Flota: A Programmer Assistant for Locating Faulty Changes in AspectJ Software Evolution

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

ABSTRACT

As Aspect-Oriented Programming (AOP) wins more and more popularity, there is increasing interest in using aspects to implement crosscutting concerns in object-oriented software. During software evolution, source code editing and testing are interleaved activities to assure code quality. If regression tests fail unexpectedly after a long session of editing, it may be difficult for programmers to find out the failure causes. In this paper, we present Flota, a fault localization tool for AspectJ programs. When a regression test fails unexpectedly after a session of source changes, Flota first decomposes the differences between two program versions into a set of *atomic changes*, and then identifies a subset of affecting changes which is responsible for the failure. Programmers are allowed to select (and apply) suspected changes to the original program, constructing compliable intermediate versions. Thus, programmers can re-execute the failed test against these intermediate program versions to locate the exact faulty changes by iteratively selecting, applying and narrowing down the set of affecting changes. Flota is implemented on top of the *ajc* compiler and designed as an eclipse plugin. Our preliminary empirical study shows that Flota can assist programmers effectively to find a small set of faulty changes and provide valuable debugging support.

1. INTRODUCTION

During software development process, coding and testing are interleaved activities to assure code quality. Normally, when new program functionalities are implemented, or an existing program is modified, the updated software version needs to be regress tested to validate these changes. After a long code editing session, regression tests are executed to ensure that the changes in the updated program version do not conflict with previous releases. In this phase, any test case that produces unexpected result may indicate potential defects in the updated software. Difficulties occur when regression tests reveal unexpected behaviors, such as assertion failure or exceptions. Sometimes, although the programmer knows that he has introduced a bug, he still does not know which part of editing should be responsible for the bug. If the edits are trivially small, it may be easy to find the faulty changes by inspecting all these modified places manually. However, as a code base and its test suite grow in size, running all tests after each minor change become infeasible, and the number of changes between successive executions of the test suite is likely to increase. In such cases, examining each place of source modifications and pinpointing the few that introduces the failure become a burden task for programmers. Moreover, edits are inter-related in many ways especially when developing large software system, and there can be more than one change that should be responsible for the failed tests.

Aspect-Oriented Programming (AOP) [5] has been proposed as a technique for improving separation of concerns in software design and implementation. In AspectJ software, with the inclusion of *join point*, an aspect woven into the base code is solely responsible for a particular crosscutting concern, which raises the system's modularity. However, when locating failure-inducing changes during AspectJ software evolution, it involves more complex situations:

- ***Control flow and data dependence between aspect and base code***. Since the woven aspect may change control and data dependency to the base code, adding or changing the aspect code can significantly affect the semantics of whole program.
- ***Multiple Advice Invocation***. While multiple advices apply at the same join point, *precedence rules* [2] determine the order in which they execute. The interactions between the base and aspect code or even the aspect weaving sequences will also dramatically affect the program behavior.
- ***Non-local Failure Causes***. Failures of regression testing could either result from changes in the base code or a particular aspect. The more complex cases raise from the interactions between the base and aspect code or even the aspect weaving sequences. In such a case, no single location corresponds to the failure, and the difficulty of finding failure-inducing changes rises dramatically.

Although many fault localization techniques have been presented in the literature, most of the work is focused on procedural or object-oriented software [4, 10], seldom effort has been made for aspect-oriented software. In this paper, we present Flota, a new fault localization tool for AspectJ programs. Our goal is to provide programmers with tool support to assist the debugging process of AspectJ programs. Flota uses the source level *atomic change* [4, 11] representation to capture precisely the semantic differences between two AspectJ program versions. Then it constructs static AspectJ call graph for the failed test to identify a subset of affecting changes. Programmers can select and apply (or rollback) the suspected atomic changes to the original program, constructing compliable intermediate versions. Therefore, programmers can re-execute the failed tests against these intermediate program versions to locate the exactly failure-inducing reasons by repeatedly selecting, applying and then narrowing down the set of affecting changes.

LATE Linking Aspect Technology and Evolution Workshop, April 1st, Brussels, Belgium

Flota constructs the compilable intermediate program versions automatically and programmers do not need to be concerned with the syntactic inter-relationship of source changes. We present an experimental study on the Tracing benchmark [1] using Flota. The result shows that Flota can effectively reduce the number of responsible changes and provide valuable debugging support.

The rest of this paper is organized as follows. Section 2 briefly introduces the background of atomic changes and change impact analysis approach. Section 3 presents our fault localization approach implemented in Flota. Section 4 reports an empirical evaluation on the Tracing benchmark. The related work and concluding remarks are given in Section 5 and Section 6, respectively.

2. BACKGROUND AND EXAMPLE

We next briefly introduce the background of atomic changes and change impact analysis for AspectJ programs. Figure 1 show a small AspectJ example program containing classes `BaseFee`, `TaxFee`, and `ExtraFee`, and aspect `PositiveFeeChecker`. Associated with the program are three JUnit [3] tests shown in Figure 2. The original program consists of all program fragments except for those marked by underline. We assume a sequence of source modifications as shown in Figure 1. The editing parts to the original version are all new added and marked by underline.

2.1 Atomic Changes in AspectJ

In our previous work [11], we extended the concept of atomic changes [4] to AspectJ programs, and identified a catalog of atomic changes (shown in Table 1). Those atomic changes represent the source code modifications at a coarse-grained model (that is, method-level), which is amenable to analysis. Most of the atomic changes in Table 1 are self-explanatory except for **AIC**. As defined in [11], the formal definition of **AIC** is:

AIC =

$$\{<j, a> \mid <j, a> \in ((J' \times A' - J \times A) \cup (J \times A - J' \times A'))\}$$

where J and A are the sets of join points and advices in the original program, and J' and A' are the sets of join point and advices in the modified program. $J \times A$ denotes the matched join points and advice tuple set in the original program while $J' \times A'$ denotes the matched tuple set in the updated program version. **AIC** change reflects the semantic differences between the original program and the edited program; and indicates that the advices invoking at certain join points have been changed.

Additionally, there are semantic dependencies between atomic changes. Intuitively, 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 causes a syntactically invalid program that contains some, but not all of the atomic changes. The syntactic dependence relationship (i.e., in above example, C_2 is a *prerequisite* for C_1) between atomic changes is crucial to construct intermediate program versions. Due to the space limitation, for the semantic dependence rules between atomic changes, please refer to [12].

2.2 Change Impact Analysis

Our fault localization technique relies on the change impact analysis [11] of AspectJ programs. Change impact analysis is performed by Celadon[1] to decompose the source code changes between two program versions into a set of atomic changes. Celadon also builds the semantic dependencies between atomic changes. Flota then uses the output of Celadon, to construct syntactically valid intermediate program versions. These intermediate program versions contain some, but not all of these atomic changes. Notice that, if a set of atomic changes likely contains a bug, then applying certain subsets of that changes will not lead to a buggy program. Thus, the construction of intermediate programs allows us to localize faults more effectively by ignoring the irrelevant changes, focusing our attention on viable, interesting ones.

[1] Our change impact analysis framework for AspectJ programs [11].

```
public class BaseFee {
    public int totalNum, singleNum;
    public BaseFee(int singleNum) {
        this.singleNum = singleNum;
    }
    public int calculateNum() {
        totalNum = singleNum*12;
        return totalNum;
    }
    public int getTotalNum() {
        return calculateNum();
    }
}
public class TaxFee extends BaseFee {
    public int taxNum;
    public TaxFee(int num, int taxNum) {
        super(num);
        this.taxNum = taxNum;
    }
    public int calculateNum() {
        totalNum = super.calculateNum() + taxNum;
        return totalNum;
    }
}
 public class ExtraFee extends TaxFee {
  private final int extraNum = 10;
    public ExtraFee(int num, int taxNum) {
        super(num, taxNum);
    }
  public int calculateNum() {
      totalNum = super.calculateNum() + extraNum;
      return totalNum;
  }
}

public aspect PositiveFeeChecker {
  pointcut singleFeeCheck(BaseFee base) :
   execution(* BaseFee.calculateNum()) && this(base);
  before(BaseFee base) :singleFeeCheck(base) {
      if(base.singleNum < 0) {
          base.singleNum = 0;
      }
  }
  pointcut taxFeeCheck(BaseFee tax) :
    execution(* TaxFee.calculateNum()) && this(tax);
  before(BaseFee tax):taxFeeCheck(tax) {
      if(((TaxFee)tax).taxNum < 10) {
         ((TaxFee)tax).taxNum = 10;
      }
  }
}
```

Figure 1: A Sample AspectJ Program.

Figure 3 shows the atomic changes inferred from the source edits in Figure 1. Each atomic change is shown as a box, where the top half of the box shows the category of the change, and the bottom half shows the 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.

Example: In Figure 3, the addition of advice: `before(BaseFee tax): taxcheck(tax)` is represented by atomic change 6 (**AEA**: *Add Empty Advice*), which depends on atomic change 5 (**ANP**: *Add new Pointcut*) because the new added advice uses the pointcut declaration `taxFeeChecker`. Atomic change 6 (**AEA**) would lead to a syntactically invalid program unless the referred pointcut is also added (i.e., atomic change 5). Therefore, atomic change 5 is a prerequisite of atomic change 6. The careful reader may also find that there is a **CAB** change (atomic change 8) which depends on atomic change 6. This is because in our change impact analysis model,

```
public class TestFees extends TestCase {
    public void testBaseFee() {
        BaseFee fee = new BaseFee(100);
        int totalNum = fee.getTotalNum();
        assertTrue(totalNum == 1200);
    }
    public void testTaxFee() {
        BaseFee fee = new TaxFee(50, 20);
        int totalNum = fee.getTotalNum();
        assertTrue(totalNum == 620);
    }
    public void testExtraFee() {
        BaseFee fee = new ExtraFee(0, 10);
        int totalNum = fee.getTotalNum();
        assertTrue(totalNum == 10);
    }
}
```

Figure 2: Test Case for motivating example

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 1: A catalog of atomic changes in AspectJ

we decompose the source code editing of adding a new advice into two steps: the addition of an empty advice (i.e.,atomic change 6: **AEA**), and the insertion of the advice body (i.e., atomic change 8: **CAB**), where the later is dependent on the former. Similarly, the deletion cases are in a reverse order.

Another core part of our change impact analysis approach is the call graph representation [7, 11] for AspectJ programs. Call graph is constructed to determine: (1) the affected tests after program changes; and (2) the affecting atomic changes for each affected test. A test is determined to be affected if, (1) its call graph contains a node that corresponds to a base code change [8], like a changed method (**CM**) or deleted method (**DM**) or contains an edge that corresponds to a lookup change **LC**, and (2) its call graph contains a node that corresponds to an advice body change (**CAB**), delete empty advice change (**DEA**), modify inter-type method body (**CIMB**), or delete the inter-type method (**DIM**) or contains an edge that corresponds to an advice invocation change **AIC** or contains a node involved in an **AIC** change. Using the call graphs [2] in Figure 4, we can easily find: (1) `testBaseFee` is not affected, and (2) `testExtraFee` and `testTaxFee` are affected, because their call graphs each contains a node for `TaxFee.calculateNum()` which involves in the **AIC** change (The ninth change in Figure 3).

[2]Celadon constructs static call graphs for safely analysis, but it can also work with dynamic call graphs generated from execution traces.

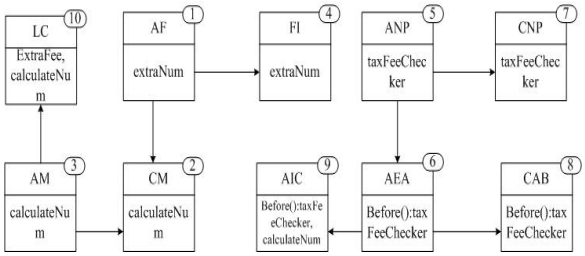

Figure 3: Atomic changes inferred from the example program, with their dependencies

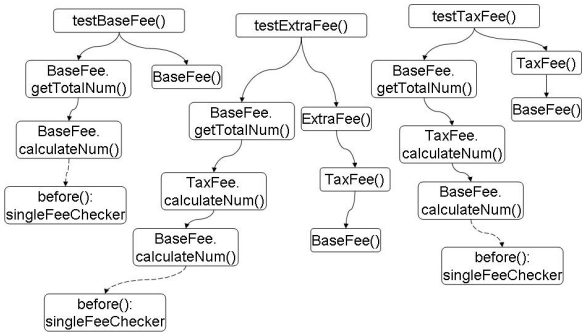

Figure 4: Call graphs for the tests in the original programs

The call graphs for the affected tests (`TestFees.testTaxFee` and `TestFees.testExtraFee`) are shown in Figure 5, the set of atomic changes that affect a given test includes: (1) atomic changes occurred in the base code, changes like changed methods (**CM**) and added methods (**AM**) that correspond to a node in the edited call graph, and changes like lookup change(**LC**) that corresponds to an edge in the call graph; (2) the atomic changes appearing in the aspect code, including changes of adding new advice (**AEA**), changing advice body (**CAB**), introducing new inter-type declared method (**INM**) or changing inter-typed declared method body (**CIMB**) that corresponds to a node in the call graph. The affecting atomic changes also include the advice invocation change (**AIC**) that corresponds to an edge in the call graph; and (3) the *aspect precedence* and *soften exception declaration* changes (correspond to the **AAP**, **DAP**, **ASED** and **DSED** atomic changes) that crosscut the affecting aspects and classes. The whole affecting atomic change set also includes the transitively prerequisite atomic changes of all above changes.

Example: We use shadows to annotate the modified method or advice in Figure 5. The call graph of `TestFees.testTaxFee` contains a node corresponding to atomic change 8 and an edge labelled <`before():taxFeeCheck`,`TaxFee.calculateNum()`>, which corresponds to the atomic change 9. Atomic change 9 depends on atomic change 5 and 6. Therefore, `TestFees.testTaxFee` is affected by atomic change 5,6, 8 and 9. Similarly, the call graph of `TestFees.testExtraFee` contains method `ExtraFee.calculateNum()` corresponding to atomic change 2 which depends atomic changes 1 and 3, an edge corresponding to atomic change 10, and an edge corresponding to the atomic change 9 which depends on change 5 and 6. Consequently, test `ExtraFee.calculateNum()` is affected by atomic changes 1, 2, 3, 5, 6, 8, 9 and 10.

3. FAULT LOCALIZATION APPROACH

The original program version passes all the three tests in Fig-

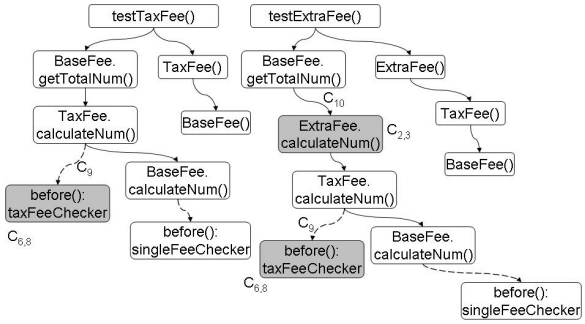

Figure 5: Call graphs for the tests in the updated programs (the call graph of `testBaseFee` remains unchanged after modifications)

ure 2, but test `testFees.testExtraFee` fails after the source editing. As shown in Figure 3, there are totally ten atomic changes from the source editing, in which eight are identified to affect `testFees.testExtraFee` by Celadon. The question is: *which of those eight changes are the likely reason(s) for the test failure?* Our tool Flota provides programmers fault localization help by allowing automatic construction of valid intermediate program versions containing certain suspected atomic changes. Flota works as follows: first, a programmer selects the suspected atomic changes that he thinks are the most likely failure reasons, then Flota automatically constructs an intermediate program version by applying the selected changes and all their prerequisites to build a syntactically correct program. The programmer can then re-execute the failed tests against the intermediate program version. If the tests pass, the programmer can ignore the applied changes that do not result in the failure, and continues to narrow down the remaining smaller set of changes until they locate the exactly reasons.

Flota also provides programmers a *rollback* function that allows them to undo their selection, and restore the original program. Therefore, programmers can construct the intermediate versions iteratively by applying the suspected changes.

Example: For `TestFees.testExtraFee`, there are eight atomic changes that may be responsible for the test failure. Programmer may first guess the new added advice invocation is cause of test failure and then he selects change 9 and 8 to apply. Flota automatically applies atomic changes 5, 6 and 7 prior to applying atomic change 9 and 8 to the original program to construct a intermediate program version, which is shown in Figure 6.

```
pointcut taxFeeCheck(BaseFee tax) :
  execution(* TaxFee.calculateNum()) && this(tax);
before(BaseFee tax):taxFeeCheck(tax) {
   if(((TaxFee)tax).taxNum < 10) {
       ((TaxFee)tax).taxNum = 10;
 }
```

Figure 6: Intermediate program version by applying atomic change 8 and 9

When programmer re-executes `Tests.testExtraFee` and finds it passes, he ignores the five applied changes and focuses on the last remaining three atomic change.

Though a toy motivating example to illustrate our fault localization approach, the debugging support provided by Flota can be useful to real world software and the benefits of having tools to assist in locating faulty changes are undeniable (discussed in Section 4), especially when there are tens or hundreds changes. In our approach, the syntactic dependence between each atomic change is calculated automatically and programmers only need to focus on the valid, interesting intermediate program versions.

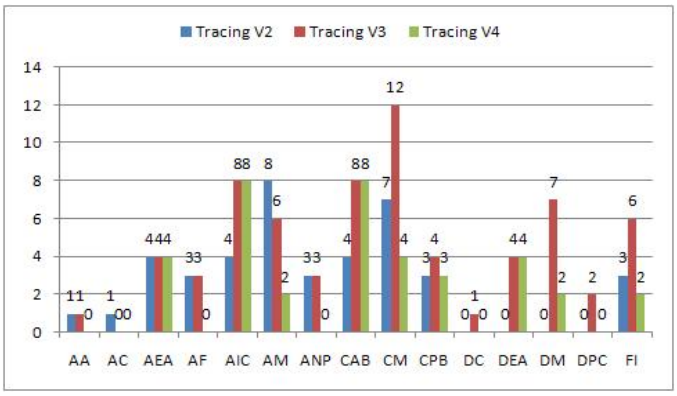

Figure 7: Number of atomic changes between each version pair of Tracing benchmark

4. EMPIRICAL EVALUATION

To evaluate our proposed technique, We performed a preliminary empirical evaluation on the Tracing benchmark using Flota.

4.1 Tracing benchmark

Programs	#Loc	#Ver	#Me	#Shad	#Tests	%mc	%asc
Tracing	1059	4	44	32	15	100	100

Table 2: The Tracing benchmark

The Tracing benchmark is included in the AspectJ compiler example package [1], and has been widely used by other researchers to evaluate their work. Table 2 shows the number of lines of code in the original program (#Loc), the number of versions (#Ver), the number of methods (#Me), the number of shadows (#Shad), the size of the test suite (#Tests), the percentage of methods covered by the test suite (%mc), and the percentage of advice shadows covered by the test suite (%asc) [3]. For this benchmark, we made the first version v_1 a pure Java program by removing all aspectual constructs and developed a test suite for it.

4.2 Experiment Result

4.2.1 Atomic Changes

The atomic change categories between each successive versions are shown in Figure 7[4]. Note that not every category of atomic change occurs between each version pair. There are total 15 categories of changes among three version pairs, in which eight are aspect-related changes. The most frequent aspect-related change is **AIC**, while the overall most frequent change is **CM**.

4.2.2 Affected Tests and Affecting Changes

The number of affecting changes and affected tests are shown in Figure 8. Interestingly, 100% of the tests in each version are affected, because version v_2 adds two pointcuts: `execution(* *(..))` and `execution(* new(..))` which crosscut base Java methods and are always executed at runtime. In version v_3 and v_4

[3] An *Advice shadow interaction* occurs if a test executes an advice whose pointcut statically matches a shadow. The *Advice shadow coverage* is the ratio between *Advice shadow interaction*s and the number of shadows in program.

[4] The atomic changes between version v_{n-1} and v_n are shown in bar v_n.

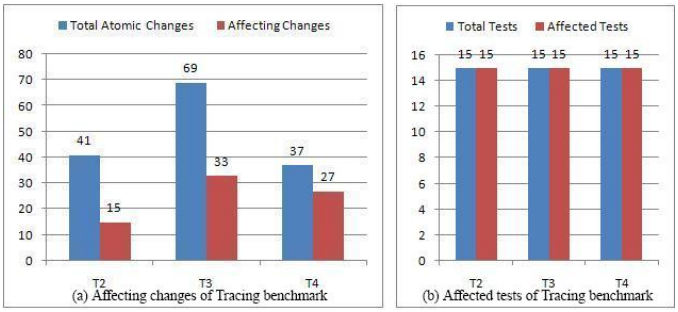

Figure 8: (a) Number of affected changes, affecting changes, and (b) affected tests for each version pair of Tracing benchmark

of Tracing benchmark, this two pointcuts and their associated advices are all modified. For the affecting changes, in each version, 36.5%, 47.8% and 72.9% of the total atomic changes are responsible for the affected tests.

4.2.3 Debugging Using Flota

It is a challenge to find appropriate test data for Flota to stimulate the debugging activities. However, we found one test, `SquareTest.testPerimeter` which passed in Tracing's 2nd version, but failed in its 3rd version. We use Flota to find the failure-inducing changes of this test. First, Celadon takes the Tracing program version v_2 and v_3 as inputs, and generates atomic changes to represent the source code changes. Second, for the failed test `SquareTest.testPerimeter()`, Celadon lists its affecting atomic changes. Finally, the output of Celadon is passed to Flota.

There are totally 8 affecting changes for the failed test, which account for 13.5% of the total number. First, we selected the **AIC** changes, which is caused by the new added advice `after(): perimeterAndArea()`. This **AIC** change depends on other 4 changes. Then, Flota constructs an intermediate program version only containing these 5 changes. We re-execute the failed test cases against this intermediate version and find it passed. Next we select another change **CM** (`Square.square(double,double,double)`) and find it is responsible for the program failure. In order to confirm our result, we apply other atomic changes to the original version except for **CM** and re-executed `testPerimeter()`, which then succeeded. This indicates that **CM** is the only failure-inducing change.

4.3 Discussion of Experiment Result

In the experiment, we demonstrate the potential ability of Flota to identify the faulty changes in AspectJ programs. During the experiment, programmers can ignore certain changes that do not result in the failure, and narrow down a smaller set of changes until they locate the exactly failure reasons. From the experiment, we find Flota can effectively reduce the number of responsible changes when a specific test fails. In the Tracing example, Flota isolates 33 affecting changes from a total number of 69 changes, and then reduces 8 responsible changes to 2 after two iterations, which is much more efficiency than manually inspecting one by one.

The method level coarse grained atomic changes simplify the exploration process. However, the programmer may have to consider several related affected tests together with their affecting changes. For AspectJ programs, faults may be caused by multiple source changes, so the exploration of intermediate program versions may also correspond to changes from multiple places. Therefore, in Flota implementation, we provide a *roll back* function to allow programmers undo the applied changes to ease the process of identifying failure-inducing edits. For more experimental study and implementation issues, please refer to [12].

5. RELATED WORK

We next discuss some related work in the area of change impact analysis and fault localization techniques.

Recently, many change impact analysis [6,9,13] techniques have been proposed, which are mainly focused on object-oriented languages. Ryder et al. [9] first use atomic changes to perform change impact analysis for Java programs. They presented a catalog of atomic changes and a sophisticated definition of dependencies between atomic changes as well as their analysis tool in [8]. However, they focused on the Java language features, and our previous work [11] is an extension of the concept of atomic changes to aspect-related constructs to perform impact analysis for AspectJ programs.

Perhaps the most similar work to ours is the *Crisp* debugging tool for Java programs described in [4]. They use a novel approach to find fault locations in Java programs by isolating the likely failure-inducing changes. In this paper, we extends their fault localization technique to handle the unique features of AspectJ programs. Our tool Flota is based on the AspectJ change impact model [11], and is aimed to provide debugging support in AspectJ software evolution.

6. CONCLUDING REMARKS

In this paper, we present Flota, a tool for locating faulty changes in AspectJ software evolution. The fault localization technique is based on the *atomic change* representation and change impact analysis technique. Flota isolates failure-inducing changes from others by constructing intermediate program versions, and assists programmers to locate the likely failure reasons. In our preliminary experimental study, we find Flota can be helpful in reducing the number of failure responsible changes and improving the effectiveness of locating faulty changes in AspectJ programs.

As our future work, we intend to investigate the effectiveness of our approach on larger AspectJ programs. We also want to integrate the delta debugging approach to support automatic debugging.

Acknowledgements

This work is supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 60673120), and Shanghai Pujiang Program (Grant No. 07pj14058). We would like thank Cheng Zhang, Si Huang and Chong Shu for their help.

7. REFERENCES

[1] The AspectBench Compiler. `http://abc.comlab.ox.ac.uk/`.
[2] The AspectJ Team. The AspectJ Programming Guide. Online manual, 2003.
[3] Junit, Testing Resources for Extreme Programming, 2006.
[4] 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.
[5] 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.
[6] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proc. 25th International Conference on Software Engineering*, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.
[7] N. Li. The call graph construction for aspect-oriented programs. Master's thesis, School of Software, Shanghai Jiao Tong University, March 2007 (in Chinese).
[8] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proc. OOPSLA 2004*, pages 432–448, Vancouver, BC, Canada, October 26–28, 2004.
[9] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *PASTE'01*, pages 46–53, 2001.

[10] 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.

[11] S. Zhang and J. Zhao. Change impact analysis for AspectJ programs. Technical Report SJTU-CSE-TR-07-01, Center for Software Engineering, SJTU, Jan 2007.

[12] S. Zhang and J. Zhao. Locating faults in AspectJ programs. Technical Report SJTU-CSE-TR-07-03, Center for Software Engineering, SJTU, Sep 2007.

[13] 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.