AutoFlow: An Automatic Debugging Tool for AspectJ Software

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

Aspect-oriented programming (AOP) is gaining popularity with the wider adoption of languages such as AspectJ. During AspectJ software evolution, when regression tests fail, it may be tedious for programmers to find out the failure-inducing changes by manually inspecting all code editing. To eliminate the expensive effort spent on debugging, we developed ***AutoFlow****, an automatic debugging tool for AspectJ software. AutoFlow integrates the potential of delta debugging algorithm with the benefit of change impact analysis to narrow down the search for faulty changes. It first uses change impact analysis to identify a subset of responsible changes for a failed test, then ranks these changes according to our proposed heuristic (indicating the likelihood that they may have contributed to the failure), and finally employs an improved delta debugging algorithm to determine a minimal set of faulty changes. The main feature of AutoFlow is that it can automatically reduce a large portion of irrelevant changes in an early phase, and then locate faulty changes effectively.*

1 Introduction

Programmers often spend a significant amount of their time debugging programs in order to reduce the number of bugs in software releases. Typically, when regression tests fail unexpectedly after a long session of editing, the search of suspicious changes is an arduous, highly manual involved and time-consuming process. The high cost of fault localization causes in software evolution has motivated the development of automatic debugging techniques such as [1,2].

Aspect-Oriented Programming (AOP) [3] has been proposed as a technique for improving separation of concerns in software design and implementation. AspectJ, one of the most widely used AOP languages, is designed as a seamless extension to Java. The new features of AspectJ program present new challenges for program analysis tasks. When designing programming tool supports, besides the object-oriented features like sub-typing and dynamic dispatching, the unique aspectual constructs like advice, pointcut and inter-type declaration should also be handled appropriately. Though the executable code of AspectJ software is pure Java bytecode, the existing debugging techniques for Java can not be applied directly to the bytecode, since there is a significant discrepancy between AspectJ source code and the woven bytecode. Therefore, an alternative approach taken in AutoFlow implementation is to perform source-code-level static analysis for AspectJ programs.

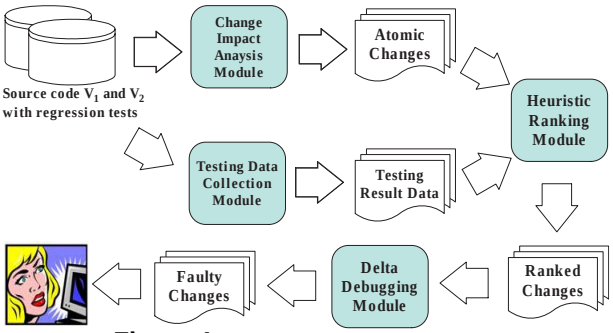

Figure 1. Overview structure of AutoFlow.

When a regression test fails unexpectedly after a session of source changes, AutoFlow works as follows, first it decomposes the code modifications into a set of *atomic changes* (at method-level); then it employs change impact analysis to isolate a subset of responsible changes for that failed test; in the third step, AutoFlow ranks these changes according to our proposed heuristic, and finally employs an improved delta debugging algorithm to determine a minimal set of faulty changes.

2 AutoFlow Tool

AutoFlow [6] is implemented on top of the ajc compiler and designed as an eclipse plugin. The overview structure of AutoFlow is shown in Figure 1. We next present the implementation details of its four main components.

Change Impact Analysis Module. We implemented the change impact analysis module based on our Celadon [7] framework. This module decomposes the

source editing into a set of atomic changes [4, 8]. The atomic changes generated in this module are used to reflect the semantic differences between the initial and updated software version. There are also inter-dependencies between atomic changes, which will be future explored to construct compliable intermediate program versions for isolating failure-inducing changes. For the failed test, this analysis module also identifies a subset of responsible changes based on the aspect-aware call graph construction. For the detail explanation of our change impact analysis approach used in this module, please refer to [8].

Testing Data Collection Module. This module executes all the regression tests, including both passed and failed ones, in context of the updated software version, and collects the testing result. It also constructs static AspectJ call graph for each test case. The collected testing result and the constructed call graphs are passed to the heuristic ranking module as inputs.

Heuristic Ranking Module. The responsible changes for the failed test and the testing data collected are fed into this module. We use the following heuristic to rank all responsible changes, to indicate the likelihood that they may contribute to the failure:

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

In this equation, for one specific responsible change c, %failed(c) is a function that returns, as percentage, the ratio of the number of failed tests that cover c as an affecting change[1] to the total number of failed tests in the test suite. Likewise, %passed(c) is a function that returns, as a percentage, the ratio of the number of passed test cases that cover c as an affecting change to the total number of passed test cases in the test suite.

This heuristic utilizes the result provided by many test case executions instead of only the failed test. It can help the developer understand more complex relationships in the system, instead of the limited information provided by few test cases.

Delta Debugging Module. The responsible changes after ranking are passed to this module. AutoFlow employs the delta debugging algorithm to automatically determine a minimal subset of faulty changes. By utilizing all above information, the delta debugging algorithm used in this module improve the effectiveness of the original algorithm [5] in the following aspects:

- ***Searching Space.*** The original delta debugging algorithm searches the entire set of source changes for identifying the faulty ones. However, in our approach, for a specific failed test, a large set of uncorrelated changes can be ignored by change impact analysis, and we only focus on the responsible changes.
- ***Change Selection.*** Delta debugging selects and applies the changes (the configurations) randomly. However, in our approach, the changes that are most likely to contribute to the failure are ranked highest and applied first.
- ***Handle Inconsistence.*** One of the most important practical problems of delta debugging is *inconsistent configurations*. When combining changes in an arbitrary way, it is likely that several resulting configurations are inconsistent. However, in our implementation, the inter-dependencies between atomic changes guarantee the syntactic correctness of the constructed intermediate program versions[2].

AutoFlow have been applied to several AspectJ benchmarks to evaluate its ability for identifying different kinds of bugs [6]. For most cases, AutoFlow outputs the correct result in form of faulty *atomic changes*. Since the delta debugging algorithm is able to handle configuration interference situations when locating faults, therefore, even there is a combination of multiple failure-inducing changes, AutoFlow can still find them.

Acknowledgements This work was 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).

References

[1] 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.

[2] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *In Proc. Int'l Conf. Softw. Eng., 2002*.

[3] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th ECOOP*, pages 220–242. 1997.

[4] 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.

[5] 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.

[6] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. AutoFlow: An automatic debugging framework for AspectJ programs. Technical Report SJTU-CSE-TR-08-01, Center for Software Engineering, SJTU, Jan 2008.

[7] 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.

[8] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Change impact analysis for AspectJ programs. In *Proc. 24th IEEE International Conference on Software Maintenance*, Sep 2008.

[1] As defined in [8], the set of affecting changes that affect a given test includes the transitively prerequisite of all atomic changes appearing on its call graph.

[2] Since when applying one atomic change to the original program, its syntactic dependent changes will also be applied automatically.