.06%)	0
v_6	12	198	1 (0.06%)	9 (0.54%)
v_7	23	198	0	1 (0.06%)
v_8	6	198	0	0
v_9	1	21	0	0
v_{10}	2	156	5 (0.3%)	1 (0.06%)

As the results show, the differences in coverage are negligible, even for changes that involve a number of methods and are traversed by most test cases. In the worst case, from v_6 to v_7, the imprecision introduced by using coverage information collected on the old version of the program is 10 methods, which is 0.6% of the total number of methods covered. Therefore, at least for the case we considered, estimating coverage information does not introduce any considerable imprecision.

5.5 Threats to Validity

Like any empirical validation, ours has limitations. Some threats to the validity of the studies are described along with the studies. In the following, we discuss the limitations that apply to the overall experimental design and setting.

Some limitations involve external validity. First, we have considered the application of our techniques to a single program and test suite. Second, we have considered only a limited number of users and, thus, collected only a small set of field data. Therefore, we cannot claim generality for our results. However, our subject program is a real program, the test suite that we used for the experiments is the actual regression test suite for the program, the users involved in the experiment are real users, and the changes considered are real. Nevertheless, additional studies with other subjects are needed to address such questions of external validity.

Other limitations involve internal and construct validity. We have approximated static slicing with reachability, which may produce imprecise results. This imprecision generally results in computing larger impact sets than those that slicing would compute, therefore affecting the results of the comparison of the four techniques in Study 1. We have also assumed that there is a given degree of stability in the users' behavior over time. When we have collected enough historical data, we will need to study whether and to what extent this assumption holds.

In short, our results support an existence argument: cases exist in which it is feasible to use field data and their use can produce benefits in impact analysis. Therefore, these results motivate us to perform further research, followed by carefully controlled experimentation, to investigate whether such results will generalize.

6. RELATED WORK

To the best of our knowledge, our work represents the first attempt at using field data to directly support impact analysis and regression testing. However, other researchers have investigated the idea of performing quality-assurance activities, such as analysis and testing, after deployment.

The Perpetual Testing project recognizes the need to develop "seamless, perpetual analysis and testing of software through development, deployment and evolution," and proposes Residual Testing [13]. Although related, Residual Testing uses field data with a different goal: continuously monitoring for fulfillment of test obligations that were not satisfied in the development environment.

Another approach is Expectation-Driven Event Monitoring (EDEM). EDEM uses software agents deployed over the Internet to collect application-usage data to help developers improve the fit between application design and use [7]. This approach addresses the problem of monitoring deployed software and collecting field data, but it focuses on the human-computer-interaction aspects of the problem.

Other related work does not consider the use of field data, but it performs similar kinds of analyses.

Srivastava and Thiagarajan [18] present a system, Echelon, for prioritizing the set of test cases to be rerun on an application during regression testing. Their technique is based on identifying changes and mappings between old and new versions of a program and on coverage estimation.

Law and Rothermel [9] define a technique for impact analysis based on executing a program with a set of inputs, collecting compressed traces for those inputs, and using the traces to predict impact sets. The technique can improve the accuracy and the precision of existing techniques based on static analysis, but there is no evidence that it can be efficiently used in the field.

7. CONCLUSION

In this paper, we have presented the results of our investigation of how field data can be leveraged to support and improve maintenance tasks. In particular, we focused on impact analysis and regression testing. We defined new techniques to perform those two tasks based on field data and we performed an empirical evaluation of our approach.

Although preliminary, the results of our empirical evaluation are significant because they were obtained on a real subject distributed to real users. These results are promising in that they show the twofold importance of field data: (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; the use of such data can therefore hamper the effectiveness of dynamic analyses.

Importantly, the studies performed and their results fueled further research, by suggesting a number of directions for future work.

First, we will perform studies on the stability of users' behavior. One of the underlying assumptions of our approach is that the users' behavior is stable, and thus historical field data provide useful information for the future. The data that we have so far are too limited to perform a study and assess whether our assumption holds. Therefore, we will expand our user population by releasing JABA to a larger number of users. We will use the field data collected on a larger number of users and for a larger period of time to study the dynamics of the user population and its behavior.

Second, we will investigate the use of statistical analysis to perform clustering on the field data and to study whether it is possible to identify discriminating characteristics among users. We are especially interested in the use of these techniques to perform anomaly detection of deployed software. We will also investigate the use of clustering to identify representative executions or users. The ability to do so would be extremely useful for testing and analysis tasks (e.g., it may allow the selection of a small set of test cases that approximate well the users' behavior).

Third, we will continue to investigate the use of field data for impact analysis and regression testing. We will verify whether the impact sets computed by our technique reflect the actual impact of changes in the field. To this end, we will apply our technique to 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. We will also experiment with prioritizing test cases based on information on critical entities and study how this approach affect the fault-detection capabilities of regression testing.

Fourth, we will investigate which other execution-related information may be leveraged for dynamic analysis tasks. In particular, we will consider information that is not control-flow related, such as memory occupation or flow of data in the program, which may provide important information on how a program is used in the field.

Fifth, based on our findings when we start collecting data from an increasing number of users, we will investigate scalability issues. For example, we might discover that collecting complete information from all users is not a viable solution when a high number of users are involved, and define sampling techniques. For another example, we may discover that monitoring at the site level, rather than at the user level, lets us perform more efficient data collection.

Finally, we will investigate efficient mechanisms to record users' actions and inputs to enable recreation, at least partially, of users' execution in-house (e.g., to augment a regression test suite). This challenging problem involves both static analysis, to identify parts of the code where inputs are received and that are good candidates for instrumentation, and dynamic analysis, to collect input data.

Acknowledgments

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. Anil Chawla helped with the development of GAMMATELLA and its use for our experiments. Saurabh Sinha and the anonymous reviewers provided valuable feedback on a previous version of the paper.

8. REFERENCES

[1] T. Apiwattanapong and M. J. Harrold. Selective path profiling. In *Proc. of the ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42, Nov 2002.

[2] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *Proc. of the International Conf. on Software Maintenance*, pages 292–301, Sep 1993.

[3] S. Bohner and R. Arnold. *IEEE Software Change Impact Analysis.* Computer Society Press, Los Alamitos, CA, USA, 1996.

[4] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *Proc. of the ACM Workshop on Program Ana lysis for Software Tools and Engineering*, pages 2–8, Nov 2002.

[5] Y. F. Chen, D. S. Rosenblum, and K. P. Vo. TestTube: A system for selective regression testing. In *Proc. of the 16th International Conf. on Software Engineering*, pages 211–222, May 1994.

[6] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 312–326, Oct 2001.

[7] D. M. Hilbert and D. F. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, 32(4):384–421, Dec 2000.

[8] J. Larus. Whole program paths. In *Proc. the ACM Conf. on Programming Language Design and Implementation*, pages 1–11, May 1999.

[9] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proc. of the 25th International Conf. on Software Engineering*, pages 308–318, May 2003.

[10] A. Orso, J. Jones, and M. J. Harrold. Visualization of program-execution data for deployed software. In *Proc. of the ACM symposium on Software Visualization*, pages 67–76, Jun 2003.

[11] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 65–69, Jul 2002.

[12] A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Proc. of the IEEE International Conf. on Software Maintenance*, pages 649–658, Oct 2002.

[13] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proc. of the 21st International Conf. on Software Engineering*, pages 277–284, May 1999.

[14] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *ACM International Symposium on Software Testing and Analysis*, pages 169–184, Aug 1994.

[15] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, Apr 1997.

[16] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, Oct 2001.

[17] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proc. of the ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 46–53, Jun 2001.

[18] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 97–106, Jul 2002.

[19] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[20] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on text differencing. In *International Conf. on Reliability, Quality, and Safety of Software Intensive Systems*, May 1997.