Identifying Semantic Differences in AspectJ Programs

Martin Th. Görg
Department of Computer Science and Engineering
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai
nimoth@cs.tu-berlin.de

Jianjun Zhao
Department of Computer Science and Engineering
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai
zhao-jj@cs.sjtu.edu.cn

ABSTRACT

Program differencing is a common means of software debugging. Although many differencing algorithms have been proposed for procedural and object-oriented languages like C and Java, there is no differencing algorithm for aspect-oriented languages so far. In this paper we propose an approach for difference analysis of aspect-oriented programs. The proposed algorithm contains a novel way of matching two versions of a module of which the signature has been modified. For this, we also work out a set of well defined signatures for the new elements in the AspectJ language. In accordance with these signatures, and with those existent for elements of the Java language, we investigate a set of signature patterns to be used with the module matching algorithm. Furthermore, we demonstrate successful application of a node-by-node comparison algorithm originally developed for object-oriented programs. Using a tool which implements our algorithms, we set up and evaluate a set of test cases. The results demonstrate the effectiveness of our approach for a large subset of the AspectJ language.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Algorithms, Experimentation, Performance

Keywords

AOP, AspectJ, difference analysis, static analysis, semantic analysis, disjunctive matching

1. INTRODUCTION

Program differencing is a common means of software debugging [15]. Software systems remain complex despite efforts to separate, modularize, and encapsulate problems. In fact, the modularization can make it especially difficult to understand the control flow of the program. The task is particularly complicated for aspect-oriented programs. Aspect-Oriented Programming (AOP) has been proposed as a means to improve separation of concerns for the design and implementation of software [5, 14, 16, 21]. In AOP, complex pointcuts, multiple advice invocation, and cross-aspect precedence rules are common obstacles for understanding and maintaining aspect-oriented programs. A simple modification can have unforeseen effects on the program's behavior. Program difference analysis can aid debugging by automatic detection of behavioral changes. It can also be used by other frameworks for further program analysis. Regression test selection frequently uses program differencing to select all tests which execute one or more of the found changes [25, 7, 17]. Change impact analysis, as carried out by Celadon [27] or by program slicing techniques [2, 22, 24], often relies on syntactic differencing to detect the changes on which to perform further impact analysis. Accurate detection of changes is vital for analyses of this kind. Substituting semantic for syntactic differencing would enable them to operate even more precisely.

It is necessary to extend existing program differencing techniques as well as to develop new ones to handle the new programming concepts introduced by AOP. Of the several existing techniques concerned with semantic program differencing [3, 4, 7, 9, 10, 12, 15, 25], only the work of Xu et al. [25] can also be used for aspect-oriented programs. During AspectJ code compilation an intermediate step, called *weaving*, produces regular Java code [8]. Thus, one could easily apply existing OOP byte code analysis tools. However, it would be impossible to accurately determine the correct location of changes. Moreover, the woven code may vary between different compilers and versions. Consequently, applying differencing to woven Java byte code cannot yield safe results.

In this paper, we present a differencing algorithm to work with programs written in the popular AOP language AspectJ [23]. However, the techniques we employ deal with general programming and design paradigms which are applicable to other AOP realizations as well. Our algorithm first identifies matching modules of two program versions, P and P', and then uses an extended Control Flow Graph (CFG) representation, based on the one developed by Huang and Zhao [11], to apply a hammock graph comparison algorithm [3, 4, 15]. This paper makes the following contributions:

- It presents a novel algorithm, called *disjunctive match-*

ISSTA'09, July 19–23, 2009, Chicago, Illinois, USA.

```
aspect Constraints {
  public boolean Shape.immovable = false;
  void around(Shape s) :
    execution(public * Shape+.set*(..))
                        && args(s) {
    if(!s.immovable) {proceed(); }
  }
}
```

Figure 1: Aspect for adding movement constraints to graph elements

ing, to identify matching modules.

- It proposes a set of signatures for AspectJ modules along with patterns for the disjunctive matcher.
- It applies the hammock graph comparison algorithm to aspect-oriented programs.
- It gives a set of evaluations to show the effectiveness and feasibility of the proposed approach.

The rest of this paper is organized as follows: In the next section, we provide background information of important technologies used in this paper. In Section 3, we provide a motivating example which we will use throughout the paper. In Section 4 we give a detailed description of our differencing algorithm. In Section 5, we provide tests and evaluations using a tool implementation. Section 6 discusses related work. Finally, Section 7 contains our conclusions and a preview of future work.

2. BACKGROUND

2.1 AOP and The AspectJ Language

AOP has been proposed as a means to isolate, compose, and reuse functionality which cuts across an object-oriented type hierarchy. The goal is to improve separation of concerns [5, 14, 16, 21]. AspectJ is an implementation of AOP. It is a seamless extension to the Java language and it defines a new type, called *aspect*, as well as a new set of predefined keywords. The predominant construct in an aspect is the *advice* construct. It is a set of statements executed *before*, *after*, or *around* a designated point in the program. That point is said to be *advised* by the advice. The pattern, which defines the point being advised, is called the *pointcut designator* and the points, which can be specified by a pointcut designator, are called *join points*. Another major feature of AOP and AspectJ in particular is *inter-type definition* (ITD) for methods, constructors, and fields. An ITD is defined in an aspect, but it is declared on another type effectively adding a new field or method to another class. In this way, it is possible to inject additional functionality into another class. Figure 1 shows an example of an ITD and an around advice. The around advice intercepts all calls to setters of shapes and uses the ITD to determine whether or not the execution is allowed to proceed. Existing classes do not need to be modified.

2.2 Hammock Graphs

In the CFG of a program, subgraphs with a single entry and a single exit node can be identified as *hammock* graphs [13, 6, 26, 15].[1] More precisely, using the definition from Ferrante et al. [6], let G be the CFG of a program P. A hammock H is an induced subgraph of G with a distinguished node n_s in H called the *entry node* or *start node* and a distinguished node n_e not in H called the *exit node*, such that all edges from $(G-H)$ to H go to n_s and all edges from H to $(G-H)$ go to n_e.

Figure 3(a), shows a CFG where the subgraphs encircled by dashed lines have been identified as hammocks. The hammock entry nodes are marked yellow, the exit nodes are marked green. Apiwattanapong et al. [3, 4] also define that a hammock is *minimal* if there exists no other hammock which (1) has the same start node and (2) contains a smaller number of nodes. Hereafter, whenever we use the term hammock, we refer to a minimal hammock, unless otherwise specified.

3. MOTIVATING EXAMPLE

In this section we present a simple example to illustrate how our proposed differencing technique supports maintenance and debugging of aspect-oriented programs.

Consider a drawing application with a package *geometry* which contains types like *Point*, *Line*, or *Rectangle*. Figure 2(a) shows part of a possible implementation of type Point. Some operations require a shape to be locked to its current position. The developers decide to use the aspect shown in Figure 1 to control the setters of all shapes. Figure 3(a) shows the CFG for the around advice. The encircled hammocks can be ignored for this example. For simplicity, we only consider the type *Point* here. Later, in the course of code cleanup and refactoring, a developer decides to reduce the visibility of the setters of shapes, because they are only accessed from inside the *geometry* package. He first removes the *public* keyword from the method *setY* of type *Point*, but he forgets to adjust the pointcut of the around advice. As a result, the movement constraint is no longer applied to the method *setY*. Figure 3(b) shows the resulting CFG of the around advice when applied to the new version of type *Point*. There is no node for a possible call to *Point.setY* anymore.

We may now consider two possible scenarios. One is that there exists a complete test suite. The developer can use a tool which implements our differencing algorithm to detect that the around advice behaves differently in version 1 and version 2 of the program. This result can be used to perform a regression test selection to determine that the tests which invoke the around advice need to be run again. If a test fails, the developer can quickly discover the broken constraint check.

Another possible scenario is that there is no test suite but while using the program an erroneous behavior is observed (e.g. a point is movable when it should not be). The developer can then use our differencing algorithm to realize that his change in visibility caused the around advice to behave differently and modify its pointcut accordingly.

Without the use of our differencing algorithm, it takes more effort to detect the error in the modified version, because the *Constraints* aspect and the *Point* class might reside in different locations of the application. Moreover, the developer implementing the constraints using AOP might be

[1] Laski and Szermer [15] initially used the term *cluster* instead of hammock.

```
1  public class Point extends Shape {
2    private int x, y;
3    public void setX(int i) {
4      x = i;
5    }
6    public void setY(int i) {
7      y = i;
8    }
9  }
```

(a)

```
1  public class Point extends Shape {
2    private int x, y;
3    public void setX(int i) {
4      x = i;
5    }
6    void setY(int i) {
7      y = i;
8    }
9  }
```

(b)

Figure 2: Version 1 and version 2 of type Point with a change in visibility.

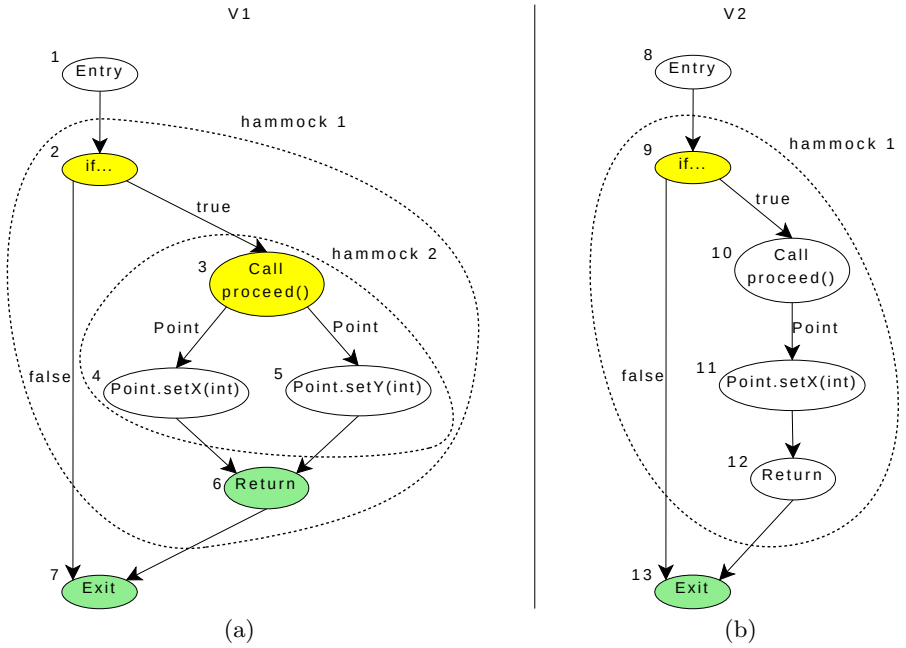

Figure 3: Hammock graphs for the around advice Constraints applied to version 1 and 2 of type Point from Figure 2

a different developer from the one who is undertaking the code refactorings. The latter might not even be aware of the constraint check. Therefore, it is difficult for him to find the problem.

4. DIFFERENCING ALGORITHM FOR ASPECTJ

Our differencing algorithm can be subdivided into three main tasks. Given a program P and a modified version of P called P':

1. For every module in P, find a matching module in P'.
2. Build extended CFGs for all modules in P and P' and identify hammocks.
3. Perform node-by-node comparison on every pair of hammock graphs.

The output of the algorithm is a set of modified and unmodified node pairs, a set of removed nodes and a set of added nodes. These sets represent changed statements, unchanged statements, statements deleted in P, and statements added in P', respectively. From the sets, a developer or programmer can obtain the differences between the two versions.

4.1 Module Matching

A module is considered to be a method or constructor, an advice, or an inter-type method or constructor. Pointcuts are not handled separately but are included in the referencing advice instead. Module matching is an important part of the algorithm because it dominates the results of the node comparison algorithm of Section 4.3. The matching is made difficult by two factors. First, a change to a module's signature denies simple signature matching. Second, not every AspectJ construct has a unique signature. Therefore, we propose a set of signatures for the elements of an AspectJ aspect and describe an algorithm, which solves the problem of finding corresponding modules.

4.1.1 AspectJ Signature Definitions

We propose sound definitions in order to distinguish between any two elements of an AspectJ aspect. Some of these elements (e. g. declare declarations) are not explicitly matched in the proposed differencing algorithm. We give their signatures here only for completeness and contribution to further research. Following are our proposed signatures. For each signature definition, the bold text elements form the signature. All other parts originate from the AspectJ or Java syntax. ITDs and pointcuts already have signatures defined by the AspectJ language specification, which is why we omit them here. For advice and declare declarations we propose signatures that aim to correspond with what we have observed to be common modifications during software development.

Advice declaration:
[strictfp] AdviceSpec [throws TypeList] : **Pointcut**{Body},
where AdviceSpec is of the form
before (**Formals**)
after (**Formals**)
after (**Formals**) **returning** [(**Formal**)]
after (**Formals**) **throwing** [(**Formal**)]
Type **around** (**Formals**)

Parents declaration:
declare parents : **TypePat extends** Type;
declare parents : **TypePat implements** TypeList ;

Warning and error declaration:
declare (warning | error) : **Pointcut** : String ;

Soft declaration:
declare soft : **Type** : **Pointcut** ;

Precedence declaration:
declare precedence : **TypePat1** [, TypePat2 [, . . .]] ;

Our proposed advice signatures contain three parts – namely the advice kind, the formal parameters, and the pointcut. If two advices $a1$ and $a2$ are of the same kind, have the same pointcut designator, and define the same type in their returning or throwing clause (if existent), then $a1$ and $a2$ are by definition applied to the same set of join points. If $a1$ and $a2$ additionally expose the same context by specifying the same formal parameters, then their body is most likely to also contain the same functionality. If so, then $a1$ and $a2$ are the same, which justifies our signature definition.

The signature of a parent declaration contains the type pattern and the keyword for implementation or extension. Java allows only single inheritance. Therefore, the same type pattern may only occur once together with the keyword extends. An implement declaration, however, allows a list of types to be specified. This list is likely to change in the future. Such a change does not represent an added but a modified statement and should thus be identified as such. Accordingly, the list of implemented types is not part of the signature.

Warning and error declarations have only the pointcut as their distinguishing part because two warnings or errors declared on the same pointcut can be merged into one without loss of functionality, and the message string does not qualify for a signature. For a soft declaration, the AspectJ language requires writing multiple declarations for the same pointcut, in order to wrap more than one exception. Therefore, the signature for a soft declaration must include all elements.

Our precedence declaration signatures include only the first type pattern, because any two precedence declarations with the same first type pattern can either be merged into one or the first type pattern can be removed without loss of functionality. The signature does not include more than the first type pattern, because the pattern list is likely to change with aspects being added or removed during development.

4.1.2 Disjunctive Matching

We first match modules based on their signatures. If they match, then the node-by-node comparison algorithm will be performed on their hammock graphs. Otherwise, the module-matching algorithm continues to look for a different pair of matching modules. However, matching modules only by their signature does not yield sufficient results. Any change to a module's signature would prevent a match with the original module. Apiwattanapong et al. [3] try to increase matched methods by looking only at the method identifier and leaving out the formal parameters. We extend their approach and propose a whole set of module-matching patterns for the new AspectJ elements as well as additional matching patterns for regular Java methods and constructors. Multiple patterns are applied using logical-OR connectors. All matched modules are therefore the union of the module sets matched by each matching pattern. We refer to this kind of module matching as *disjunctive matching*.

When proposing a matching pattern, one has to consider two things.

1. How likely is the pattern to find corresponding modules with different signatures?

2. How likely is the pattern to match non-corresponding modules (false positives)?

If test results are unsatisfactory, one can remove signature elements to increase the matching possibility, or add additional non-signature elements to reduce false positives. During this adjustment, one has to avoid application of two matching patterns where one matches a subset of the modules of the other one. The more sophisticated pattern, which matches the subset, only slows down performance because its matched modules are already included by using the simpler pattern.

Following are the patterns employed in our algorithm. An asterisk (*) indicates any number of non-white space characters and two dots (..) indicate any number of formal parameters. This syntax corresponds to the pointcut designator patterns of AspectJ. Compare the patterns to the signatures given in Section 4.1.1

Methods:
<package>.<type> <name> (..)
OR *.<type> <name> (<args>)

Advices:
<package>.<type> <kind> (..) * (..) : <pc>
OR <package>.<type> <kind> (<args>)
<returning/throwing> (<args>) : *
OR <package>.<type> * (<args>) * (<args>) : <pc>

Inter-type methods and constructors:
. <type>.<name> (..)
OR <package>.<type> *.<name> (..)

The disjunctive matching creates a problem. It creates the possibility that multiple matches are found. We solve the problem in our algorithm by performing the node-by-node comparison algorithm on all possible matches. The

module pair with the highest number of matched nodes is considered to be the best match.

4.2 CFG Construction and Hammock Identification

An algorithm for the construction of CFGs for aspect-oriented programs has been developed by Huang and Zhao [11]. The algorithm can handle a subset of object-oriented and aspect-oriented programs including the predominating concepts of the two languages. Exception handling, dynamic pointcut designators, and declare precedence declarations are not supported. This limits the set of programs for which our algorithm can correctly detect all differences in behavior. However, the set of supported programs suffices to demonstrate the success of our program differencing approach, because it is only necessary to extend the CFG construction in order to support additional features. Other parts of the algorithm can be applied to the extended CFGs without need for modification. We will address the shortcomings in our future work. Nevertheless, for modules that make use of currently unsupported OOP features, a valid CFG is constructed. The unsupported parts are either ignored or summarized in a single node.

4.2.1 Representation of Object-Oriented Paradigms

We partially adopt the methods of Apiwattanapong et al. [4] to augment the node and edge labels of the constructed CFGs. Using the extend information, we can model the object-oriented features of polymorphism, object types, and type hierarchies as illustrated in the following paragraphs.

Polymorphism.

Using static analysis, the dynamic target of a polymorphic function call cannot always be determined. Therefore, for a call node we create an outgoing edge for each possible dynamic target type. The edge is labeled with the type name and the edge's destination node contains information to determine the actual place of invocation (i. e. the declaring type). Thus, we model all possible dynamic invocations using only static analysis.

Types and hierarchies.

A change in the type of a variable can have significant impact on a program's behavior each time the variable is being used. It is therefore necessary to properly model variable types and their hierarchies into the CFG representation. For variables of a primitive type a change should be indicated every time the variable is referenced. To achieve this, our algorithm simply prefixes the type to the identifier at all occurrences. A change to the type of a non-primitive variable, i. e. to an object, will become apparent at any method invocation of that variable because possible callees are represented in the CFG, as mentioned above. This technique works not only for a change in a variable's type, but also for any kind of change to the type's hierarchy. The change will be visible at the point where it has effect. Only *cast* and *instanceof* expressions require explicit comparison of the inheritance tree of the referenced types, because a modification to the referenced type's hierarchy can result in a runtime exception or even silently alter the program behavior. During CFG construction, we augment all cast and instanceof expressions with the complete inheritance tree of the referenced type. For performance reasons, java.lang.Object is not included in the hierarchy list, except for the class Object itself. We use the following inheritance tree representation format to augment our CFGs:

$$I_1^A, I_2^A, \ldots, I_n^A, A : I_1^B, \ldots, I_n^B, B : \cdots : I_1^{Ref}, \ldots, I_n^{Ref}, Ref\,,$$

where $I_i^{\mathcal{X}}$ is the i-th interface implemented by type $\mathcal{X}$, $\mathcal{X} \underset{lex}{>} \mathcal{Y} \Rightarrow$ ($\mathcal{X}$ is a super type of $\mathcal{Y}$), with $\underset{lex}{>}$ being the lexicographical order. Ref is the actual referenced type. Furthermore, to ensure correct comparison, for each class in the hierarchy, all of its implemented interfaces are also sorted by lexicographical order as defined in String.compareTo [20], regardless of the order given in the source code. The resulting character sequence is the *globally-qualified name* [3] of the referenced type. In Figure 4 we show how, for example, the class java.util.LinkedHashMap would be represented when referenced in a cast or instanceof expression.

java.util.Map,java.util.AbstractMap:java.io.Serializable,
java.lang.Cloneable,java.util.Map,java.util.HashMap:
java.util.Map,java.util.LinkedHashMap .

Figure 4: Inheritance tree of java.util.LinkedHashMap

4.2.2 Representation of Aspect-Oriented Paradigms

In this section, we explain how our algorithm represents AOP features and properly maps the weaving of advice onto the CFG. The CFG construction algorithm correctly models inter-type declarations, declare declarations, pointcuts, and advice, including correct precedence handling for application of multiple advices to the same join point. The algorithm does not support declare precedence declarations or dynamic join-points. We leave them for our future work.

Inter-type declarations.

An inter-type field, method, or constructor can be represented analog to a regular field, method, or constructor, respectively. There is no additional handling necessary. Changes within an ITD are visible once at the point where the ITD resides – which is the containing aspect. Changes to the signature of an ITD are visible at each point the ITD is referenced.

Declare declarations.

Declare declarations also do not require explicit representation in the CFG. In Section 4.2.1, we explained how changes to the type hierarchy become apparent implicitly through the CFG construction. The same is true for hierarchy and method invocation changes caused by declare declarations in aspects.

Pointcuts.

A pointcut does not contain executable code for which a CFG can be built. Pointcuts are effective only in combination with a referencing advice. Therefore, the modeling of pointcuts is integrated into the modeling of advice. Changes in pointcut definitions are visible where they have semantic effect, which is at the referencing advice's point of application.

Advice.

An important element is the representation of advice weaving. The way this is done by our algorithm can be outlined as follows:

1. At every possible join point, find all advices which advise the current join point.
2. Sort the found advices in accordance to the precedence rules.
3. Iterate through the sorted list of advices and apply each advice to the current join point.

The precedence rules are applied as specified in The AspectJ Programming Guide [23]. The construction of the CFG is not the main contribution of this paper, therefore we refer to Huang and Zhao's algorithm [11] for more examples and detailed coverage of all possible combinations of advice application and precedence. Here, we only point out two major differences compared to their algorithm, which are necessary because, in our approach, we build module-level CFGs instead of whole-program CFGs.

First, if an advice matches a join point, only the advice's entry node is inserted between the weave and return nodes to indicate the invocation. Second, when weaving an around advice, the nodes surrounded by the advice will always be included exactly once, regardless of the number of proceed calls contained in the advice. Changes to proceed calls from within the advice become apparent in the advice's own CFG. Therefore, we do not need to check for proceed calls when weaving the advice. Figure 3 shows how the proceed call is represented in our approach.

4.2.3 Hammock Identification

The goal of the hammock identification is to encapsulate single-entry single-exit regions starting with a decision node. We use the algorithm described in Ferrante et al. [6] to identify hammocks. When a hammock is found, it is enclosed within an artificial pair of hammock entry and exit nodes, which allow "collapsing" [3] the hammock to a single node called *hammock node*. In the example of Figure 3 the encircled hammocks are each reduced to a single node.

4.3 Hammock Graph Matching

An important benefit of constructing hammock graphs is that we can straightforwardly apply the hammock graph comparison algorithm from Apiwattanapong et al. [3]. The fact that our hammock graphs are constructed from CFGs for aspect-oriented programs instead of pure object-oriented programs is abstracted.

The algorithm takes as input two hammock graphs H and H', which have been constructed from P and P', respectively, and returns a set N of matched node pairs with label *modified* or *unmodified*. The sets of added and removed nodes are inferred by regarding every node in H which is not included in N as removed and regarding every node in H' not included in N as added.

The algorithm operates by simultaneously traversing the two given graphs using their labeled edges. At every step, the current two nodes are compared either by their labels or, in the case of two hammock nodes, by the ratio of their contained unmodified node pairs to the number of nodes in the smaller hammock. This ratio is called the *similarity* [3] of two hammocks. To obtain the ratio, the algorithm recursively compares the subgraphs of the two hammock nodes. A threshold value S for the minimum required similarity is provided as a parameter to the algorithm.

If during traversal two nodes $x \in H$ and $x' \in H'$ do not match, a *lookahead* technique is employed to find a matching node further down the control flow of each other's graph. The algorithm keeps comparing x with successors of x' and x' with successors of x until 1) a match is found or 2) the maximum number of nodes to look ahead, LH, is reached. In the first case, the found pair is added to N as *unmodified* and graph comparison continues from these two nodes. Nodes which did not match while looking ahead are skipped and will be marked as added or removed, if they are not matched again in another recursion of the algorithm. In the second case, x and x' are added to N as *modified* and graph comparison continues from these two nodes. The lookahead threshold LH must be provided as a parameter. For the pseudo-code of the algorithm we refer to the original paper. Instead, we show a partial run of the procedure to provide an example which effectively illustrates the nature of the algorithm.

As explained in Section 3, Figure 3 depicts the two graphs constructed from the around advice Constraints (Figure 1) when applying it to the two versions of type Point, respectively, shown in Figure 2. Because of the change of visibility, the around advice is no longer applied to the method *setY* and will therefore not be called by the proceed in the advice.

In the example, hmMatch first compares the two entry nodes. They match, so the pair $\langle 1, 8\rangle$ is added to the return set N. Next, the two hammock nodes representing *hammock 1* in *V1* (Figure 3(a)) and *V2* (Figure 3(b)) are processed. Because they are both hammock nodes, they are expanded and hmMatch recursively continues matching of their contained nodes. Nodes 2 and 9 match and are added to ${N_1}^2$ as unmodified. The following step will first try to match the hammock node representing *hammock 2* with node 10. They do not match, so the algorithm tries to look ahead in both graphs and finds node pair $\langle 3, 10\rangle$, which is then added as unmodified to N_1. The hammock node for *hammock 2* will be considered deleted in *V2*. Information contained in each edge label allows the algorithm to correctly resume comparison with nodes 4 and 11. Node 5 will never be processed and therefore also be considered deleted in *V2*. The remainder of the current recursion adds pairs $\langle 4, 11\rangle$ and $\langle 6, 12\rangle$ to N_1. *hammock 1* of *V2*, the smaller one, contains four nodes and N_1 contains four unmodified matched node pairs. Thus, the similarity to *hammock 1* of *V1* is 1.0 and the two hammocks are considered unmodified and all node pairs in N_1 are added to N. In the last step, the exit nodes 7 and 13 are also matched and added to N and the algorithm terminates. This example should suffice to demonstrate the algorithm.

4.4 Time Complexity Analysis

Our results of Section 5 show that the node-by-node comparison algorithm is the dominating part of our differencing approach. Let m and n be the number of all nodes in all CFGs of the two programs P and P', respectively, and let n_{max} be the global maximum number of nodes in a single module. Let d be the maximum hammock nesting level, and

[2]Subscripts distinguish between the return sets of different recursions.

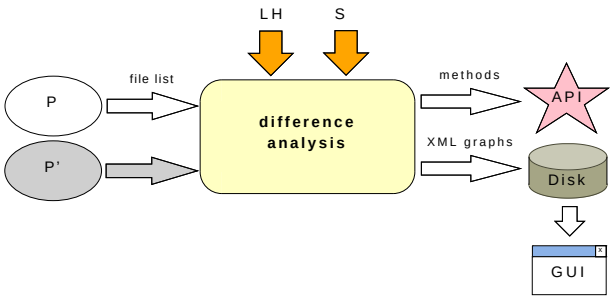

Figure 5: AJDiffer architecture

let $l > n_{max}$ be the maximum lookahead. Then, as already determined by Apiwattanapong et al. [3], the complexity for the node level comparison is $O(n_{max} \cdot \min\{d, l\} \cdot \min\{m, n\})$. This is because matching at different hammock levels can lead to the same pair of nodes being matched more than once. When two hammock nodes are compared, their contained nodes are compared in the recursive call. If lookahead is performed later, the same nodes will be processed again. Therefore, the complexity incorporates the maximum nesting depth and lookahead.

5. TESTS AND EVALUATION

This section contains a set of tests we conducted using a tool implementation of our algorithm to validate the proposed difference analysis algorithm. First, we test what kinds of changes the algorithm is capable of detecting and the correctness of the results. Second, we test the time performance of the implemented tool. There is currently no other tool available to automatically verify the results obtained from our algorithm. Therefore, in the first evaluation part, we use programs of limited size to illustrate the differencing results. The second part, on the other hand, focuses on execution time disregarding the identified differences, because it is not feasible to manually check these for a large program.

5.1 Tool Implementation

The tool *AJDiffer*, implements our proposed difference analysis algorithm for aspect-oriented programs written in AspectJ. *AJDiffer* itself is also a Java application written in AspectJ. It serves as a validation and evaluation means for our algorithm.

Figure 5 depicts the process-oriented flow of *AJDiffer*. The difference analysis can be viewed as a black-box, as far as the user is concerned. There is no interaction on his side needed while the tool computes the differences. As input, the tool takes two versions of a program, P and P', the maximum lookahead LH, and the hammock similarity threshold S. All arguments are passed upon invocation. The two programs are represented by paths to a text file, which contains a list of all source files to be analyzed.

AJDiffer provides two kinds of output to serve different purposes. Program analysis tools, which incorporate difference analysis as part of their algorithms, may benefit from the Application Programming Interface (API), which provides a number of methods to obtain the differencing results and carry out further analysis on them. *AJDiffer* also allows to output the results in XML format. One possible application to use the XML-output is the Graphical User Interface (GUI) provided by the *ajframework*, developed by the Software Theory And Practice Group (STAP) [19][3]. The GUI is build for end-users and for testing and evaluation. It displays the matched module graphs to give visual feedback on the differencing results.

Table 1: A catalog of atomic changes in AspectJ

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

5.2 Differencing Results

The goal of this evaluation is to show that the results of our algorithm correctly reflect the changes in behavior. For best results, the algorithm needs to detect as many matching node pairs as possible. A less precise algorithm will match less nodes and instead mark more nodes as added or removed. Table 3 shows the results of ten differencing tests. The test programs are taken from the examples of the AspectJ compiler distribution and from the *ajbenches* package of the AspectBench compiler [1].

We run our differencing tool with two different versions of each test program (also denoted by $v1$ and $v2$). For the *ants*, *bean*, *coma-stack*, *introduction*, and *spacewar* example, we manually applied a set of changes to create complex situations with multiple advices applying at the same join point and major advice invocation changes. For the other five examples, we used different versions provided in the example packages. To classify the changes, we use the definitions for atomic changes in AspectJ, given by Zhang et. al [27]. Changes to the base code are classified using the notation of Ryder and Tip [18]. For better illustration, we reproduce both tables of atomic changes here in Table 1, and Table 2.

For each test, we first obtain the lines of code (LOC) and the textual differences ($Diffs$) between the two program versions. We then classify the textual changes into the aforementioned atomic change patterns.

The last three columns show the results obtained from running our differencing tool. All tests are run using $LH =$

[3] *AJDiffer* is also part of the *ajframework*.

Table 2: Catalog of atomic changes in base code

Abbreviation	Atomic Change Name
AF	Add a field
DF	Delete a Field
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

20 and $S = 0.4$. In general a higher lookahead produces more precise results in exchange for slower performance as described in Section 4.4. However, a lookahead of about 20 is usually sufficient and yields the best results for our tests as well. The similarity threshold used for our evaluation is also empirically determined. A too low value will match too many non-corresponding modules; a too high value will fail to match modified but corresponding modules. The similarity of 0.4 has proven successful for our evaluations with only a minor impact on performance.

The number of affected nodes includes added, deleted, and matched but modified nodes. The number of matched nodes includes matches of modified as well as matches of unmodified nodes. The percentage indicates how many of all possible matches are found. Errors are marks which do not precisely reflect the kind of change (e. g., two nodes are matched while they should have been marked as added and deleted).

5.2.1 Individual Test Cases

The *ants* and *cona-stack* examples serve to evaluate the signature definitions and the disjunctive matching of our algorithm. In the *ants* example, the only changes are a renaming of a package (automaton to automaton_ren) and a change to a method signature (int getState() to int getState(Object dummy)) along with the necessary refactorings. The modified package and method are rarely referenced, so there are only four changes in total. However, the resulting output shows, that 26 nodes have been affected. This is due to the change in the signature of all methods invoked on the class in the changed type. Our algorithm detects each of those changes correctly. More important, all of the affected nodes are modified matched pairs. There are no added or deleted nodes, despite the fact that a package with one contained class has been renamed. A look at the kind of change patterns shows, that renaming the package technically results in the deletion and subsequent addition of one class and all its members. Based on our AspectJ signature definitions, the disjunctive matcher is able to detect the similarity in the members, which differ only by their package name. All of the nodes from a member method's graph are correctly matched. The entry and exit nodes of the graph are matched and marked as having been modified. The disjunctive matcher also proves successful when matching the original and modified version of the getState method.

In the modified version of *cona-stack*, we changed the type of an around advice from *before* to *after*. The result is a set of removed nodes before every advised join point and a set of added nodes after every advised join point. Here, again, the improvement through the disjunctive matching can be observed, because the changed advice itself contains only two pairs of modified matches for entry and exit. The interior graph nodes are marked as unchanged. Without a disjunctive matcher, all nodes of the original and modified graph would be marked as added and deleted, respectively.

The *bean* example incorporates some simple aspect-related changes, which are common during development of a program. We changed the functionality of two around advices, which advise setter methods, so that only a certain number of changes are allowed for a single object. To accomplish this, we introduced a new field and added a guard to the proceed call of the two around advices. This results in changes to method invocation and control flow, all of which were correctly identified by the differencing tool.

The test program *spacewar* serves as an example for minimal syntactic changes with maximal semantic effects. The debug version of the *spacewar* example contains an aspect that advises nearly all methods of all classes in the program. In order to avoid recursion, calls to the debugging aspect itself, its member class, InfoWin, and any of their subclasses are not advised. We used this exclusion and added a *declare parents* declaration on class SpaceObject, which has several subclasses. Thus, that class and all its subclasses are made subclasses of the InfoWin class. Two advices advise a total of 46 methods in the original version. Of these methods, two also include an advised constructor call. Every applied advice is represented by three nodes. An additional modified node pair is added (counting as one), because the initial super constructor call of class SpaceObject is modified to call InfoWin instead of Object. Altogether, 283 nodes or node pairs are affected. All other nodes, including the one modified pair, match. This is equivalent to the results produced by our tool implementation of the difference algorithm.

5.2.2 General Observations

The first general observation is that in many cases the number of affected statements[4] exceeds the number of textual changes. This shows that semantic differencing – as opposed to syntactic differencing – is necessary to obtain precise results. The second observation is that our algorithm detects close to 100 % of all possible matches and makes almost no errors when marking the changes.

The reason for the 78 erroneously marked nodes in the *nullcheck* example is the combination of variable inlining and variable name change. An optimal result would be to detect the new location of the statement despite the different variable name and identify the two nodes as a modified pair. Our algorithm correctly identifies a difference, but it marks the nodes as removed and added in the original and modified version, respectively. Nevertheless, our algorithm is still useful for this combination of change patterns, because all the locations in the program where the modification has effect are correctly determined. Only the kind of change is erroneous. The modification of the original *nullcheck* example affects three nodes which should be matched, but which are marked as added and deleted. The nodes are part of a newly introduced advice, which is applied to 26 methods. Therefore, we consider all of the 78 marked nodes ($3 * 26$) as errors.

The reason for the errors in the *introduction* example are swapped statements. Our algorithm currently cannot han-

[4]Two affected nodes approximately correspond to one affected statement.

Table 3: Results of differencing benchmarks ($LH = 20, S = 0.4$)

Program	LOC	Diffs	Change Patterns	Affected	Matched	Errors
ants	1451	4	AF, DF, AM, DM, CM, AC, DC, LC, AIC, CPB	26	2446 (100 %)	0
bean	199	19	INF, CIFI, INM, CIMB, CAB, AEA, ANP, CPB, AIC	30	268 (100 %)	0
cona-stack	381	1	AEA, DEA, CAB	9	730 (100 %)	0
dcm	3406	684	AA, INM, CIMB, AEA, CAB, ANP, CPB, AHC, AIC, AF, AM	1523	2771 (100 %)	0
figure	148	42	AA, AEA, CAB, ANP, CPB, AF, CM, DM,	101	177 (100 %)	0
introduction	233	4	CIMB, DIM, LM	18	362 (98.3 %)	6
nullcheck	2990	136	AA, AEA, CAB, ANP, CPB, CM, DM, DC	258	2828 (98.2 %)	78
quicksort	127	14	AEA, CAB, ANP, CPB	27	155 (100 %)	0
spacewar	3053	1	AHD, AIC	283	4496 (100 %)	0
tracing	333	55	AEA, DEA, CAB, ANP, CPB, DPC, AIC	164	442 (100 %)	0

dle these optimally, if the path between the two nodes representing the swapped statements is shorter than the provided maximum lookahead parameter. In that case, the nodes lying on that path will be marked as added and deleted, because the algorithm correctly detects the two swapped node pairs as matching using the lookahead technique. However, it then continues matching at their respective locations further down the graph. Thus, the nodes inbetween are left out. As in the problem illustrated by the *nullcheck* example, a difference is still detected by our algorithm. This time, however, more nodes are being marked than necessary. We will work on a solution for the problem in our future work.

Another observation from the differencing results is that our disjunctive matching technique is successful. Evidence for this is given by the *ants* and *quicksort* examples, where the signatures of several modules have been modified. Especially, the *quicksort* example incorporates changes to several advice signatures. Based on our AspectJ signature definitions, our tool employs the disjunctive matching algorithm and is able to determine which modules should be compared. As a result, all statements therein are successfully matched.

5.3 Performance Results

The goal of the performance tests is to show that our proposed algorithm is feasible for real-world applications. A common problem of AspectJ tool evaluations is the lack of large real-world applications. Therefore, we artificially increased the size of one of the modules by in-lining calls to other modules. This technique is useful, because the time-dominating node-level matching is performed independently for each pair of matched modules. We also added a few changes to the second version of the large module to produce more work for the hammock matching algorithm. Additionally, we combined nine of the previously used examples (*ants*, *bean*, *cona-stack*, *figure*, *introduction*, *nullcheck*, *quicksort*, *spacewar*, and *tracing*) into one application (hereafter called *combined*). We thereby obtain an application of approximately 9150 lines of source code with close to 306 textual differences and over 1000 affected nodes.

In Figure 6, we show, for a similarity threshold of 0, 0.6, and 1, how the run time increases when increasing the maximum lookahead. Time measurements are scaled to log for better illustration. The tests are carried out on a Pentium M 1.6 GHz laptop computer with 1 GB of memory, running Windows XP SP2. The maximum heap size of the JVM is set to 512 MB.

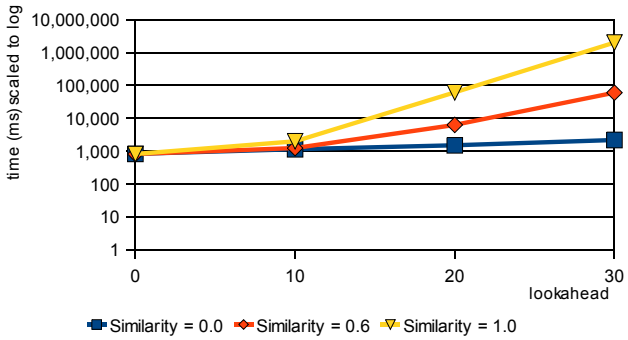

Figure 6: Results of performance benchmarks using the *combined* example (~ 9150 LOC)

The time measurements show the time needed to perform the matching of nodes using the hammock graph comparison algorithm. The time needed for all other parts of the algorithm remains constant at about 30 seconds.

From the results, we can observe that for $S < 0.6$ the lookahead has only minor impact on performance, and for $LH \leq 20$ the algorithm always performs within one minute. It also becomes apparent that performance decreases when using a similarity threshold of more than 0.6 in combination with a lookahead of more than 20. However, as shown in our first evaluation, a high similarity threshold is not essential to achieving good difference results. We obtained the best results with $S = 0.4$. The similarity threshold is the dominating factor for run time because with an increasing threshold it becomes more difficult to match two hammocks, which results in more nodes being compared multiple times.

5.4 Threats to Validity

The primary threat to external validity is the question to what degree the test examples are representative of ap-

plications in real practice. To oppose this threat, we paid attention to cover and combine as many change patterns as possible and create complex control flow situations, as described in Section 5.2. Furthermore, we artificially increased size and complexity of the examples, as explained in Section 5.3, to better match those of real-world applications.

Threats to the internal validity of our evaluation stem from possible errors in tool implementation and errors in our manual change analysis. We tackled the former by examining the tool's output for combinations of change patterns. To minimize the latter error, we employed third party code analysis tools to aid our evaluation.

6. RELATED WORK

Much related research exists in the field of difference analysis. The Unix *diff* tool performs textual differencing by solving the 'longest common subsequence' problem. This approach is fast and simple but it fails to identify semantic changes related to OOP or AOP paradigms.

Xu and Rountev [25] propose an algorithm, for regression test selection for AspectJ programs using source code analysis. They get closest to solving the differencing problem for AOP and also achieve a high precision in terms of false positives. However, they are looking for modified edges to do a test selection, whereas, in our approach, we are looking for modified nodes and present them directly as output. Another related research is provided by Apiwattanapong et al. [3, 4]. They first perform matching on the class level, then on the module level, and finally on the node level using a hammock comparison algorithm on extended CFGs. The test evaluations suggest this being a promising program differencing technique. Nevertheless, their algorithm is only designed for object-oriented programs. The node-level matching of our approach is based on their work. We also divide matching into different levels but omit the class level and substitute it with a powerful disjunctive matching algorithm at module level.

Horwitz [9] proposes a way to compute both syntactic and semantic differences using dynamic analysis. She uses a partitioning algorithm, which is based on a Program Representation Graph (PRG), but PRGs are defined only for a limited subset of programming languages. That is, it only supports scalar variables, assignment statements, conditional statements, while loops, and output statements. As a result, the technique cannot be used in general, especially not for aspect-oriented programs. Another dynamic analysis approach is the research done by Jackson and Ladd [12]. Their tool semantic diff performs procedure-level comparison like the algorithm of Apiwattanapong. But semantic diff does not represent polymorphisms like method overriding and therefore cannot capture all OOP paradigms. It is even less suitable for aspect-oriented programs.

7. CONCLUDING REMARKS

We developed an algorithm capable of identifying semantic differences between two versions of an AspectJ program. The success of our algorithm shows that type level comparison is not needed. Instead, we proposed an innovative technique for module-level comparison using a combination of signature based matching and so-called disjunctive matching. Omitting the class level is superior because ITDs in aspect-oriented programs are not bound to the declaring aspect but have a cross-aspect scope. Moreover, our two-level approach can easily handle the renaming of a type, which is a common code refactoring. Along with our module-level proposition, we defined a set of signatures for all AspectJ constructs and put forward matching patterns for both AOP and OOP constructs. As a result of using disjunctive matching, we are able to fully automate the program differencing. Previous works [4] depend on user interaction to find corresponding classes and modules having different signatures. For all module signature modifications which do not violate the defined set of matching patterns, our algorithm successfully identifies their correspondence. If desired, the patterns can be easily adapted. As a promising conclusion of our work, the basic idea of using hammock graphs for CFG comparison is applicable for aspect-oriented programs also, as long as corresponding modules are correctly identified and their CFGs model the language features sufficiently.

In our future work, we will further analyze common signature change patterns. Currently, the disjunctive matching patterns are based on programming experience and common sense. Optimally, they should be determined using statistics from the development of large real-world applications. We will also investigate an improvement of the node lookahead technique to solve the problem of swapped statements, which is the only change type that causes errors in our evaluations. Finally, we will work on completing CFG construction to support all features of AOP and OOP. In particular, we will address handling of *cflow* and *cflowbelow* pointcuts, which is a yet unsolved problem. Moreover, we will consider incorporating the approach of Xu et al. [25] to support dynamic pointcuts and exception handling.

8. ACKNOWLEDGEMENTS

This work was supported in part by the National Natural Science Foundation of China (NSFC) (Grant No.60673120), National High Technology Development Program of China (Grant No.2006AA01Z158) and Shanghai Pujiang Program (Grant No.07pj14058). We would like to thank Sai Zhang and Si Huang for their valuable discussions on this work.

9. REFERENCES

[1] abc. The AspectBench Compiler for AspectJ. http://abc.comlab.ox.ac.uk/.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, 1990.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold. A Differencing Algorithm for Object-Oriented Programs. In *Proc. of the 19th IEEE International Conference on Automated Software Engineering*, pages 2–13, Washington, 2004. IEEE Computer Society Press.

[4] T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engg.*, 14:3–36, 2007.

[5] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44:51–57, 2001.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, 1987.

[7] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. *SIGPLAN Not.*, 36:312–326, 2001.

[8] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proc. of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, New York, 2004. ACM.

[9] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, New York, 1990. ACM.

[10] S. Horwitz and T. Reps. The use of Program Dependence Graphs in Software Engineering. In *Proc. of the 14th International Conference on Software Engineering*, pages 392–411, New York, 1992. ACM.

[11] S. Huang and J. Zhao. A Framework and Tool Support for Control Flow Analysis of AspectJ Programs. Technical Report SJTU-CSE-TR-08-02, Shanghai Jiao Tong University, School of Software, Shanghai, 2008.

[12] D. Jackson and D. A. Ladd. Semantic Diff: A Tool for Summarizing the Effects of Modifications. In *Proc. of the International Conference on Software Maintenance*, pages 243–252. IEEE Computer Society, 1994.

[13] V. N. Kas'janov. Distinguishing hammocks in a directed graph. *Soviet Math. Doklady*, 16:448–450, 1975.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP 1997*, volume 1241 of *LNCS*, pages 220–242, Heidelberg, 1997. Springer.

[15] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Proc. of the International Conference on Software Maintenance*, pages 282–290, 1992.

[16] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44:39–41, 2001.

[17] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6:173–210, 1997.

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

[19] Software Theory and Practice Group (STAP), 2005. http://stap.sjtu.edu.cn/.

[20] Sun Microsystems. *Java Platform, Standard Edition 6 API Specification.* Sun Microsystems, 2007.

[21] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proc. of the 21st International Conference on Software Engineering*, pages 107–119, Los Alamitos, 1999. IEEE Computer Society Press.

[22] T. ter Braak. Extending Program Slicing in AspectOriented Programming with InterType Declarations. In *5th Twente Student Conference on IT*, 2006.

[23] The AspectJ Team. The AspectJ Programming Guide. http://aspectj.org.

[24] M. Weiser. Program slicing. In *Proc. of the 5th International Conference on Software Engineering*, pages 439–449, Piscataway, 1981. IEEE Press.

[25] G. Xu and A. Rountev. Regression Test Selection for AspectJ Software. In *Proc. of the 29th International Conference on Software Engineering*, pages 65–74, Washington, 2007. IEEE Computer Society Press.

[26] F. Zhang and E. H. D'Hollander. Using Hammock Graphs to Structure Programs. *IEEE Trans. Softw. Eng.*, 30:231–245, 2004.

[27] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Change Impact Analysis for AspectJ Programs. Technical report, Shanghai Jiao Tong University, School of Software, Shanghai, 2007.