Hammock-based Identification of Changes in Advice Applications between Aspect-oriented Programs

Marija Katic
Independent Researcher, London, U.K

Keywords: Aspect-oriented Programming, Program Differencing, CFG Comparison, Hammock, Hammock Graphs.

Abstract: In an aspect-oriented program, the cross-cutting functionalities are defined in pieces of advice such that they apply to program-execution points for the core functionalities. Program changes can affect the application of pieces of advice. To that end, a source-code differencing tool, for two versions of an aspect-oriented program, needs to support the identification of changes in pieces of advice at locations of their applications. To alleviate this task, we introduce an extension of the existing differencing technique for object-oriented programs. We implemented a tool AjDiff and used it to evaluate our technique on the two examples of aspect-oriented programs: Tracing and Telecom. We manually verified that our tool can successfully identify changes in pieces of advice at locations of their application.

1 INTRODUCTION

Aspect-oriented programming (AOP) (Kiczales, Lamping, et al., 1997) has been introduced as a way for separating the cross-cutting functionalities from the core program functionalities. With AOP, the program code is divided into the two parts: the base code (BC) for core functionality concerns and the aspect code (AC) for cross-cutting concerns. The interactions between the two parts are established following the concepts of a *joinpoint*, *advice*, and a *pointcut*.

Join-points are program-execution points where the cross-cutting code, which is contained in pieces of advice, is injected (applied) according to the rules specified in pointcuts. For example, method calls represent join-points. Advice is contained within the main unit of the AC, *aspect*. There can be *before*, *after*, and *around* advice executed preceding, succeeding, and surrounding a join-point, respectively. The final executable aspect-oriented (AO) program is created by merging the BC and the AC, which is called weaving. Although the advice can be woven into the another advice, we only consider the weaving of pieces of advice into methods.

AOP works such that it extends the features of another programming paradigm (Kiczales, Lamping, et al., 1997; Kiczales et al., 2001; Coady et al., 2001). The well-established AOP language AspectJ (Kiczales et al., 2001) complements the object-oriented programming paradigm (OOP). In this paper, we base our approach on AspectJ.

Like with traditional programs such as object-oriented (OO) programs, there is a need for automated techniques that support the maintenance and evolution of AO programs as well (Mens and Demeyer, 2008; Przybyłek, 2018). At the heart of many of such techniques, there is the identification of changes between two versions of a program: an original version and a modified version, which means the identification of differences and correspondences between the two versions (Apiwattanapong et al., 2007). Following the identification of changes, program entities are classified as deleted, added, and matched (modified or unchanged). When discussing the identification of changes, we assume such a classification of program entities. The examples of the techniques that use the result of the identification of changes between two versions of a program include change impact analysis (Arnold, 1996; S. Zhang et al., 2008), code review (Barnett et al., 2015), finding patterns of changes (Qian et al., 2008), and dynamic update generation (Katić, 2013).

The identification of changes between two versions of a program can be done with differencing algorithms (Apiwattanapong et al., 2007) such as algorithms proposed by (Fluri et al., 2007; Falleri et al., 2014; Apiwattanapong et al., 2007). For two AO

Katic, M.
Hammock-based Identification of Changes in Advice Applications between Aspect-oriented Programs.
DOI: 10.5220/0007747504420451
In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2019)*, pages 442-451
ISBN: 978-989-758-375-9

programs, in addition to identification of changes between program entities, a differencing algorithm also needs to identify changes in applications of pieces of advice at join-points. This is needed because a change introduced in AO program can be manifested in the advice applications at join-point locations. Consider an example of moving advice from one aspect to another; this can introduce an unwanted change in the order of advice applications at places where multiple pieces of advice apply to the same join-point. Or, consider the example when renamed method changes the semantics of pointcuts thus causing them to incorrectly capture or miss capturing join-points that are related to the renamed method (Koppen and Störzer, 2004). This issue is known as the fragile pointcut problem.

The manual identification of changes in advice applications might be tedious and time-consuming even if the total number of pieces of advice per join-point is small. This is because software systems often contain millions of source code lines and applying advice can be found at many of them. For example, advice such as logging can be applicable to many, if not all the method calls in a program. Changing such a program can affect the application of logging advice at many join-point locations. To this end, the automated support is needed for the identification of changes in pieces of advice per join-points.

The application of the existing differencing approaches for OO programs such as (Apiwattanapong et al., 2007; Fluri et al., 2007; Falleri et al., 2014) to AO programs does not yield the expected results because such proposals do not account for interactions between the BC and the AC. For that reason, researchers extend or complement the existing proposals by focusing on these interactions. Thus, (Koppen and Störzer, 2004; Stoerzer and Graf, 2005), in their approach, which tackles the fragile pointcut problem, focus on the identification of changes in advice applications at join-points. However, they do not deal with matching of program statements, and only match join-points based on the program elements (e.g. methods) that contain or reference join-points. The approach for differentiating AO programs at the statement level, proposed by (Görg and Zhao, 2009), misses to relate changes in advice applications to their corresponding join-points. Furthermore, the work proposed by (Khatchadourian et al., 2017), which deals with the fragile pointcut problem, does not provide a differencing result as it is, but suggests pointcuts that possibly become incorrect after the change of the BC. It works in real-time, thus notifying a developer about possibly incorrect pointcuts (causing incorrect advice applications) as soon as the change has been introduced. Approaches that work in the context of another task such as change impact analysis (S. Zhang et al., 2008) and regression testing (Xu and Rountev, 2007) do not focus on the classification of changes in advice applications.

In summary, existing pure differencing approaches provide separate approaches to the ideas of the identification of changes between program statements and the identification of changes between advice applications at join-points. In this paper, we unify these two existing ideas and propose a novel technique that deals with both of them at the same time, thus providing a single differencing approach for AO programs.

We propose the technique as an extension of the technique from (Apiwattanapong et al., 2007) because their algorithm `CalcDiff` works on control-flow graphs[1] (CFGs) and because CFGs have already proved convenient, as noted by (Xu and Rountev, 2007), for modelling of the complicated semantics of AO programs. In addition, compared to other proposals such as (Fluri et al., 2007; Falleri et al., 2014), `CalcDiff` improves the detection of changes that are specific to OOP. Some of these changes can even be responsible for the changed pointcut semantics. For instance, changing the position of a class in the inheritance tree can cause that pointcuts unexpectedly capture or miss capturing join-points within that class.

It is worth mentioning that (Görg and Zhao, 2009) applied `CalcDiff` to their CFG representation of AO programs. Although the details of the representation are not available in the English language, we concluded that it accounts for the interactions between the BC and the AC.

The focus of our technique is on the identification of changes between advice applications at those join-points that are matched between the two versions. Our work includes the clarification and the alterations of existing extensions of the CFG that account for the interactions between the AC and the BC, and it includes the extending of `CalcDiff` algorithm. The contributions of this paper are as follows:

[1] The CFG of a method m $CFG_m=(N, E, n_s, n_e)$ is a directed graph that represents all possible paths traversed through the method (Aho et al., 2006). Nodes (set N) represent statements, and edges (set E) represent flow of control between statements. There are a single entry node n_s and a single exit node n_e.

- The definition of the alterations of the CFG representation for the interactions between the BC and the AC, which is inspired by the work provided by (Görg and Zhao, 2009).
- The novel definitions of single-entry-single-exit CFG sub-graphs, referred to as *artificial hammocks*, that are used to relate pieces of advice to join-points.
- The extension of `CalcDiff` algorithm to AO programs, called `CalcDiffAO`, which uses the artificial hammocks to identify changes in advice applications for matched join-points.
- The evaluation of the usefulness of our approach for two simple examples of AO programs, which is based on our tool that implements the proposed representation and `CalcDiffAO` for AspectJ programs.

The rest of the paper is structured as follows. The motivating example is given in Section 2. Section 3 brings out the CFG representation, definitions of artificial hammocks and the differencing algorithm. Section 4 presents the AjDiff tool and the evaluation studies. Section 5 gives conclusions and outlines the directions for future work.

2 MOTIVATING EXAMPLE

In this section, we present an example demonstrating how a program change can affect the application of pieces of advice in an undesirable way.

In AspectJ, the order of execution of multiple pieces of advice that apply to the same join-point is inferred not only from the type of the advice (`before`, `after`, `around`) as mentioned in the introduction, but also from the precedence rules (Laddad, 2009). Around advice that has precedence over another advice, surrounds that advice and determines (via the call to `proceed`) whether that advice will be executed or not (Laddad, 2009). It also controls the execution of a join-point that it surrounds via the call to `proceed`.

We extended Telecom example from the AspectJ example suite. For the original version we modified and extended the example. The modified version is created from the original version by changing only the corresponding AC. In Listings 1 and 2, we present only an excerpt of the code that is sufficient for the illustrative purposes.

The BC, which is the same in both versions, is not listed. It includes the code that enables the communication via the telephone call between a caller and a receiver. For such a call to be established, there must be established a connection between the caller and the receiver (method `complete` does this). The connection is modelled with a class `Connection`. We extended Telecom such that the caller can request that the connection is established only if the receiver accepts to pay for the call. If the caller makes such a request, then the call and the connection are considered to be *conditional*.

Listing 1: The AC in the original version.

```
1: aspect Aspect {/*...*/
2: pointcut pcConn (Connection c):
   target(c) && call(void
   Connection.complete());
3: after (Connection c): pcConn(c){
   /*...timing advice...*/}
4: void around (Connection c):pcConn(c)
   {/*...check conn...*/ proceed(c);}
5: }
```

Listing 2: The AC in the modified version.

```
1: aspect Aspect {/*...*/
2: pointcut pcConn (Connection c):
   target(c) && call(void
   Connection.complete());
3: void around (Connection c):pcConn(c)
   {/*... check conn...*/ proceed(c);}
4: after (Connection c): pcConn(c){
   /*...timing advice*/ }
5: after (Connection c): pcConn(c)
   {/*...recording advice...*/ }
6: }
```

We investigate the application of pieces of advice to join-points that refer to invocations of the method `complete`. Listing 1 shows `after` and `around` pieces of advice that apply to such join-points in the original version of Telecom. Listing 2 shows `around` and two `after` pieces of advice that apply to such join-points in the modified version of Telecom.

The `around` advice from Listing 1 wraps all join-points with calls of the method `complete` in order to prevent the execution of that method for conditional connection that must not be established as a consequence of the (reject) response provided by the receiver. The `after` advice starts a timer for measuring the duration of the call once the connection is established. It is placed before the `around` advice within the aspect `Aspect` so that its execution can be controlled by the `around` advice.

In the modified version of Telecom (Listing 2), we added new `after` advice that supports the recording of calls and that applies according to the same pointcut as the timing `after` advice. To

illustrate the failure, we placed that advice below the `around` advice and we moved the timing `after` advice also below the `around` advice. In this way, the execution of two `after` pieces of advice is not controlled by the `around` advice. This is not correct because `after` pieces of advice must not execute unless the connection is established, which is controlled by the `around` advice. This mistake might appear small but it can have adverse and potentially widespread consequences on program behaviour because it affects all the statements with invocations of the method `complete`. The same error might happen if a programmer, during the refactoring process, creates a separate aspect for each advice and forgets to define the precedence order among the aspects.

Our approach can be used to detect changes in the application of timing and recording pieces of advice.

3 APPROACH

`CalcDiffAO` is the differencing algorithm that extends `CalcDiff` such that it works for AO programs (Katić, 2013; Katic and Fertalj, 2013). It accepts two versions of an AO program, the original version *P* and the modified version *P'*, and compares them at the three levels in the following order: the *class and interface levels* (level 1), the *method level* (level 2), and the *node level* (level 3). The three-level comparison originates from `CalcDiff`. `CalcDiffAO` introduces the comparison of aspects at the level 1 and the comparison of pieces of advice at the level 2. The classification of compared program entities is done such that equal entities are classified as matched. Otherwise, they are classified as deleted if they are from *P* or as added if they are from *P'*.

At the **Level 1**, `CalcDiff` compares classes and interfaces based on their fully-qualified names (package name joined with class or interface name). This also works for the comparison of aspects, in which case the fully-qualified name of an aspect consists of the package name joined with the aspect name.

At the **Level 2**, for matched classes, interfaces and aspects, `CalcDiffAO` compares fully-qualified names of methods (fully-qualified class / interface / aspect name joined with method signature) in a similar way as `CalcDiff`. In particular, while `CalcDiffAO` matches only methods where their signature is equal, `CalcDiff` also matches methods with same names if it previously misses to match them based on their signatures. In order for this comparison to hold for pieces of advice from matched aspects, we define the *advice identifier* as a combination of the *aspect name*, the *advice declaration* and the *pointcut specification*.

At the **Level 3**, `CalcDiff` needs to be changed significantly so that changes in the application of pieces of advice can be identified. This is where is the main contribution of our proposal.

The output of `CalcDiff` consists of sets of matched classes, interfaces, and methods, and of the set *N* of matched pairs of CFG nodes along with the status of their comparison (*modified* or *unchanged*). `CalcDiffAO` extends this output with sets of matched aspects and pieces of advice, and extends the set *N* such that for each matched node pair, there are relevant advice-nodes classified in sets as added, deleted, modified or unchanged.

3.1 Comparison at the Level 3

For matched pairs of methods, `CalcDiff` compares their statements. In addition to that, `CalcDiffAO` compares pieces of advice per matched method statements. Statements of matched pieces of advice are also compared, but not pieces of advice applied to them. Further, we only refer to method statements.

To compare pieces of advice per matched statements, `CalcDiff` is changed as follows: (1) Methods are represented *with aspect-oriented control-flow graphs* (AO-CFGs) - CFGs that account for the interactions between the BC and the AC (Katić, 2013); (2) The notion of a single-entry-single-exit CFG sub-graph, which is called hammock, is extended so that aspect-related parts of AO-CFG can be recognized. Such a hammock that marks an aspect part of the AO-CFG is referred to as the artificial hammock; and (3) To support the comparison of artificial (aspect) hammocks, new steps of the algorithm are introduced.

AO-CFG: CFGs for AO program proposed by (Zhao, 2006; Bernardi and Lucca, 2007; Xu and Rountev, 2007) are not suitable to be used in `CalcDiffAO` without their alterations because they provide the inter-procedural representation, while we need the intra-procedural representation (as it has been used in `CalcDiff`). The authors in (Görg and Zhao, 2009) used the intra-procedural representation, but they did not provide enough details about the creation of graph nodes. To overcome this issue, we have defined the extended CFG of a method, which is called AO-CFG, that is

suitable for our idea of extending `CalcDiff`. After defining the AO-CFG, we will clarify the differences between the AO-CFG and the representation used by (Görg and Zhao, 2009).

`CalcDiff` is based on a traditional CFG representation of a method called ECFG (Apiwattanapong et al., 2007). The ECFG is enhanced CFG that models OO features such as dynamic binding and exception handling. For method call statements, in AO-CFG, we apply the modelling of dynamic binding from the ECFG, which facilitates the detection of method-invocation changes emerged because of changes in class hierarchies.

In the process of building AO-CFG, the extensions are done at the CFG nodes that represent join-points. Which nodes represent join-points depends on the type of a join-point. We consider only the method execution and the method call join-point types. Other types such as field get and set join-points can be considered in an analogous way.

Since the method execution join-point encompasses the body of a method, all the CFG nodes for the method represent this join-point. For a statement corresponding to the call of a method, there are at least three nodes in the CFG as defined by ECFG. They all represent the method call join-point. Hereafter, we refer to all CFG nodes for a join-point as a *join-point-node*.

How the CFG is extended is described with the *aspect graph* (AG) and the *around graph* (ARNG). The AG for a join-point *p* and pieces of advice that apply to *p* is a CFG $AG_p = (N_p, E_p, aentry_p, aexit_p)$ that represents the paths of execution for *p* and the corresponding pieces of advice. There are the *aspect-entry-node* $aentry_p$ and the *aspect-exit-node* $aexit_p$ that represent the entry and the exit node of the AG_p respectively. N_p contains the join-point-node, and, for each advice that applies to *p*, the *advice-node*. The advice-node is labelled with the corresponding advice identifier. Edges in E_p represent flow of control between the join-point and the corresponding pieces of advice.

For around advice, apart from the corresponding advice-node, which is also called *around-entry-node*, in AG, there is the *around-exit-node* that denotes the end of the execution of the around advice. This helps to recognize pieces of advice that are surrounded with the around advice. The ARNG is a sub-graph of AG such that it is also AG in which the aspect-entry-node is equal to the around-entry-node and the aspect-exit-node is equal to the around-exit-node.

The AG represents pieces of advice that can be applied according to static pointcuts (evaluated during compile-time) and dynamic pointcuts (evaluate during run-time). Similarly to (Stoerzer and Graf, 2005), for the dynamic pointcuts, we conservatively assume compile-time evaluation.

The AO-CFG for a method *m* is the CFG in which each join-point-node *p* with applying advice is replaced with a corresponding AG. If *p* is the execution join-point, then, after the replacement of *p* with AG_p, new entry and exit nodes for the AO-CFG are created. An edge is created from a newly created entry node to $aentry_p$, and an edge is created from $aexit_p$ to a newly created exit node of the AO-CFG. This is to facilitate the comparison between two execution join-points where applying advice exists for only one of them. The definition of AO-CFG conforms to the formal definition of CFG.

The similarities and differences between the AO-CFG and the CFG used by (Görg and Zhao, 2009) are as follows. The main similarity between the two representations is that in both of them there is a single node that corresponds to each advice. Furthermore, in the representation used by Görg and Zhao, it is not clear if they create *weave and return nodes* (this is how they specify them) for each advice node that is created at a join-point location or if these nodes are created only once per join-point. Another difference is the modelling of the around advice. In particular, they do not create a node that could be considered as an equivalent to the around-exit-node of the AO-CFG. For that reason, in their representation, it is not clear how to recognize pieces of advice that are surrounded with the around advice.

Hammocks: For a CFG *G*, a hammock $H = (N', E', n', e')$ is its sub-graph with the start node n' in *H* and the exit node e' not in *H*, such that all the edges from $(G \backslash H)$ to *H* go to n' and all the edges from *H* to $(G \backslash H)$ go to e' (Ferrante et al., 1987). A hammock is minimal if there is no another hammock with the same start node and with a fewer number of nodes (Apiwattanapong et al., 2007). We assume minimal hammock, unless it is specified differently. Differencing based on hammocks was adapted in `CalcDiff` because the hammock structure appeared to be useful for the identification of changes between two CFGs (Apiwattanapong et al., 2007).

The structure of the AO-CFG with hammocks is built in a way proposed by Apiwattanapong et al. First, we identify hammocks in AO-CFG, then, for each hammock *H* we replace all its nodes with a newly created node called a *hammock node*. The

replacement is done in such a way that all predecessors of the start node of *H* are appropriately connected with the hammock node and the hammock node is appropriately connected with the exit node of *H*. This procedure is repeated on an obtained graph, called a *hammock graph*, until a single hammock node is left. Similarly to `CalcDiff`, `CalcDiffAO` also uses algorithms for the identification of hammocks that are defined by (Laski and Szermer, 1992) and (Ferrante et al., 1987). Further, we focus on special hammocks, called artificial hammocks, because they are our main novelty with respect to hammocks.

Artificial Hammocks: By its definition, a hammock is "unaware" of whether a node that it contains belongs to the BC or to the AC. Therefore, in an environment where we rely on hammocks to facilitate the differencing of two AO-CFGs, the association of advice-nodes to the corresponding join-point-nodes is hampered. Having a graph structure that supports such an association would enable us to extend `CalcDiff` so that it preserves the matching of two versions of the BC while identifying changes in pieces of advice at the corresponding BC statements. To this end, we define the *aspect hammock*, the *join-point hammock*, and the *around hammock*. They are called artificial hammocks because they are defined to conform to the formal hammock definition, but they would not necessarily be identified as hammocks when applying the formal hammock definition. They are not necessarily minimal, and, for a join-point *p*, they are defined as follows:

The **Aspect Hammock** (AH) corresponds to the AG $G_p = (N_p, E_p, aentry_p, aexit_p)$. Its start and exit nodes correspond to $aentry_p$ and to a successor of $aexit_p$ respectively. If *p* is the execution join-point, then AH is called the execution AH (EAH). If AG is ARNG, then the AH is referred to as the **Around Hammock** (ARNH).

The **Join-Point Hammock** (JPH) corresponds to the join-point-node for *p* and it is a sub-hammock of the AH for *p*. Its start node is the node from the join-point-node that dominates any node from the join-point-node. Its exit node is the node (not from the join-point-node) that post-dominates any node from the join-point-node. This means that for a call join-point that is modelled with more than three nodes, the corresponding JPH is not minimal. This facilitates the matching of join-point-nodes, independently of the aspect-related nodes.

We adapted the algorithms from (Laski and Szermer, 1992) and (Ferrante et al., 1987) such that, in the AO-CFG, the artificial hammocks are identified in addition to minimal hammocks. An artificial hammock is collapsed into a single node in the same way as a minimal hammock.

3.1.1 Comparison of Aspect Hammocks

In order to match hammocks from *P* and *P'*, `CalcDiff` uses the algorithm `HmMatch` which is based on the algorithm for finding an isomorphism between two graphs (Laski and Szermer, 1992). For a pair of hammock nodes (*n*, *n'*), `HmMatch` recursively expands them and, while traversing through the resulting graphs in a depth-first search manner, it compares and matches their nodes via the `comp` procedure. `HmMatch` uses the values of the two input parameters *LH* and *S*. *LH* is used to determine the depth of the graph until which the comparison is done. *S* is used as a similarity threshold when `HmMatch` determines the similarity between two hammocks. `HmMatch` returns a set of matched node pairs from the pair (*n*, *n'*).

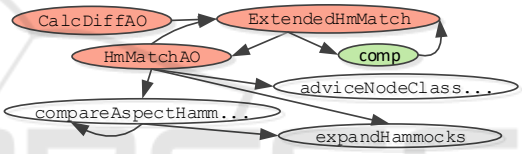

Figure 1: Call graph for the differencing algorithm.

We changed `HmMatch` so that it can identify changes in aspect-related nodes for matched join-point-nodes. We call the changed algorithm `ExtendedHmMatch`. Figure 1 presents our interventions of `CalDiff` and `HmMatch` in form of the call graph.

Our changes of `HmMatch` include: (1) The input of `ExtendedHmMatch` also includes the set *NA* with the information about changes in bodies between pieces of advice matched at the level 2. This is used in the classification of advice-nodes; (2) Before a pair of nodes is compared with `comp` procedure, an additional check for AHs is incorporated so that, if at least one of the two nodes is AH node, they are compared with the algorithm `HmMatchAO` instead; (3) For each matched pair of BC nodes ($n \in P$, $n' \in P'$), the algorithm returns the status of their comparison (modified or unchanged) and the four sets of related advice-nodes. Let *a* and *a'* denote advice-nodes. These sets are:

- $D = \{a \in P \mid a$ applies to n; ($\nexists\ a' \in P'$ such that a' is a counterpart advice-node for a)$\}$
- $A = \{a' \in P' \mid a'$ applies to n'; ($\nexists\ a \in P$ such that a is a counterpart advice-node for a') $\}$

- M ={(a, a') | a applies to $n \wedge a'$ applies to $n' \wedge a$ and a' are classified as modified because of the change in the body or in the execution order for the corresponding pieces of advice}
- U ={(a, a') | a applies to $n \wedge a'$ applies to $n' \wedge a$ and a' are classified as unchanged for the corresponding pieces of advice }

Algorithm 1 illustrates the steps of `HmMatchAO` that, for a pair of nodes (n, n') returns the pair of join-point nodes (jp, jp') and, in sets A, D, M, and U, related advice-nodes. It also returns the comparison result for jp and jp' after comparing them by calling `ExtendedHmMatch`. This result is returned via the variable *status* ("modified" if at least one pair in N is modified, otherwise "unchanged").

Algorithm 1: HmMatchAO.

Algorithm `HmMatchAO`

1: **if** n is not AH and n' is AH **then**
2: **if** n' is EAH **then** jp←super-hammock of n
3: **else** $jp \leftarrow n$ **end if**
4: $A, jp' \leftarrow$ `expandHammocks`(n')
5: **else if** n is AH and n' is not AH **then**
6: **if** n is EAH **then** jp'←super-hammock of n'
7: **else** $jp' \leftarrow n'$ ***end if***
8: $D, jp \leftarrow$`expandHammocks`(n)
9: **else if** n is not EAH and n' is EAH **then**
10: $jp \leftarrow$ super-hammock of n
11: $A, jp' \leftarrow$ `expandHammocks`(n')
12: **else if** n is EAH and n' is not EAH **then**
13: $jp' \leftarrow$super-hammock of n'
14: $D, jp \leftarrow$ `expandHammocks`(n)
15: **else**
16: $A, D, M, U, (jp, jp') \leftarrow$ `compareAspectHammocks`(n, n', NA, false)
17: $A, D, M' \leftarrow$ `adviceNodeClassification`(A, D)
18: $M \leftarrow M \cup M'$
19: **end if**
20: $N \leftarrow$***`ExtendedHmMatch`*** (jp, jp', LH, S, NA)
21: return {(jp, jp'), {A, D, M, U}, *status*}

`HmMatchAO` uses the following procedures. For a hammock node n, `expandHammocks` expands n and returns a join-point-node and a set of corresponding advice-nodes. The procedure `adviceNodeClassification` matches advice-nodes from sets A and D and returns a set of nodes that are matched (M'), while removing corresponding matched nodes from A and D (sets are also returned). For a pair of AH, EAH or ARNH nodes, `compareAspectHammocks` returns a pair (jp, jp') and, in sets A, D, M, and U, related advice-nodes. It does that following these steps:

(1) Hammock nodes n and n' are expanded and prepared such that the aspect-entry-node, the aspect-exit-node and the around-exit-node, which are not relevant for the comparison, are removed from them. In the process of removing a node, predecessors and successors of the node are connected with the rest of the graph so that the control-flow remains preserved.
(2) A pairwise comparison of nodes from the two hammocks is done starting from their start nodes. Thus, advice-nodes with the same order of application with respect to the matching join-point-nodes can be matched.
(3) Nodes are compared based on their types. For a pair of advice-nodes, their advice identifiers are compared. Also, the result of the comparison of their bodies (available from *NA*) is used to conclude if the pair if equal or not. For the same identifiers and bodies, the pair is added to U. Otherwise, only if identifiers are the same, the pair is added to M. For two ARNH nodes, `compareAspectHammocks` is called recursively if the identifiers of their around-entry-nodes are the same. Two nodes with different identifiers or two nodes that are not of the same type are classified into A or D, depending on their origin.
(4) To increase the number of matched advice-nodes, added and deleted ARNHs are compared. Two ARNHs with the same identifiers are compared with `compareAspectHammocks`, and they are removed from A and D. This accounts for matching of ARNHs at different positions with respect to matched join-point-nodes. Matched advice-nodes from such ARNHs are all classified as modified because of the change in the order of execution. To detect this particular origin for advice-nodes, `compareAspectHammocks` uses a Boolean variable that it receives via the input parameters. Its value is set to *true* for advice-node from ARNHs at different positions regarding join-points.
(5) For the ARNH classified as deleted (added), all nested advice-nodes are classified as deleted (added). Advice-nodes and join-point-nodes within the ARNH are detected with `expandHammocks`.

For a pair of call join-points from Section2, Figure 2 partly illustrates the steps of the comparison in advice applications from the point when `HmMatchAO` receives a pair of AHs and continues the comparison at lines 16 and 17. We can see that a pair of AH nodes is expanded, and then a pair of ARNHs is expanded. Finally, into `HmMatchAO`, there are returned a pair of call join-points, a set A with two `after` pieces of advice, a set D with `after` advice, and a set U with a pair of `around` pieces of advice. This classification is

improved by `adviceNodeClassification`, which takes two matching `after` pieces of advice from *A* and *D* and returns them to `HmMatchAO`. This pair of `after` pieces of advice has been classified as modified because of the modification in their application order (unlike in *P'*, in *P* the advice execution is controlled by the `around` advice).

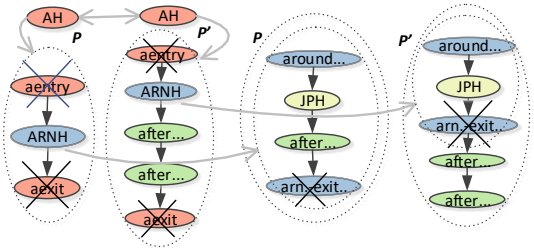

Figure 2: Comparison of advice applications.

Worst-Case Time Complexity: We only analyse the cost for the comparison of advice-nodes introduced by `CalcDiffAO`. Let m and n be the maximum number of advice-nodes per join-point-node in *P* and *P'* respectively. In the worst case, we compare advice-nodes of two AHs, and all of them are different. The two main steps are: graph traversal for a comparison and classification of advice-nodes; comparison of the sets *A* and *D*. If we assume a bit vector representation of a set, then the running time of the first step is O(max(m,n)) and the running time of the second step is O(min(m,n)). This means that the additional comparison of the advice-nodes costs us O(m+n).

4 EVALUATION

In order to evaluate our approach, we developed a tool called AjDiff whose main components are: *compilation*, *graph construction*, *graph storage and data access layer (DAL)*, and *change identification*. For the compilation, we used abc compiler which uses Soot (Vallée-Rai et al., 1999) to generate Jimple intermediate code representation (Vallee-Rai and Hendren, 1998). Jimple is suitable for building AO-CFGs which are built as a part of the graph construction component. The implementation of the advice nesting tree (Xu and Rountev, 2007) from AJANA framework (Xu and Rountev, 2008) is used for the calculation of the advice order of execution in AO-CFGs. AO-CFGs are stored in the graph database created with Neo4J [2] . The *Change Identification* component provides the implementation of `CalcDiffAO`. It is dependent on graph storage via DAL for providing the access to AO-CFGs in the database. Although AjDiff does not support the exception-handling, it works for programs that apply exception-handling principles.

Table 1: Tracing and Telecom - main characteristics.

	Class/Aspect	Method/Advice	Call/Exec.
v1	5/1	32/4	0/19
v2	4/2	32/4	0/19
v3	4/2	32/4	0/16
v1	7/0	32/0	0/0
v2	8//2	55/3	2/0
v3	8/2	46/4	2/0

We run AjDiff on pairs of versions of two programs from the AspectJ (version 1.8.6) example suite: Tracing and Telecom. We used three versions of Tracing, and build configurations basic, billing and timing for the versions v1, v2 and v3 of Telecom respectively. Table 1 presents the main characteristics for versions of Tracing in the first three rows and, in the other rows, it presents the main characteristics for versions of Telecom. There are only execution join-points in Tracing. In Telecom, there are only call join-points.

We run AjDiff on pairs of Tracing (v1, v2), (v1, v3), and (v2, v3) as well as on the same pairs of Telecom. The parameters LH and S were both set to zero. We manually verified the correctness of identified changes. To get an indication of the amount of matched join-points with changes in pieces of advice, we looked for the number of join-points pairs in the four categories: (1) pairs with added pieces of advice only, (2) pairs with deleted pieces of advice only, (3) pairs of the same type (call or execution) with at least one of: deleted advice, added advice, or a pair of matched-modified pieces of advice, and (4) pairs of the same type with matched-unchanged pieces of advice only. We also looked for changes in classes, aspects, pieces of advice and methods. The results are presented in Table 2 where in each of the four categories (Added, Deleted, Modified, Unchanged), the first, the second and the third column refer to the results of AjDiff for the pairs (v1, v2), (v1, v3), and (v2, v3) respectively. The results show that with our approach it is possible to identify numerous join-point locations with changes in advice applications, while the number of pieces of advice in compared programs is small. There are 19 such join-point locations found for (v1,v2) of Tracing while there are only 4 pieces of advice in both compared programs.

[2] https://neo4j.com/

Table 2: Tracing and Telecom - comparison results.

		Added			Deleted		
TRACING	Call	-	-	-	-	-	-
	Execution	-	-	-	-	3	3
	Class	-	-	-	1	1	-
	Aspect	2	2	2	1	1	2
	Advice	4	4	4	4	4	4
	Method	11	11	11	11	11	11
TELECOM	Call	2	2	-	-	-	-
	Execution	-	-	-	-	-	-
	Class	2	2	1	1	1	1
	Aspect	2	2	1	-	-	1
	Advice	3	4	2	-	-	1
	Method	26	17	8	3	3	17
		Modified			Unchanged		
TRACING	Call	-	-	-	-	-	-
	Execution	19	16	16	-	-	-
	Class	3	3	3	1	1	1
	Aspect	-	-	-	-	-	-
	Advice	-	-	-	-	-	-
	Method	19	19	19	2	2	2
TELECOM	Call	-	-	1	-	-	1
	Execution	-	-	-	-	-	-
	Class	5	2	5	1	4	2
	Aspect	-	-	1	-	-	-
	Advice	-	-	-	-	-	2
	Method	5	5	6	24	24	32

It is worth mentioning several interesting observations: (1) A pair of matched classes or aspects is considered modified if there is at least one change between them, which can be a change in advice applications. (2) Even though the numbers reported might be the same among versions, identified changes might be different. For example, for pairs (v1, v2) and (v1, v3) of Telecom, AjDiff identified two call join-points with only added pieces of advice. In (v1, v2), there are two added `after` pieces of advice, and in (v1, v3), there is only one added after advice. (3) For (v1, v3) of Telecom, for the aspect `Timing`, we found the example of AjDiff not supporting the identification of pieces of advice, which was expected. (4) The example of the change in the advice order of execution is found in (v2, v3) of Telecom, for a matched pair of call join-points `drop`.

5 CONCLUSIONS

The current approaches to identifying changes between two versions of an AO program do not work at the statement level, they are not designed for pure program differencing, or they do not provide enough precision with respect to the classification of changes in advice applications at matched join-points. In this paper, we present a differencing algorithm for AO programs that, for two versions of the AO program, gives us matched code parts and for each matched pair of join-points it reports the corresponding added, deleted, modified and unchanged pieces of advice. Although our approach cannot identify changes where advice does not apply but should apply, which is possible with the work in (Khatchadourian et al., 2017), it is the first approach that gives the most precise information about changes in advice applications at matched join-point statements that are matched using one of the state-of-the-art techniques for OO programs. To evaluate our approach, we developed the AjDiff tool and executed two small studies. The results show that the approach could help developers and researchers who need to identify changes in advice applications.

In future, we plan to improve our tool and to fully evaluate our approach on real-world AO programs to confirm our current findings. We are interested in evaluating the usefulness and efficiency of our approach with and without the use of the database for storage of program versions. We also plan to investigate the impact of the value of LH and S on the precision and correctness of our technique.

ACKNOWLEDGEMENTS

I am grateful to Professor Kresimir Fertalj who provided me with the environment to work on this research under the project grant 036-0361983-2022 funded by the Ministry of Science, Education and Sport, Republic of Croatia, as well as for his valuable pieces of advice. For useful discussions, I am grateful to Dr Boris Milasinovic, and PhD students Dubravka Pukljak-Zokovic and Mario Brcic.

REFERENCES

Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D., 2006. Compilers: Principles, Techniques, and Tools. Prentice Hall.

Apiwattanapong, T., Orso, A., Harrold, M.J., 2007. JDiff: A differencing technique and tool for object-oriented programs. Autom. Softw. Eng. 14. https://doi.org/10.1007/s10515-006-0002-0

Arnold, R.S., 1996. Software Change Impact Analysis. IEEE Computer Society Press Los Alamitos, CA, USA.

Barnett, M., Bird, C., Brunet, J., Lahiri, S.K., 2015. Helping Developers Help Themselves: Automatic

Decomposition of Code Review Changesets, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15. IEEE Press, Piscataway, NJ, USA, pp. 134–144.

Bernardi, M.L., Lucca, G.A. di, 2007. An Interprocedural Aspect Control Flow Graph to Support the Maintenance of Aspect Oriented Systems, in: 2007 IEEE International Conference on Software Maintenance. pp. 435–444. https://doi.org/10.1109/ICSM.2007.4362656

Coady, Y., Kiczales, G., Feeley, M., Smolyn, G., 2001. Using aspectC to Improve the Modularity of Path-specific Customization in Operating System Code. SIGSOFT Softw Eng Notes 26, 88–98. https://doi.org/10.1145/503271.503223

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., Montperrus, M., 2014. Fine-grained and Accurate Source Code Differencing, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14. ACM, New York, NY, USA, pp. 313–324. https://doi.org/10.1145/2642937.2642982

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The Program Dependence Graph and Its Use in Optimization. Acm Trans. Program. Lang. Syst. 9, 319–349.

Fluri, B., Wuersch, M., PInzger, M., Gall, H., 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. IEEE Trans Softw Eng 33, 725–743. https://doi.org/10.1109/TSE.2007.70731

Görg, M.T., Zhao, J., 2009. Identifying Semantic Differences in AspectJ Programs, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09. ACM, New York, NY, USA, pp. 25–36. https://doi.org/10.1145/1572272.1572276

Katić, M., 2013. Dynamic Evolution of Aspect Oriented Software (PhD Thesis). University of Zagreb, Zagreb, Croatia.

Katic, M., Fertalj, K., 2013. Identification of Differences between Aspect-Oriented Programs. Seminar Series on Advanced Techniques & Tools for Software Evolution.

Khatchadourian, R., Rashid, A., Masuhara, H., Watanabe, T., 2017. Detecting broken pointcuts using structural commonality and degree of interest. Sci. Comput. Program. https://doi.org/10.1016/j.scico.2017.06.011

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., 2001. An Overview of AspectJ, in: Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01. Springer-Verlag, London, UK, UK, pp. 327–353.

Kiczales, G., Lamping, J., et al., 1997. Aspect-oriented programming. Presented at the ECOOP 97, Springer, pp. 220–242.

Koppen, C., Störzer, M., 2004. PCDiff: Attacking the Fragile Pointcut Problem. Presented at the European Interactive Workshop on Aspects in Software (EIWAS).

Laddad, R., 2009. Aspectj in Action: Enterprise AOP with Spring Applications, second. ed. Manning Publications.

Laski, J., Szermer, W., 1992. Identification of Program Modifications and its Application in Software Maintenance. Presented at the ICSM, IEEE, pp. 282–290. https://doi.org/10.1109/ICSM.1992.242533

Mens, T., Demeyer, S., 2008. Software Evolution, first. ed. Springer.

Przybyłek, A., 2018. An empirical study on the impact of AspectJ on software evolvability. Empir. Softw. Eng. 23, 2018–2050. https://doi.org/10.1007/s10664-017-9580-7

Qian, Y., Zhang, S., Qi, Z., 2008. Mining Change Patterns in AspectJ Software Evolution, in: Computer Science and Software Engineering, 2008 International Conference On. pp. 108–111. https://doi.org/10.1109/CSSE.2008.802

Stoerzer, M., Graf, J., 2005. Using pointcut delta analysis to support evolution of aspect-oriented software, in: 21st IEEE International Conference on Software Maintenance (ICSM'05). pp. 653–656. https://doi.org/10.1109/ICSM.2005.99

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V., 1999. Soot - a Java bytecode optimization framework, in: Press, I. (Ed.), . Presented at the Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research.

Vallee-Rai, R., Hendren, L., 1998. Jimple: Simplifying Java Bytebode for Analyses and Transformations.

Xu, G., Rountev, A., 2008. AJANA: a general framework for source-code-level interprocedural dataflow analysis of AspectJ software. Presented at the 7th International Conference on Aspect-Oriented Software Development, ACM. https://doi.org/10.1145/1353482.1353488

Xu, G., Rountev, A., 2007. Regression Test Selection for AspectJ Software. Presented at the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society. https://doi.org/10.1109/ICSE.2007.72

Zhang, S., Gu, Z., Lin, Y., Zhao, J., 2008. Change impact analysis for AspectJ programs, in: Software Maintenance, 2008. ICSM 2008. IEEE International Conference On. pp. 87–96. https://doi.org/10.1109/ICSM.2008.4658057

Zhao, J., 2006. Control-Flow Analysis and Representation for Aspect-Oriented Programs. Presented at the Sixth International Conference on Quality Software, pp. 38–48. https://doi.org/10.1109/QSIC.2006.20