Effective Identification of Failure-Inducing Changes: A Hybrid Approach[*]

Sai Zhang, Yu Lin, Zhongxian Gu, Jianjun Zhao
School of Software
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
{saizhang, linyu1986, ausgoo, zhao-jj}@sjtu.edu.cn

ABSTRACT

When regression tests fail unexpectedly after a long session of editing, it may be tedious for programmers to find out the failure-inducing changes by manually inspecting all code edits. To eliminate the expensive effort spent on debugging, we present a hybrid approach, which combines both static and dynamic analysis techniques, to automatically identify the faulty changes. Our approach first uses static change impact analysis to isolate a subset of responsible changes for a failed test, then utilizes the dynamic test execution information to rank these changes according to our proposed heuristic (indicating the likelihood that they may have contributed to the failure), and finally employs an improved *Three-Phase* delta debugging algorithm, working from the coarse method level to the fine statement level, to find a minimal set of faulty statements.

We implemented the proposed approach for both Java and AspectJ programs in our AutoFlow prototype. In our evaluation with two third-party applications, we demonstrate that this hybrid approach can be very effective: at least for the subjective programs we investigated, it takes significantly (almost 4X) fewer tests than the original delta debugging algorithm to locate the faulty code.

1. INTRODUCTION

Programmers often spend a significant amount of time debugging programs in order to reduce the number of bugs in software releases. In modern software development, coding and testing are interleaved activities to assure code quality. Typically, when regression tests fail or produce any unexpected result after a long session of editing, it may indicate potential defects in the updated software. When attempting to fix an exhibited bug, programmers usually: (1) identify statements involved in failed tests, (2) narrow the search by selecting suspicious changes that might contain faults, (3) hypothesize about the suspicious faults, and (4) restore the program variables to a specific state [8]. However, the search of suspicious changes is an arduous, highly involved, and manual process. This phase can be quite time-consuming and expensive.

The high cost of locating fault causes in software evolution has motivated the development of automatic debugging techniques, such as [7, 12, 14, 16, 17, 19, 21, 33]. Of particular interest for our work is the delta debugging algorithm provided by Zeller [27]. In the work [27] by Zeller et al. on delta debugging, the reason for a program failure is identified as a set of differences (the deltas) between program versions that distinguish a passed program execution from a failed one. A set of failure-inducing differences is determined by repeatedly applying different subsets of the changes (the configurations) to the original program and observing the outcome of executing the intermediate programs. By correlating the outcome of each execution with the set of changes applied, one can narrow the set of failure-inducing changes. It was shown [27] that delta debugging is effective in finding faulty code even in large software applications like GDB [3]. However, the original delta debugging algorithm is a general and language-independent technique applicable to minimizing the suspicious changes for better debugging. When employing it to a specific programming language like Java, this approach can be improved. For instance, the original delta debugging algorithm searches the entire set of changes to identify the failure-inducing ones. However, for a specific failing test, a portion of uncorrelated changes can be ignored and we only need to focus on the related changes. Also, delta debugging selects and applies the changes in an arbitrary order. But ideally, the changes which are most likely to contribute to the failure should be ranked highest and tried first. Furthermore, one of the most important practical problems in delta debugging is *inconsistent configurations*. Since the original delta debugging builds the intermediate program versions by using the structural differences between a succeeding and failing program version (e.g., changing one line or one character to generate an intermediate program version), it is likely that several resulting configurations are inconsistent - combinations of changes that do not result in a testable program. In addition, delta debugging treats all program changes as one flat atomic list. However, if we apply delta debugging to each level of configurations, a large portion of irrelevant changes might be pruned out earlier. Therefore, there is clearly scope to improve upon delta debugging.

In this paper, we present a hybrid approach that involves both static and dynamic analysis techniques to automate the debugging process. Static analysis lends itself to obtaining generalized properties from the program text, while dynamic analysis offers the semantics and ease of concrete program execution. The need and benefit to combine the two approaches has been repeatedly stated in the software engineering community [10, 26]. More specifically, in locating the failure-inducing changes, a static analysis can soundly analyze an entire program and prune out all unrelated changes for a specific failure, while a dynamic analysis can avoid analysis ap-

proximations and help pinpoint the faulty code. In our approach, we use atomic change [20, 32] representation to capture the semantic differences between a succeeding and a failing program version. Then we use static change impact analysis [32] to isolate a subset of responsible changes for the failed test (reduce the search space). We next collect the dynamic tests execution information to rank these responsible changes according to our proposed heuristic, indicating the likelihood that they may have contributed to the failure (optimize the change selection). Finally, we employ an improved *Three-Phase* delta debugging algorithm to multiple levels of program changes, working from the coarse method level to the fine statement level, to determine a minimal set of faulty changes (handle multiple configuration granularities). Furthermore, the interdependence relationships between atomic changes guarantee most of the generated intermediate program versions are syntactically valid. Therefore, the number of tests that need to be re-tested due to program inconsistency can be further reduced.

The goal of our work is to provide a general approach for improving the effectiveness of locating faulty changes. It should not be only applicable to one specific language. We currently implement our approach for Java and AspectJ [15] - two popular languages in object-oriented and aspect-oriented paradigms, - in our AutoFlow debugging framework. In the experiment with two non-trivial third-party applications, we demonstrate the effectiveness of this hybrid approach compared to the original delta debugging: for both Java and AspectJ programs, it takes significantly (almost 4X) fewer tests to locate the failure-inducing changes.

In summary, the main contributions of this paper are: **(1)** a novel hybrid analysis approach for identifying failure-inducing changes in software evolution, **(2)** a tool that implements the proposed approach for Java and AspectJ programs, and **(3)** an experimental study showing that our approach can be effective in locating the faulty code with significantly less overhead.

We first begin with the background of delta debugging and change impact analysis (Section 2), and then present our hybrid debugging approach (Section 3). Two case studies using our AutoFlow prototype highlight the practical issues (Section 4). We close with a discussion of related and future work, where we recommend our hybrid approach as an integrated part of the delta debugging technique after any failing regression test (Section 5 and Section 6).

2. BACKGROUND AND EXAMPLE

We first introduce the background of the delta debugging algorithm and change impact analysis technique. Then we present a motivating example to give an intuition of our approach.

2.1 Delta Debugging

Delta debugging, proposed by Zeller [28], is a general approach for automating faulty change isolation and test case minimization. As summarized in [17], it consists of two algorithms:

- **Simplification:** This algorithm simplifies the failure-inducing input by examining smaller configurations of the input. The algorithm recurses on the smaller failure configurations until it can not produce a smaller configuration which still produces a failure.
- **Isolation:** This algorithm attempts to find a passing configuration such that with the addition of some elements it becomes a failing configuration.

Simplification produces inputs which still cause a program failure. It may be less intuitive as a fault localization support for programmers. Given the simplified test inputs, programmers may still need to manually inspect the potentially large change set to identify which part should be responsible for the failure. In this paper, we focus on the second algorithm, isolation. The isolation delta

debugging algorithm [27], hereafter called *dd*, is to isolate failure-inducing configurations. The input to *dd* is a set of configurations that cause a program fail when applying to the base program, and the output is a minimal set of failure-inducing configurations.

2.2 Change Impact Analysis

Change impact analysis [6, 31] aims to determine the effects of a source editing session. The analysis technique used in this paper relies on the computation of a set of atomic changes. The concept of atomic change is proposed in [6] for Java, and is extended to aspect-oriented paradigm [31] in our previous work. The atomic changes shown in Table 1 and 2 represent the source code modifications at a coarse method level. Our earlier research prototype, Celadon [31], performs static change impact analysis for AspectJ programs[1]. When given two program versions, Celadon divides program edits into a set of atomic changes, identifies all affected program fragments and a subset of regression tests affected by these changes. In essence, for each affected test, Celadon automatically identifies all its responsible changes which may affect its behavior. Initial experiments with Celadon on both Java and AspectJ programs have evidenced promising results for these analyses [32].

Abbreviation	Atomic Change Name
AF	Add a field
DF	Delete a Field
FI	Change a Field Initialization
AM	Add an Empty Method
DM	Delete an Empty Method
CM	Change Body of Method
AC	Add an Empty Class
DC	Delete an Empty Class
LC	Change Virtual Method Lookup

Table 1: A catalog of atomic changes for Java programs

Additionally, there are semantic dependencies between atomic changes. That is, an atomic change C_1 is dependent on another atomic change C_2, if applying C_1 to the original version of the program without also applying C_2 will cause a syntactically invalid program. The syntactic dependence relationship (i.e., in the above example, C_2 is a *prerequisite* for C_1) between atomic changes is crucial in constructing syntactic valid intermediate program versions during delta debugging and will be further explored in Section 3. Though the abstraction used by Celadon is the method-level change, in this hybrid debugging approach we will extract all statement changes to determine the faulty code at the statement level.

2.3 Example

We next present a contrived AspectJ program to give an intuition of our debugging approach. Details will be given in Section 3. Since the AspectJ language is a superset of Java, the Java programs can be treated in the same way in our approach.

Figure 1 shows a small AspectJ program. In Figure 2, there are three JUnit tests, testDebit, testCredit and testTransfer for the program. The original program consists of all program fragments except for those marked by underline. In this example, we assume the source edits marked by underline in Figure 1 are all newly added.

The JUnit tests in Figure 2 pass in the original program, but method testTransfer() fails in the updated program version. We start to debug by first decomposing the source editing into a set of atomic changes. The atomic changes corresponding to the source edits in Figure 1 are shown in Figure 3. Each atomic change is shown as a box, where the top half of the box shows the category of the change (Table 1 and 2), and the bottom half shows the

---

[1]Since AspectJ is an extension to Java, Celadon can also handle Java programs.

Abbreviation	Atomic Change Name
AA	Add an Empty Aspect
DA	Delete an Empty Aspect
INF	Introduce a New Field
DIF	Delete an Introduced Field
CIFI	Change an Introduced Field Initializer
INM	Introduce a New Method
DIM	Delete an Introduced Method
CIMB	Change an Introduced Method Body
AEA	Add an Empty Advice
DEA	Delete an Empty Advice
CAB	Change an Advice Body
ANP	Add a New Pointcut
CPB	Change a Pointcut Body
DPC	Delete a Pointcut
AHD	Add a Hierarchy Declaration
DHD	Delete a Hierarchy Declaration
AAP	Add an Aspect Precedence
DAP	Delete an Aspect Precedence
ASED	Add a Soften Exception Declaration
DSED	Delete a Soften Exception Declaration
AIC	Advice Invocation Change

Table 2: A catalog of atomic changes for AspectJ programs

method, field or advice involved. An arrow from an atomic change A_1 to A_2 indicates that A_2 is dependent on A_1.

After getting all atomic changes, we next isolate the responsible changes for the failed test. For `testTransfer()`, all the responsible changes are number 1, 2, 4, 11, 12, 13, 14, 15, and 16. From these nine changes, AutoFlow generates two *Atomic-Change-Chain*s, in which the first chain includes changes 1, 2, and 4; and the second chain includes others. The *Atomic-Change-Chain*s are fed as inputs of the core *Three-Phase* delta debugging algorithm. After the first phase, AutoFlow ignores the second chain and outputs the first chain as the suspicious one. In the second phase, AutoFlow ranks the atomic changes in the first chain as 2, 1 (ignore change 4); and identifies the failure-inducing change to be atomic change 2. In the third phase, AutoFlow takes the statement changes in atomic change 2 as the input, and finally determines the change `money = money - FEE` is the only faulty statement.

Our hybrid approach sketched above can be fully automated and quite effective. In this example, it locates the faulty changes within only 7 iterations, reducing the number of tests dramatically from 25 iterations by the original delta debugging.

3. APPROACH

We next present our debugging approach in detail. Our approach consists of three main components: a change isolation module based on static change impact analysis (Section 3.1), a change ranking module based on test execution information (Section 3.2), and a *Three-Phase* delta debugging module (Section 3.3).

3.1 Isolating Responsible Changes

Static change impact analysis is used to isolate all the responsible changes for a specific failed test. As presented in Section 2, our change impact analysis (presented in full detail in [31, 32]) relies on the computation of a set of atomic changes, which capture all source modifications at a method level.

Another core part of our impact analysis approach is the (aspect-aware) call graph representation [32]. The call graph is constructed for the failed test to determine all its responsible atomic changes. The set of responsible atomic changes of a given test includes: (1) atomic changes contained in the Java code, changes like changed methods (**CM**) and added methods (**AM**) that correspond to a node in the call graph, and changes like lookup change (**LC**) that corresponds to an edge in the call graph; and (2) the atomic changes

```
public class Account {
    protected double money;
    protected Auth auth;
    private final double FEE = 2;
    public Account(double money, Auth auth)
    {  this.money = money; this.auth = auth; }
    public double balance(){return money;}
    public void debit(double m)
    {  if(!auth.login()) return;
       money = money - m; }
    public void credit(double m)
    {  money = money + m; }
    public void transfer(Account other, double m) {
        if(!auth.login()) return;
        other.credit(m);
        money = money - m;
        if(m<100) {
            money = money - FEE;
        }
    }
}
public class Auth {
    private String usr, pwd;
    public Auth(String usr, String pwd)
    {  this.usr = usr; this.pwd = pwd; }
    public boolean login()
    {  if(usr.equals("usr") && pwd.equals("pwd"))
            return true;
       else return false; }
}
public aspect AccountAspect {
  pointcut AccountDebit(Account account, double m) :
   execution(pubic void Account+.debit(double))
   && target(account) && args(m);
  before(Account account,double m):AccountDebit(account,m){
      if(account.balance() < m) {
         throw new IllegalArgumentException("..."); }}
}
public aspect AuthAspect {
  pointcut LoginEntry():execution(boolean Auth.login());
  boolean around():LoginEntry() {return true; }
}
```

Figure 1: An example program

appearing in the aspect code, including changes of adding a new advice (**AEA**), changing an advice body (**CAB**), introducing a new inter-type declared method (**INM**) and changing an inter-typed declared method body (**CIMB**) that correspond to a node in the call graph. The responsible atomic changes also include the advice invocation change (**AIC**) that corresponds to an edge in the call graph. The whole responsible atomic change set further includes the transitively prerequisite atomic changes of all above changes.

Our implementation AutoFlow builds static call graphs for failed tests to soundly identify all responsible changes, but it can also work with dynamic call graphs generated from the real program execution trace. After the change impact analysis, we can ignore a portion of irrelevant changes, and focus on the viable and interesting changes in the following steps.

▶**Example.** The call graph for the failed method `testTransfer()` is shown in Figure 4. We use shadows to annotate the modified method or advice. As we can see, the responsible changes for this test include atomic changes 11, 12, 13, and 14, which appear on the graph; and their dependent changes 1, 2, 4, 15, and 16. Therefore, all responsible changes are: 1, 2, 4, 11, 12, 13, 14, 15, and 16.◀

3.2 Ranking Suspicious Changes

The heuristic for ranking responsible atomic changes is based on the hypothesis that, if an atomic change is covered by a high percentage of failed regression tests, it is more likely to be the faulty one. For each responsible change of a failed test, a score is assigned based on the test's execution information. Then we sort these changes in *descending* order based on the score, so that the change with the highest score gets the top ranking [14]. The higher

```
public class AccountTest extends TestCase {
    public void testDebit()
    { Account a = new Account(100, new Auth("usr","pwd"));
        a.debit(10);
        assertTrue(a.balance() == 90); }
    public void testCredit()
    { Account a = new Account(100, new Auth("usr","pwd"));
        a.credit(10);
        assertTrue(a.balance() == 110); }
    public void testTransfer()
    { Account a1 = new Account(100, new Auth("usr","pwd"));
        Account a2 = new Account(80, new Auth("usr","pwd"));
        a1.transfer(a2, 10);
        assertTrue(a1.balance() == 90);
        assertTrue(a2.balance() == 90); }
}
```

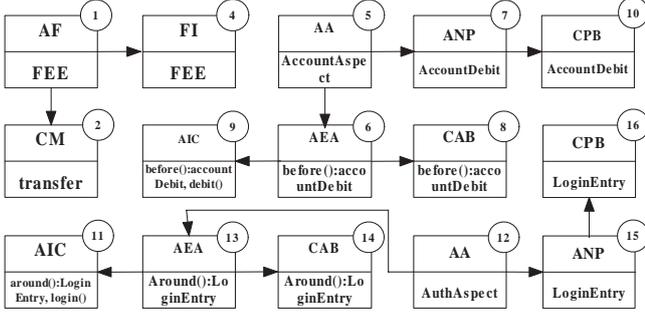**Figure 2: JUnit tests for the motivating example**

Figure 3: Atomic changes inferred from the source changes.

Figure 4: Call graph for the failed test: testTransfer().

it ranks, the more likely it may have contributed to the test failure. We define the score of an atomic change, c, as the following equation:

$$score(c) = \frac{\%failed(c)}{\%passed(c) + \%failed(c)}$$

In this equation, $\%failed(c)$ is a function that returns, as percentage, the ratio of the number of failed tests that cover c as a responsible change to the total number of failed tests in the test suite. Likewise, $\%passed(c)$ is a function that returns, as percentage, the ratio of the number of passed test cases that cover c as a responsible change to the total number of passed test cases in the test suite. For example, if an atomic change is covered by 100% of the failed tests and 50% of the passed tests, its score will be 2/3.

This heuristic utilizes the result provided by many test executions instead of only the failed ones. It can help programmers understand more complex relationships in the system, rather than the limited information provided by few tests. Unlike the heuristics in [14] and [19], which will be further discussed in Section 5, our heuristic utilizes the call graph coverage information instead of statement profiling to rank suspicious changes at the method level.

▶**Example.** Consider the atomic change 2 and 8 in Figure 3. Change 2 (**FI**) is covered by testTransfer which turns out to fail, and change 8 (**CAB**) is covered by testDebit which turns out to pass. Therefore, the scores of changes 2 and 8 are computed as follows:

$$score(2) = \frac{1/1}{1/1 + 0/2} = 1, score(8) = \frac{0/1}{0/1 + 1/2} = 0 ◀$$

3.3 Locating Failure-Inducing Changes

We next present the core module, an improved *Three-Phase* delta debugging algorithm. Unlike the original delta debugging algorithm, we focus on the responsi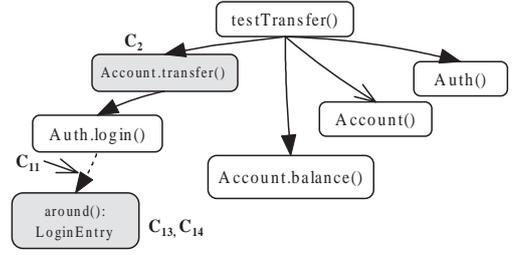ble changes of a failed test, and increase the change granularity from coarse method level to fine statement level in the following phases.

3.3.1 Compute Atomic-Change-Chain

As mentioned in Section 2, there are semantic dependencies between atomic changes. If an atomic change C_1 depends on C_2, we denote C_2 as a parent change of C_1 and C_1 as a child change of C_2. If we apply C_1 without C_2 to the original program, it will lead to a syntactically incorrect program.

An *Atomic-Change-Chain* starts from an atomic change which does not have a child change and includes all of its transitive parent changes. Like a cause-effect chain, an *Atomic-Change-Chain* contains a self-contained set (all changes connected to the starting change) for one specific change. In this phase, we generate all *Atomic-Change-Chain*s, and pass them to the delta debugging algorithm as inputs. The output result is a minimal set of suspicious failure-inducing chains.

▶**Example.** The algorithm for computing *Atomic-Change-Chain* is quite straightforward. Consider the changes in Figure 3, we compute two *Atomic-Change-Chain*s from changes 2 and 11[2]. As a result, the first chain includes changes 1, 2, and 4; and the second chain includes changes 11, 12, 13, 14, 15, and 16.◀

3.3.2 Determine Faulty Atomic Changes

After determining the minimum set of faulty *Atomic-Change-Chain*s, we increase the change granularity to the atomic change level. In this phase, we treat each atomic change in the faulty *Atomic-Change-Chain*s as one configuration. We first reduce the size of the atomic change set by pruning out all the *Def-Change*s[3]. The *Def-Change*s represent changes of declaring (or deleting) an empty language element, such as **AC** (Add Empty Class). Applying these *Def-Change*s alone without their corresponding *Body-Change*s may be meaningless or even results in syntax errors in the intermediate program version. Then, a score will be assigned to each atomic change according to the heuristic defined in Section 3.2. In this phase, we employ the delta debugging to identify all the faulty atomic changes. However, when dividing atomic changes into subsets in delta debugging, the changes with higher rank are always applied first.

▶**Example.** Consider the motivating example. After the first phase of debugging, the second *Atomic-Change-Chain* including changes 1, 2, and 4 will be output as the faulty one. Among the atomic changes in the output chain, change 1 (**AF**) is regarded as a *Def-Change* and will be pruned out. The remaining two changes 2 and 4 will be first ranked and then fed as inputs of delta debugging. In this example, changes 2 and 4 are only covered by the failed test

---

[2]Change 2 and 11 do not have a child change. Note that atomic change 4, 8, 10, 14, and 16 has already been merged into its parents as a buddy change in the change impact analysis phase. For more details on this point, please refer to [30].

[3]All *Def-Change*s include **AA**, **DA**, **INF**, **DIF**, **INM**, **DIM**, **AEA**, **DEA**, **ANP**, **DPC**, **AF**, **DF**, **DM**, **AC**, and **DC** from Table 1 and 2

testTransfer, so they have the same score: $score(2) = score(4)$ = 1. Thus, AutoFlow orders the changes arbitrarily and takes them as the inputs of delta debugging. At the end of this phase, only atomic change 2 will be output as the faulty one.◄

3.3.3 Find Faulty Statements

After determining the minimum set of faulty atomic changes, our approach extracts all changed statements from these changes, and treats each statement as one configuration. Then, again, we use the original delta debugging to identify a minimum set of faulty statements as the analysis result.

►**Example.** Consider the changes in Figure 3, atomic change 2 is output by the end of the second phase as the only faulty change. AutoFlow then extracts all changed statements (marked by <u>underline</u> in the `transfer` method body in Figure 1) and applies them to the delta debugging algorithm. Finally, statement `money = money - FEE` is identified to be the only faulty statement.◄

Note that, in this phase, it is likely that several resulting configurations (formed by combining the statement changes in an arbitrary way) are inconsistent. Therefore, we integrate the extended dd^+ [27] algorithm into our approach to handle such cases. Due to space limitations, the formal algorithm description and complexity analysis can be found in our technical report [30].

4. EMPIRICAL EVALUATION

We implemented our approach for both Java and AspectJ programs in our AutoFlow debugging framework. AutoFlow is built on top of AJDT [2] and designed as an eclipse [4] plugin. The current implementation supports Java 2 SDK v1.5 and AspectJ version 1.5. The implementation details can be found in [29].

4.1 Experiment Setup

Programs	Type	#Loc	#Ver	#Me	#Tests
XML-Security	Java	16800	4	1221	112
Dcm	AspectJ	3423	2	249	157

Table 3: Subject Programs

We evaluated our approach on two non-trivial third-party applications, `XML-Security` and `Dcm`. `XML-Security` [5] is a moderate-sized Java application that implements security standards for XML. We obtained several sequential versions with the regression test suite from the Software Infrastructure Repository (SIR) [11]. The `Dcm` application is one of the largest AspectJ benchmarks from the ajc distribution package [1]. It is also widely used by researchers [24, 25] to evaluate their work. Table 3 lists, for these two subject programs, the program type (Type), the number of lines of code in the initial version (#Loc), the number of versions (#Ver), the number of methods (#Me), and the size of the JUnit test suite (#Tests).

For each subject program, we execute the JUnit tests of version v_{n-1} in context of version v_n. If a JUnit test fails, we take the program version v_{n-1} and v_n as well as the test suite as the input of the AutoFlow to identify the failure-inducing changes.

4.2 Result and Analysis

Case Study 1: XML-Security. It is a challenge to find appropriate test data for AutoFlow to simulate the debugging activities. However, we find one test method `testSetOctetStreamGetNodeSet1()` in class `XMLSignatureInputTest` passes in its 2nd version, but fails in its 3rd version. We use AutoFlow to locate its failure causes. AutoFlow first computes the atomic changes from the source code modifications. As shown in Figure 5, there are totally 312 atomic changes between version v_2 and v_3 (shown in the v_3 bar). Then, AutoFlow isolates a set of 60 responsible changes for the failed test, which only account for 19.2% of the total number. AutoFlow next assembles 10 *Atomic-Change-Chain*s from these responsible changes and starts the *Three-Phase* delta debugging pro-

cedure. After seven iterations, one **CM**[4] change is identified as the failure-inducing change. After then, AutoFlow extracts three statement changes from this method change and starts to explore the faulty statements. Finally, after another three iterations, one statement change[5] has been output as the root of failure. In order to confirm our result, we comment out this statement change and re-execute the failed test, which then succeeds. This indicates that the statement change pinpointed by AutoFlow is a valid one. In this example, if we apply the original delta debugging directly, it requires 40 tests in our implementation.
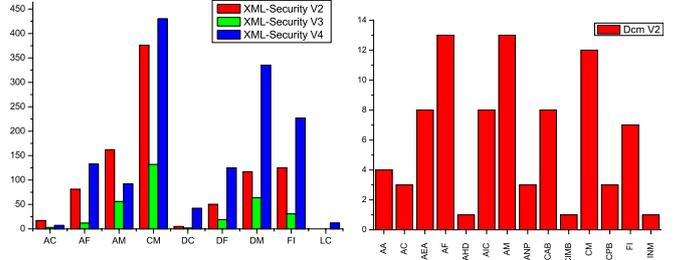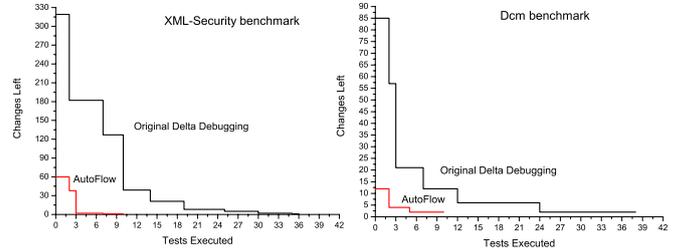

Figure 5: Atomic changes between version pairs.

Figure 6: Debugging log of locating failure-inducing changes.

Case Study 2: Dcm. Though `Dcm` is among the largest AspectJ benchmarks available, it has only 3KLOC in the initial version. The size may affect the scalability of our analysis approach, but this is not the focus of this case study. The major goal is to apply our approach to the aspect-oriented features. For this subject program, we also find one test `DCM.DataTest.testTotalDCM()` passes in the 1st version but fails in the 2nd version. Like the `XML-Security` experiment, we first compute the atomic changes between each successive versions. As shown in Figure 5, there are 85 atomic changes between `Dcm` version v_1 and v_2. After change impact analysis, AutoFlow isolates 12 responsible changes for the failed test. Those 12 changes comprise various kinds of aspectual feature changes, such as **AIC**, **AEA** and **ANP**. Following the procedures of *Three-Phase* delta debugging, AutoFlow first constructs three *Atomic-Change-Chain*s and identifies two chains containing eight atomic changes to be interfered. These eight suspicious atomic changes are further explored by AutoFlow. Then, two atomic changes, namely **CM** (`DCM.Data.totalDCM()`) and **CAB** (`DCM.handleGC.AllocFree.after():dataoutput()`), are identified by AutoFlow to be the failure-inducing ones. There are 23 lines of statement changes in these two atomic changes. AutoFlow extracts them and finally determines two statement changes `long totalDCM = 1` and `DCM.Data.decrAllocated(classname)` to be the faulty code. In this example, AutoFlow runs 11 tests to identify a combination of two faulty statements, which is significantly <u>less than the original delta debugging</u> (38 tests).

---

[4]The body change of method `org.apache.xml.security.utils.XMLUtils.circumventBug2650(Document)`

[5]`documentElement.setAttributeNS(Constants.NamespaceSpecNS, "xmlns", "");`

4.3 Discussion

The debugging log is shown in Figure 6. We can see that AutoFlow isolates one (two) faulty statement(s) from 312 (85) method-level changes after only 10 (11) tests, in the XML-Security (Dcm) case study. Compared to the original delta debugging, it significantly improves the effectiveness. In the experiment, we find that change impact analysis plays a crucial role in the debugging process, because normally, only a portion of changes (about 20% in our experiment) would affect the behavior of a specific failed test. The semantic relationships between atomic changes guarantee the syntactical correctness of intermediate program versions, avoiding a large number of inconsistent configurations. The proposed *Three-Phase* delta debugging organizes the changes hierarchically and fastly narrows down the search of suspicious changes. Since the original delta debugging algorithm handles configuration interference cases, even for a combination of multiple faulty changes (like in the Dcm case), AutoFlow can still find them. Though the current implementation of AutoFlow only supports Java and AspectJ programs, we believe the basic idea of our approach can be applied to other programming languages, as long as we employ suitable change impact analysis technique, ranking heuristic, and change selection strategy. Investigating the applicability of this approach to other languages is a topic for further research.

Threats to Validity. Like any experimental study, this evaluation also has several limitations which must be considered. As stated initially, though the subject programs are widely used by researchers, the fault types in XML-Security and Dcm may not be representative enough. Therefore, we cannot claim the results of the subject programs can be generalized to arbitrary programs.

On the other hand, threats to internal validity maybe mostly lie with possible errors in our tool implementation and the measuring of experiment results. To reduce such threats, we performed several careful checks. Another threat to internal validity may stem from our bias to perform the experiment. Though the Java application XML-Security is a real world application, the AspectJ application Dcm does not have existing bug records. Therefore, we seed several faults into the original Dcm program and augment it with a suite of tests. To reduce the threats of personal bias, we assign different people to seed faults and develop the test suite. These man-made faults are kept unknown before applying them to AutoFlow.

Analysis Cost. The analysis performed by AutoFlow runs in practical time. Our experiment is conducted on a DELL C521 PC with AMD Sempron 3.0 Ghz CPU and 1.0 GB memory. For the Java application XML-Security and the AspectJ application Dcm, the total running time (including compilation time) of our analysis is 546s and 95s, respectively. We believe the overheads are reasonable as an initial implementation, though there is still room for further performance optimizations.

5. RELATED WORK

We next discuss some closely related work in areas of fault localization techniques and delta debugging.

Recently, researchers have proposed many ways [7,12–14,16,17, 19,21,33] to automate the process of searching for faults. Delta debugging [27] is designed to be a general technique to minimize the suspicious changes for automated debugging. However, as stated in Section 1, it suffers several performance overheads. Aiming to improve it, our approach combines several static and dynamic analysis techniques to fast track failure causes.

Misherghi and Su [17] proposed a hierarchical delta debugging (HDD) algorithm to improve the effectiveness of delta debugging to minimize the test inputs. HDD generates orders of magnitude fewer test cases and scales to very large programs. However, HDD is mainly concerned with the *simplification* algorithm of delta debugging instead of the *isolation* algorithm. HDD works well on hierarchical structured inputs like a program's abstract syntax tree or XML document. However, the changes between software versions may not be a good candidate for HDD. For example, when adding several fields or methods to an existing class, there are no hierarchical relationships between these changes. Instead, the logical dependencies between software changes (captured by dependencies between atomic changes) may be more useful to assist fault localization. In this work, we explore the semantic relationship between atomic changes to facilitate the process of debugging.

There is also a lot of work [18, 22, 35] devoted to change impact analysis and its application to fault diagnosis. Ryder et al. [22] first use atomic change representations to perform change impact analysis for Java programs. Our previous work [32] is an extension of the concept of atomic changes to AspectJ programs. *Crisp* [6], a debugging tool for Java, is based on the atomic change representation to find fault locations by manually constructing intermediate program versions. Stoerzer [23] et al. also presented a classification tool *JUnit/CIA* built on JUnit and Chianti [20] which classifies Java atomic changes with respect to the tests they affect to identify the likely source edits of test failure. Unlike our approach, the exploration of fault using these tools requires profound domain knowledge of the target program (select the suspected changes based on a user's decision) and highly involved manual efforts (may need to select, apply, and roll back suspected changes repeatedly until find the faulty one), and therefore can not be fully automated.

In [19], Ren and Ryder proposed a heuristic to rank Java program edits for fault localization. Their heuristic relies on the static information (like the number of ancestors of a specific node) provided by the call graph of one specific failed test. They only consider the **CM** (*change method*) change in Java programs. In our work, we utilize the dynamic test execution information of both passing and failing test and present a heuristic based on the testing coverage. Jones and Harrold [14] proposed to visualize testing results to ease the fault localization. They instrumented each statement in a program and defined a score for it. Jeffrey et al [9] proposed a value profile based approach for fault localization, and their approach outperforms [14] in the empirical evaluation. However, their work aims to use the dynamic information to find suspicious statements during test execution and has not taken the software changes into consideration. While our work uses analysis of aspect-aware call graphs to get the coverage information for each atomic change (that is, at method-level), to facilitate the fault localization process.

6. CONCLUDING REMARKS

In this paper, we presented a hybrid approach to effectively locate failure-inducing changes in software evolution. Our approach combines both static and dynamic analysis techniques for fault localization. We also presented AutoFlow, a general automatic debugging framework for Java and AspectJ programs. The experiments on both Java and AspectJ applications confirm that this hybrid approach can significantly reduce the number of tests to run.

We recommend that this hybrid approach be an integrated part of the delta debugging technique; each time a regression test fails, irrelevant changes should be pruned out and a delta debugging that hierarchically organizes the responsible changes with proper priority should be started to resolve the regression cause.

As our future work, we plan to examine alternative techniques like dynamic program slicing [34] to improve delta debugging. We also intend to investigate the cost/effectiveness tradeoffs when incorporating program analysis techniques into debugging tasks.

7. REFERENCES

[1] AspectJ compiler 1.5, May 2005.
http://www.eclipse.org/aspectj//.

[2] AspectJ Development Tools (AJDT).
http://www.eclipse.org/ajdt//.

[3] GDB: The GNU Project Debugger.
http://gnu.org/software/gdb.

[4] The Eclipse Development Platform.
http://www.eclipse.org/.

[5] The XML-Security Project.
http://santuario.apache.org/.

[6] O. Chesley, X. Ren, and B. G. Ryder. Crisp: A debugging tool for Java programs. In *Proc. International Conference on Software Maintenance (ICSM'2005)*, Budapest, Hungary, September 27–29, 2005.

[7] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 210–220, New York, NY, USA, 2002. ACM.

[8] R. A. DeMillo, H. Pan, and E. H. Spafford. Failure and fault analysis for software debugging. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, pages 515–521, Washington, DC, USA, 1997. IEEE Computer Society.

[9] N. G. Dennis Jeffrey and R. Gupta. Fault localization using value replacement. In *In ISSTA '2008, Seattle, WA*, July 2008.

[10] M. D.Ernst. Static and dynamic analysis: Synergy and duality. In *ICSE Workshop on Dynamic Analysis, pages 24-27*, May 2003.

[11] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[12] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[13] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[14] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, New York, NY, USA, 2002. ACM.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, pages 220–242. 1997.

[16] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *PLDI 2003*, 38(5):141–154, 2003.

[17] G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 142–151, New York, NY, USA, 2006. ACM.

[18] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. Harrold. An empirical comparison of dynamic impact analysis algorithms, 2004.

[19] X. Ren and B. G. Ryder. Heuristic ranking of java program edits for fault localization. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 239–249, New York, NY, USA, 2007. ACM.

[20] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.

[21] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *In Automated Software Engineering, 2003.*, 2003.

[22] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *PASTE'01*, pages 46–53, 2001.

[23] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in java programs using change classification. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 57–68, New York, NY, USA, 2006. ACM Press.

[24] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 190–201, New York, NY, USA, 2006. ACM Press.

[25] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *In Proc. of the 29th International Conference on Software Engineering(ICSE07)*, pages 65–74, May 2007.

[26] M. Young. Symbiosis of static analysis and program testing. In *Proc. 6th International Conference on Fundamental Approaches to Software Engineering,*, pages 1–5, April 2003.

[27] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proc. 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.

[28] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

[29] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Autoflow: An automatic debugging tool for AspectJ software. In *Proc. 24th International Conference on Software Maintenance (ICSM 2008 demo)*, Oct 2008.

[30] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Autoflow: An automatic framework for AspectJ programs. In *Technical Report, SJTU-CSE-08-03*, Jan 2008.

[31] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Celadon: A change impact analysis tool for Aspect-Oriented programs. In *Proc. 30th International Conference on Software Engineering (ICSE 2008 Companion)*, May 2008.

[32] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Change impact analysis for AspectJ programs. In *Proc. 24th International Conference on Software Maintenance (ICSM 2008)*, Oct 2008.

[33] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 272–281, New York, NY, USA, 2006. ACM.

[34] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *In ICSE '2003, Portland, Oregon*, 2003.

[35] J. Zhao. Change impact analysis for aspect-oriented software evolution. In *Proc. 5th International Workshop on Principles of Software Evolution*, pages 108–112, May 2002.