Towards The Use of Program Slicing in The Change Impact Analysis of Aspect Oriented Programs

Imad Bouteraa
Noura Bounour
Towards the Use of Program Slicing in the Change Impact Analysis of Aspect Oriented Programs

Imad BOUTERAA *, Nora BOUNOUR **
** Laboratoire LRI,
*** Département d’Informatique
Université Badji-Mokhtar, BP. 12, 23000, Annaba, Algeria
*bouteraa.imed@gmail.com
**nora_bounour@yahoo.fr

Abstract:
Change impact analysis plays a crucial role in software maintenance; it can determine the effect of a change in an entity on the other entities of software. Several techniques of impact analysis for various paradigms were proposed in the literature. But little of them treat this problem in aspect-oriented programs.

In this paper, we propose a new approach for change impact analysis in aspect oriented programs. In order to get accurate results we use program slicing, a program analysis technique that explore existing dependencies in software source code.

Keywords: Change Impact Analysis, Aspect-Oriented Programming, Program Slicing.

1. INTRODUCTION

Software must frequently evolve in order to remain operational. This evolution implies that software is the subject of changes during its life cycle. A change in one entity can affect other entities, which directly or indirectly depend on it. And it’s there where the role of change impact analysis comes. Change impact analysis is the activity of identifying what to modify to accomplish a given change, or of identifying his potential consequences [1].

Among the new programming paradigms, one can find the Aspect-Oriented Programming (AOP). AOP is a technique that has been proposed for improving separation of concerns in software [4, 5]. With previous programming paradigms (e.g. object oriented programming), it was not an easy task to represent each concern by an only one artifact, which causes what is called crosscutting concerns. The basic idea of AOP for resolving the problem of crosscutting concerns is to represent every one of them by one artifact, which called aspect. And before executing the program a waver should wave those aspects with the rest of the code. While the code is not woven, the final behavior of an aspect oriented program remains unclear. Therefore, changing the structure of the program makes the control of the change effect difficult for the maintainers. So the needs of tools for analyzing the change impact increases with aspect-oriented programming.

Throughout this paper, we propose a new approach to solve the problem of change impact analysis in aspect oriented programs, with avoiding overwhelming the developer by irrelevant results. In our approach we use program slicing, a static program analysis technique that has not being stopped from use since its introduction [10]. The use of program slicing enables us to know which entities depend on a given entity, and on which entities it depends. Thus allows us to detect the effect of a change more precisely. We have chosen to apply our approach on AspectJ [4], an aspect-oriented language designed as an extension of Java.

Our paper is organized as follows: we first present the program slicing (Section 2), then slicing of aspect-oriented programs (section 3), then we proceed to our approach (Section 4), and we conclude with a conclusion in which we enumerate the prospects for continuation of this work.
2. PROGRAM SLICING

Program slicing is a program analysis and reverse engineering technique that reduces a program to statements that have a relationship with a particular computation. In other words, a slice answers the question "What are the program statements that may affect the values of a set of variables \( V \) in the statement \( S' \)?" [10, 9]. In this case the tuple \( <V,S> \) is called slicing criterion.

There are several types of program slicing. We quote among them the forward and backward slicing. The definition that we have presented in the previous paragraph is the backward slicing. Figure 1. represents a fragment of a program (a) with the backward slice (b) for the criterion \( <Total,14> \).

Concerning the forward slicing, it answers the question "What are the program statements that may be affected by the values of a set of variables \( V \) in the statement \( S' \)?" Figure 2. Represents a fragment of a program (a) with the forward slice (b) for the criterion \( <Y,2> \).

There are two principal approaches for calculating a slice in a program. The first proposed by Weiser [10] considers the problem of slicing as a data flow problem. Weiser used a control flow graph (CFG) as an intermediate representation for his algorithm, and proposed equations to calculate the slices, these equations are based on the calculation of the relevant variables at each node of the CFG according to the slicing criterion [10]. The second approach consists of: first generating an intermediate representation of the program that models the dependencies between its entities; and then calculating the slice by traversing the dependencies of this intermediate representation [3]. The intermediate representation is usually called system dependence graph (SDG). Horwitz et al. proposed a two-phase algorithm for the backward slicing and another one with two-phase for forward slicing. In those two algorithms, authors used a system dependence graph as intermediate representation [3].
parameters at the end of execution. A call vertex is added for each method call. Parameters of the call are represented by actual-in and actual-out vertices. Actual-in vertices represent input parameters of calls at call sites, and actual-out vertices represent output parameters of calls at call sites. Formal-in and formal-out vertices are control dependent on the entry vertices of their methods. Actual-in and actual-out vertices are control dependent on their associated call vertices.

The construction of the full SDG is performed by connecting call vertices to entry vertices of called methods by call edges, actual-in vertices to their corresponding formal-in vertices in the called methods by parameter-in edges, and formal-out vertices to their corresponding actual-out vertices in the caller methods by parameter-out edges. At the end, we get a SDG that contains all the MDGs interconnected according to method calls.

In this paper, we only consider methods in the system dependence graph, since the behavior of a program is basically defined by its methods. But in the literature [6, 2, 16, 8] they also represent the other object-oriented features and constructions as classes, packages, inheritance...etc.

There are several approaches to represent the SDG of an AspectJ program. For our approach, we chose the SDG of Xu and Rountev [11, 12, 13]; in their approach they use a Java’s SDG. And if an advice $A$ interacts with a join point $J$, they replace the shadow of $J$ (code corresponding to $J$) by a call to a virtual method (i.e. does not exist in the source code) $ph\text{-root}$. The role of this method is to call advices that interact with $J$ (C among others), $ph\text{-root}$ execute also the shadow of $J$ if it’s required (it’s not always the case, take the example of an around advice without a proceed).

Our choice for the SDG of Xu and Rountev returns to three points:
- The simplicity of the approach,
- Already implemented in an open source tool (AJAN [11, 13]),
- It can handle the case where several advices interact with the same join point.

Once we have the SDG of an AspectJ program, we can calculate its slices using the two-phase algorithm presented in [3].

4. THE PROPOSED APPROACH

The essential concept to spread the effect of a change is the dependence. Any change in one part can affects the parts that depend on it. Celadon [14, 15] is based on the principle that a method depends on methods that it calls. This principle seems to be promising, but it may give false positive results. Take the example of logging, a logger has no effect on its callers, while Celadon considers that a method depends on what it calls even the called is a logger. Similarly if the logger is an advice, it has no effect on its hosts\(^1\). This motivated us to propose a new definition of dependence in order to give more accurate results.

If we consider that a method takes input data and provides output data, we define the dependence between two methods as follows: "A method $a$ depends on the method $b$, if $a$ uses an output data of $b$". We define the use of data by a method as follows: "A method uses a given data $d$ if it has at least one statement that control or data depends on $d$". This control or data dependence leads us to the program slicing, since it’s designed to distinguish which entities depend on a given entity, and on which entities it depends.

4.1 ATOMIC CHANGES

Before proceeding to impact and dependencies analysis, we need to identify and categorize the changes that may exist between two versions of an AspectJ program. Basing on previous works [15, 7], we identified for our approach the atomic changes presented in Table 1. We distinguish two types of atomic changes: (a) the category of changes that reside in object part; and (b) those that reside in aspect part. We divide the second category into three sub-categories: (i) changes concerning aspects; (ii) those that concern introductions; and (iii) those that concern advices.

---
\(^1\) We mean by “Hosts of an Advice”, methods or advices to which belong the advised statements or bodies
Note that some changes exist in previous works [15], but they are not in ours. This intentional choice is up to two points: the first is that some atomic changes are not detectable from the source code only (e.g. the advices don’t have name, so we cannot detect a change in their bodies); The second is that the impact of some atomic changes is always detectable by other atomic changes, which causes some undesired redundancy (e.g. all changes concerning breakpoints cause additions or deletions of advice invocation).

4.2 DESCRIPTION OF THE APPROACH

We materialize our approach by a new system for impact analysis of AspectJ programs, we call it Souyoul.

Souyoul is composed of three main components (Figure 3.)

Atomic changes generator: it takes as input two versions of an aspect oriented program, makes the difference between them, and presents the result as a set of atomic changes.

Dependency graph generator: takes as input an aspect oriented program, and generates its system dependence graph basing on the approach of Xu and Rountev [11, 13].

Change impact analyzer: it’s the most important component of Souyoul, since we use the basic idea of our approach (i.e. program slicing) in this component. The change impact analyzer takes as input a set atomic changes and two system dependence graphs. Then for each change, it locates its corresponding entities in the system dependence graph, and from those entities, it traverse some edges of the SDG in the same direction of the arrows, which corresponds to forward slicing. The result of this step is a set of entities impacted for each atomic change. Impacted entities are those who are reached during traversing process, and the affecting atomic change is the one that’s at the origin of traversing process. We detail this component later, as it is the most important element of our system.

4.3 DEFINITIONS

In order to better formulate our ideas, we use the following notations in the rest of this paper:

- \( \mathcal{V} \) the set of all vertices,
- \( \mathcal{E} \) the set of all edges, such as: \( \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \),
- \( \mathcal{G} \) the set of all graphs,
- \( \mathcal{P} \) the set of all AspectJ programs,
- \( \mathcal{M} \) the set of all methods,
- \( \mathcal{A} \) the set of all advices.

We define a graph \( G \) by a tuple \((\mathcal{V}, \mathcal{E})\), as \( \mathcal{V} \subseteq \mathcal{V} \) is the set of all nodes of \( G \) and \( \mathcal{E} \subseteq \mathcal{E} \) the set of edges of \( G \). We note \( \nu : \mathcal{G} \to \mathcal{P}(\mathcal{V}) \) the function that associates to each graph all of its vertices \( \mathcal{P}(s) \) represents powerset of \( S \), and \( E : \mathcal{G} \to \mathcal{P}(\mathcal{E}) \) the function that associates to each graph all of its edges. We have for any graph the following property:

\[
\forall G \in \mathcal{G}, \mathcal{E}(e_1, e_2) \in \mathcal{E}(G) : e_1 \in \mathcal{V}(G) \cap e_2 \in \mathcal{V}(G)
\]

In this work we only consider the system dependence graph, so we put:

- \( \mathcal{V}' \) represents the set of entry vertices.
- \( \mathcal{V}' \) represents the set of call vertices.
- \( \mathcal{V}_{st} \) the set of statement vertices.
- \( \mathcal{V}_{ai} \) the set of actual-in vertices.
- \( \mathcal{V}_{ao} \) the set of actual-out vertices.
- \( \mathcal{V}_{fi} \) the set of formal-in vertices.
- \( \mathcal{V}_{fo} \) the set of formal-out vertices.
- \( \mathcal{E}_{d} \) the set of data-dependence edges.

Table 1. Atomic Changes in AspectJ.

<table>
<thead>
<tr>
<th>Aspect part</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>Add an Aspect</td>
<td>Adds a new advice</td>
</tr>
<tr>
<td>DA</td>
<td>Delete an Aspect</td>
<td>Removes an advice</td>
</tr>
<tr>
<td>INF</td>
<td>Introduce a New Field</td>
<td>Adds a new field</td>
</tr>
<tr>
<td>DIF</td>
<td>Delete an Introduced Field</td>
<td>Removes an introduced field</td>
</tr>
<tr>
<td>CIF</td>
<td>Change Introduced Field initial value</td>
<td>Changes the initial value of an introduced field</td>
</tr>
<tr>
<td>IMN</td>
<td>Introduce a New Method</td>
<td>Adds a new method</td>
</tr>
<tr>
<td>DIM</td>
<td>Delete an Introduced Method</td>
<td>Removes an introduced method</td>
</tr>
<tr>
<td>CIM</td>
<td>Change an Introduced Method Body</td>
<td>Changes the body of an introduced method</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object part</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI</td>
<td>Change Field Initial value</td>
<td>Changes the initial value of a field</td>
</tr>
<tr>
<td>AM</td>
<td>Add a Method</td>
<td>Adds a new method</td>
</tr>
<tr>
<td>DM</td>
<td>Delete a Method</td>
<td>Removes an existing method</td>
</tr>
<tr>
<td>CM</td>
<td>Change body of a Method</td>
<td>Changes the body of an existing method</td>
</tr>
</tbody>
</table>

Figure 3. Souyoul Architecture
the set of control-dependence edges.
\( Z_{cl} \) the set of call edges.
\( Z_{pi} \) the set of parameter-in edges.
\( Z_{po} \) the set of parameter-out edges.

We note SDG : \( P \rightarrow G \) the function that associates to each AspectJ program its system dependence graph, so we use SDG(\( P \)) to describe the system dependence graph of the program \( P \).

We note MDG : \( M \cup A \rightarrow G \) the function that associates to each method or advice; their method dependence graph.

To represent our functions with the fewest possible parameters, we assume that element \( e \) remains \( e \) in both versions, and we use the exponent \(^1\) for the first version, and \(^2\) for the second one. For example, if \( e \) is a method, we note \( MDG^1(e) \) its MDG of the first version, and \( MDG^2(e) \) the one of the second version.

### 4.4. THE CHANGE IMPACT ANALYZER

To calculate the impact of an atomic change using two-system dependence graphs (one for each version), we follow the following steps:

a. Calculate the slicing criterion.

b. Calculate the "impact slices" (a kind of forward slices that we define later) according to the criterion resulting from the previous step.

c. Analyze the slices to identify the impacted entities.

Figure 4 represents a detailed view of our change impact analyzer.

![Figure 4. The Change Impact Analyzer in Souyoul.](image)

#### 4.4.1. The Slicing Criterion Calculator

This component takes as input two system dependence graphs (one for each version) and an atomic change. Its role is to calculate the slicing criterion for the atomic change that it takes. In our case, a slicing criterion is a set of SDG’s vertices. These vertices are those by which we start spreading the impact of the change. The calculation of this set varies depending on the type of the atomic change.

We present for example, the slicing criterion of the atomic change \( AAD \) (Add Advice). Since this is one of the simplest cases. The slicing criterion for the addition of the advice \( a \) is given by \( C_{AAD} : A \rightarrow Y_{po} : \)

\[
C_{AAD}(a) = \{ v \in [MDG^2(a)] \cap Y_{po} \}
\]

#### 4.4.2. The Slicer

This component takes as input the two SDGs (one for each version) and a slicing criterion. It calculates the impact slices of the vertices that compose the slice criterion, and marks the reached vertices. The outputs of this component are the two graphs marked by the union of calculated impact slices.

To calculate the impact slice of a vertex, we traverse starting from that vertex; all edges except call and parameter-in. We avoid the call edges because the behavior of a method doesn’t depends on its callers (this is valid also for the advices). And we avoid the parameter-in edges because the behavior of a method does not depends on the data passed to it as input (this is also valid for the advices).

We define the impact slice function by \( IS : V \rightarrow \mathcal{P}(Y) \):

\[
IS(e) = \{ v \in V \mid (e', v) \in (E \cup L_{cl} \cup L_{po}) \} \land (e' = e) \lor (e \in IS(e))
\]

#### 4.4.3. The Slices Analyzer

This component takes as input the two SDG marked in the previous step. Its role is to analyze the two SDG to identify impacted entities. Any not removed method (respectively advice) having a MDG, which contains at least one non actual-out vertex marked, is a method (respectively advice) impacted by the atomic change.

The fact that a method (respectively advice) has only actual-out vertices marked means that the method (respectively advice) does not use the impacted actual-out. Therefore, the atomic change has no effect on its behavior. That is why we consider that a method (respectively advice) with only actual-out vertices marked is not an impacted method (respectively advice).
We define the function-impacted methods that determine the methods/advices impacted for a given slice criterion:

\[ \text{IM}(\{e_1, \ldots, e_n\}) = \left\{ \left. m \in M \cup A : \begin{array}{l} V[\text{MDG}^2(m)] \subseteq V[\text{MDG}^2(m')] \wedge \left( \left( V[A^2(\gamma_m \cup c)] \right) \cap \left( V[A^2(\gamma_m \cup c)] \right) \right) \neq \varnothing \right\} \right\} \]

Obviously, if an artifact is impacted, any artifact that encapsulates it is also affected. For example, if an advice is impacted, then the aspect to which it belongs is considered impacted as well.

5. CONCLUSION

Several approaches have been proposed for the change impact analysis, but little of them treat this problem in aspect oriented programs. Among these approaches we find the approach of Celadon [14, 15] which is based on a call graph to propagate the effect of a change. But this approach gives false positive results. This lack motivated us to propose our approach.

We have suggested throughout this paper, a new approach to change impact analysis in aspect-oriented programming (AspectJ). Our approach consists of the flowing steps:

a. Taking two versions of an AspectJ program.

b. Then we detect the difference between the two versions, and present it in form of atomic changes.

c. Then we generate a system dependence graph for each version.

d. Once we have the atomic changes and the system dependence graphs; we propagate the effects of changes in the graphs using the program slicing.

Spreading the change impact by program slicing minimizes false positives results. Since the program slicing is a fine granularity technique for program analysis.

The prospect scheduled to give continuity to this work is to test Souyoul -the tool implementing our approach- and compare it with other tools of change impact analysis. Note that currently there is no available change impact analysis tool for aspect oriented programs (Celadon is not currently available to the public).

References:


