International Journal of Software Engineering and Its Applications
Vol.8, No.1 (2014), pp.247-266
http://dx.doi.org/10.14257/ijseia.2014.8.1.22

Detecting Change Patterns in Aspect Oriented Software Evolution: Rule-based Repository Analysis

Hanene Cherait [1] and Nora Bounour [2]

Computer Science Department, LISCO research laboratory
Badji Mokhtar–Annaba University, P.O. Box 12, 23000 Annaba, Algeria
[1]hanene_cherait @yahoo.fr, [2]nora_bounour@yahoo.fr

Abstract

Interesting information and Meta-information about software systems can be extracted by analyzing their evolution histories. This information has been proved useful for understanding software evolution, predicting future changes, and performing an efficient change impact analysis. A rich source code repository is a prerequisite for a high quality evolution analysis. Nonetheless, the evolutionary information contained in current versioning systems for Aspect Oriented (AO) software is incomplete and of low quality, hence limiting the scope of AO software evolution analysis. In spite of AO Programming (AOP) characteristics, none of current versioning tools match the need of controlling and storing the AO software evolution, they do not perform well with obliviousness and quantification found in AO code. In this paper, we suggest a rule-based repository for AO software evolution, and specifically for AspectJ programming language. This repository is dedicated to handle the proper characteristics of AO paradigm. In our proposal changes are formulated as rewriting rules and recorded in the repository when they are applied. Then, this last is analyzed to detect change patterns in AspectJ software evolution. We give here, the details of our rule-based repository, as well as the proposed approach for change pattern detection. We present a tool validation and some experimentation to prove the feasibility and the efficiency of our proposals.

Keywords: *Aspect oriented programming, software evolution, change-based versioning systems, graph rewriting, evolution analysis, change pattern detection*

1. Introduction

To understand why software systems become less maintainable when they are changed continuously and to predict their future changes; we have to investigate their version repositories. The research field of this investigation is known as software evolution analysis. It is the retrospective analysis of the evolution, i.e. history, of a software system [16]. This field analyzes and cross-links the rich data available in software repositories to uncover interesting and actionable information about the software evolution and its future development. Analyzing evolution history can help to identify necessary changes, understand the impact of changes, and provide a facility to track the changes and to deduce logical relations between changed entities.

We focus in this paper on the AO software evolution analysis, and specifically the AspectJ program evolution. Since AO software systems are becomes more and more popular, they will be the legacy software of the future. The past decade has seen the increased use of Aspect Oriented Software Development (AOSD) techniques [14] as a means to modularize crosscutting concerns in software systems, the —Major Industrial

ISSN: 1738-9984 IJSEIA

Projects Using AOSD— highlights notable applications of AOSD, of which the most prominent is the IBM WebSphere Application Server [9]. One of the main challenges of AOP lies in the evolution of the software, so techniques and approaches are essential to analyze the evolution of such systems; in order to study and predict its development. Since, large amount of techniques is presented in the literature, to analyze the evolution of different programming paradigms (*e.g.*, procedural, object oriented *etc.*,), seldom effort has been made for AO paradigm.

AOP [23] is a technique for modularizing crosscutting concerns. AspectJ [32] is a well-established AOP language. It is the original and still the best implementation of AOP. AspectJ provides a new kind of modules, called aspects that allow one to modularize the implementation of crosscutting concerns which would otherwise be spread across various modules. This is done in terms of *join points*, *pointcuts*, *advices*, and *introductions*. They define precisely how behavioral and structural crosscutting has to take place. Join points represent well-defined points in the execution of a program, such as method calls, object field accesses and so on. After we identify join points useful for a crosscutting functionality, we need to select them using the pointcut construct. Pointcut is a construct that picks out a set of join points based on given criteria, such as method names and so on. AspectJ defines several primitive pointcut designators that can identify all types of join points. Advice defines additional code to be executed whenever a join point selected by a particular pointcut is reached. An advice can execute before, after, or around the join point. Finally, introductions are used to crosscut the static type structure of classes. They can be used by an aspect to add new fields, constructors, or methods (even with bodies) into given interfaces or classes.

AOP is characterized by obliviousness and quantification. Obliviousness states that one cannot know whether the aspect code will execute by examining the body of the base code [15] *i.e.*, the system code should be unaware to any aspects. Since the quantification is the idea that one can write an aspect that can affect arbitrarily many non-local places in a program [29]. These characteristics make AO software versioning a serious problem, current versioning systems unable to handle the crosscutting nature of AOP. Consequently, their repositories are not a good source of information for an efficient AO software evolution analysis.

In this paper, we suggest a rule-based repository for AspectJ software to store the maximal amount of information about its evolution, taking into account the proper characteristics of AOP. In our proposal, we treat change as a first class entity. In contrast to the file-based nature of classic versioning systems, we believe that change-based principle can present the complete view of AO software evolution i.e. “the fundamental unit of software evolution is the source code change, all other information is maintained to help understand, rationalize, and manage source code changes” [21].

Practically, we use the program representation presented in our previous work [7], where, the AspectJ program is converted to an attributed colored graph. And, changes are formulated as rewriting rules on the proposed program graph. Every applied rewrite rule is stored —directly— in our proposed rule-based repository. In this last, every version of the software (graph) is the set of rewrite rule sequences, where, every rewrite rule sequence presents a specific change request.

Besides, in order to analyze our proposed repository we suggest a change pattern detection approach to identify change patterns in AspectJ program evolution. So, the rule-based repository is investigated (Mined) to detect rule patterns using the Apriori algorithm [1]. Since these rules are the formulation of source code changes, we believe that our approach allows detecting change patterns in AspectJ source code. These patterns can be used to understand AO software evolution, predict future changes,

identifying potential faults, detecting new crosscutting concerns and develop new refactoring algorithms.

The rest of the paper is organized as follows. The next section gives the different ripple effects caused by AO software evolution. Section 3 proves that the information contained in current versioning repositories cannot reflect the AO software evolution. Section 4 gives the details of our proposed rule-based repository for AspectJ software systems. A change pattern detection technique is presented in Section 5. The implementation of our repository as well as the change pattern detection approach is given in Section 6. Section 7 shows the experimentation of our proposal. We pass briefly on the related work in Section 8. And we conclude our discussion in Section 9.

2. Ripple Effects of AO Software Evolution

Along with its advantages, AOP has some potential pitfalls that we must be aware of in evolution. Given, that AOP has set out to modularize crosscutting concerns, but by its mechanics breaks modularity [29]. In the AO software systems, researchers uncovered significant evidence of ripple effects, whereby changes propagated to seemingly unrelated modules. This was caused by interdependencies, created by pointcuts and inter-type declarations, between the base code and aspects. The improved separation of concerns within the AO versions makes the changes less obvious as unexpected modules were affected [27].

Previous research has mainly focused on defining the different challenges in evolving AOP software [2, 3, 29]. For instance, previous research has indicated that the use of certain AOP mechanisms can violate module encapsulation [2] and even introduce new types of faults [3]. In particular, some researchers claim that these faults are likely to be amplified in the presence of evolutionary changes [22]. For example, Pointcuts appear to be a double-edged sword: while they enable certain changes to be absorbed and thereby increase a design's stability, they are also the source of ripple effects that reduce stability [27].

Others have proved with empirical evidence the AOP evolution problems that occur in practice. Their analysis confirms that the lack of awareness between base and aspectual modules (obliviousness) tends to lead to incorrect implementations. Ferrari *et al.*, [13], for example, examined how obliviousness influences the presence of faults in evolving AO programs. They found that obliviousness facilitates the emergence of faults under software evolution conditions. They showed that 40% of reported faults were due to the lack of awareness among base code and aspects. And they indicated that the AOP mechanisms present similar fault-proneness when we consider both the overall system and concern-specific implementations. The results revealed the negative impact of obliviousness on the fault-proneness of programs implemented with AspectJ.

To resume up, -thanks to obliviousness- logical dependencies exist in AO software which makes its evolution more and more difficult *i.e.*, for example, change in a specific class may require a change in other classes or aspects, although; there exists no traditional dependencies (*e.g.*, data and control flow) between these AO software entities. Hidden (logical) dependencies exist in any programming paradigm, but according to the ripple effects of AO software evolution presented above, the existence of such dependencies in AO software is voluminous and ramous.

We believe that the analysis of a rich AO software evolution repository can give a more clear view of its dependencies. This clarity helps to avoid the effects of

obliviousness and quantification in AO software evolution, tasks like change impact analysis and change propagation will be more easy and efficient.

3. AO Software Evolution versus Current Versioning Repositories

Hence, AOP, by preventing code tangling and scattering, improves code quality in one area, and at the same time, by introducing quantification and obliviousness [15], makes its versioning more difficult. Version control in AOP development is more complex than in the traditional software.

The obliviousness property of AOP implies that the developers of core functionality need not be aware of, anticipate or design code to be advised by aspects [15]. Since, the body of an advice is much like a method body—it encapsulates the logic to be executed upon reaching a join point. In contrast to the methods of traditional object-oriented languages, advices are not called explicitly. Instead, the execution of an advice is automatically "triggered" when the control flow reaches the join point that is designated. Consequently, the program modules, in which the events in their control-flow are designated, are also oblivious to the corresponding advices. This restricts the evolvability of the AO software and makes its versioning more difficult.

In spite of AOP characteristics, CVS, Subversio, *etc.*, none of these tools match the need of controlling the AO software evolution. They were never fully adapted to AOP paradigm *i.e.*, versioning systems do not perform well with obliviousness and quantification found in AO code. When classes are oblivious to aspects, so, the crosscutting effect of aspects is not tracked by the versioning system [20].

Most current versioning systems are file-based, rather than entity-based [6]. They manage revisions of programs as text documents organized in files, so, it is not possible to present and track the effects of changes in the base code or the aspects i.e. versioning systems are associated with the storing and retrieving of unwoven files and are ignorant of any weaving information (transversal dependencies). However, AOP by nature defies this principle. First, concerns crosscut the file structure. Second, obliviousness leaves certain crosscutting effects undetected in the (textual) display of files and changes [20]. To resume up, current versioning systems does not manage, store, or display the crosscutting information. Thus, their repositories are not complete enough for an efficient AO software evolution analysis.

We believe that for an efficient AO software evolution analysis, the logical elements in a software system such as Class, Aspect, and Method… should be units of version control. This can help to follow the evolution of every entity in the software, and consequently, preserving the dependencies between the AO software entities independently of the files they belong to.

To achieve this goal, we adopt an approach to store changes on the AO source code —when they occur— in a rule-based repository, where the change is treated as a first-class entity. This repository can be a fundamental source of information for AO software evolution analysis, and will open new ways for both developers and researchers to better understand and explore the AO software evolution. The details of our proposal are presented in the next section.

4. Rule-based Repository for AspectJ Programs

4.1. Overview of our Approach

A rich evolution repository can be the subject of an interesting AO software evolution analysis *e.g.*, mining change patterns or discovering logical coupling between AO software entities. However, the research field of AOP software versioning remains very limited which lead to the absence of a suitable evolution repository (*e.g.*, do not record changes of the transversal dependencies in AO software). Concerned with these issues, we propose a rule-based repository for AO source code evolution; where change is treated as a first class entity.

In our approach, we created a software repository designed to store a maximal amount of information about evolving AspectJ software. In particular, we do not use a versioning system, but built from the ground up a rule-based software repository. In contrast to current versioning systems, changes to the software system are stored directly in the repository. So, we do not view the history of an AspectJ software system as a sequence of versions (versions of files), but as the sum of changes which brought the system to its actual state. The typical realization of a software change is a modification to the source code, so, a new version is created when a source code change occurs.

Figure 1 depicts the overview of our approach, which can be divided in three main steps: (1) as presented in Figure 1.a, the evolved AspectJ source code is considered as an attributed colored graph [7], and the changes to the software are formalized as rewriting rules that transform the graph G to a graph G'; in order to achieve the evolution requests; (2) the software maintainer modifies the colored graph of the AspectJ source code by applying sequences of rewrite rules in a certain order (Figure 1.b); (3) the colored graph is imported to the repository, and the software version is checked-in by storing rewrite-rule sequences applied by the maintainer. So, any version can be checked-out just by applying the related rule-sequences on the evolved AspectJ colored graph (Figure 1.c).

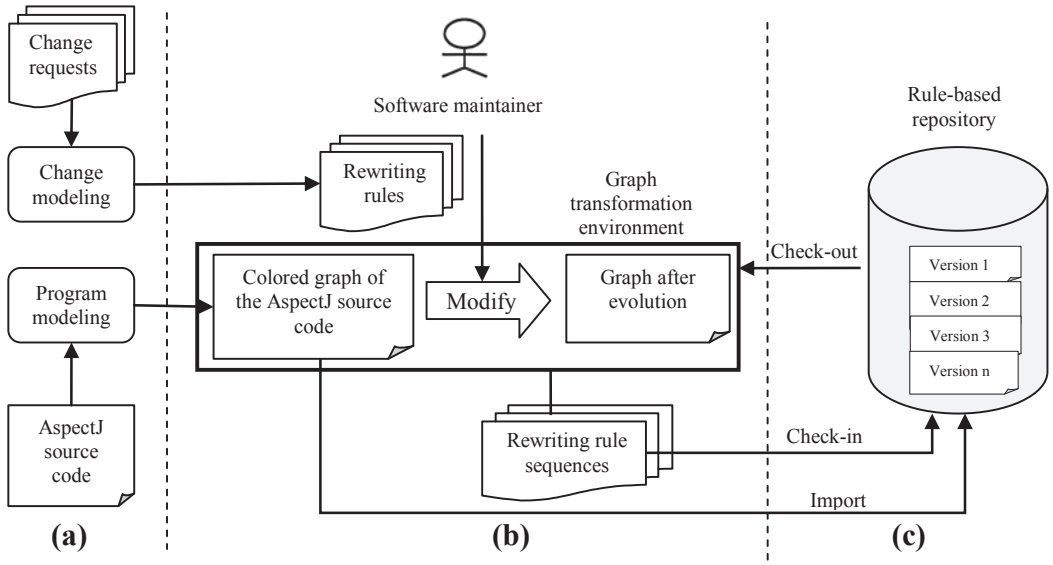

Figure 1. Overview of our Approach

The use of graph rewriting help to well track and control changes in AO software evolution. We store in the repository the complete change *i.e.*, we do not store only the change in the base code (or in the aspects) independently of its effects on the software aspects (or the base code). Rewriting rules give the complete view of the changed entities and their dependencies. For example, if we delete a method which is crosscuted with a particular pointcut, the edge between this method and the pointcut will be deleted too *i.e.*, a pointcut should not capture a deleted method. Here, we can say that our approach can reduce the negative effects of obliviousness in AO software evolution.

4.2. Program Representation

In our approach, the evolved AspectJ source code is represented as an attributed colored graph. The program graph is generated directly from the AspectJ source code. We use therefore; a type graph [10] that plays the role of a Meta-model. A graph G is called typed graph or instance graph, if there exist a distinguished graph TG, called type graph, and a graph morphism $type_G: G \rightarrow TG$, called typing graph morphism.

The AspectJ type graph, shown in Figure 2, specifies how to create well-formed colored graph of AspectJ software. It represents the different entities of the AspectJ program and their dependencies. Any well-formed AspectJ source code can be represented as a graph that conforms to this type graph. This Type graph guarantees the consistency of the graph to every transformation, which specifies what it means for a model to be valid. More details about this representation can be found in our previous work [7].

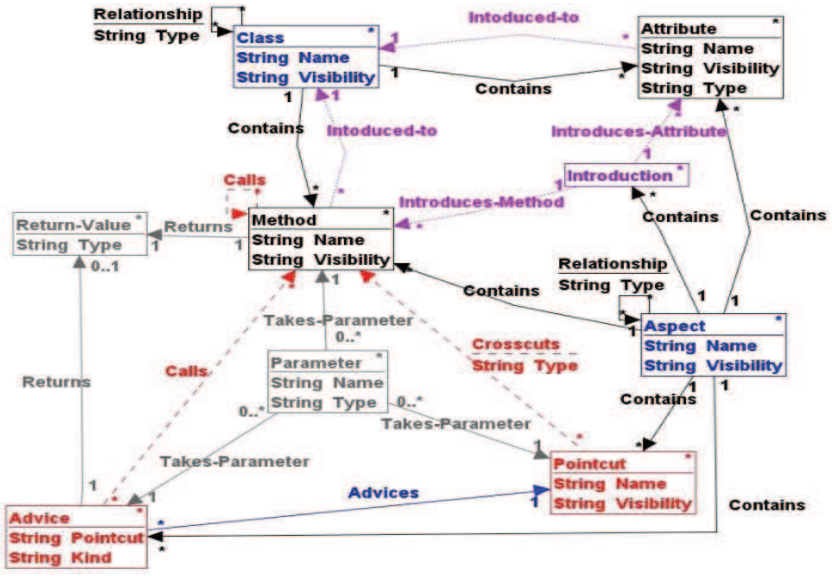

Figure 2. Type Graph of the AspectJ Program [7]

4.3. Change Representation

We represent changes to the program as explicit rewriting rules to its colored graph [7]. A graph rewrite rule [12] consists of a tuple $L \rightarrow R$, whereas L the Left Hand Side (LHS) of the rule is called pattern graph and R the Right Hand Side (RHS) of the rule is the replacement graph. Rules are compared with an input graph called host graph. If a

matching is found between the LHS of a rule and a sub-graph in the host graph, then the rule can be applied and the matching sub-graph of the host graph is replaced by the RHS of the rule. Furthermore, rules may also have conditions (*e.g.*, Negative Application Conditions "NACs") that must be satisfied in order for the rule to be applied, as well as actions to be performed when the rule is executed. A graph rewriting system iteratively applies matching rules in the grammar to the host graph, until no more rules are applicable.

For example, Figure 3 depicts a rewriting rule which create a new public Aspect "A". The NAC presented in the left side of this figure, is used here to avoid the existence of other aspect with the same name.

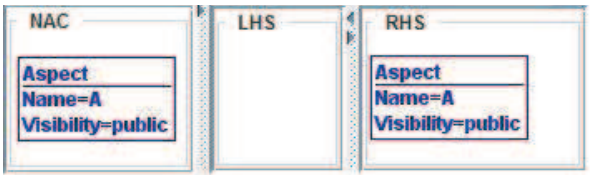

Figure 3. Create a public Aspect "A"

When a rewrite rule is applied takes as input a program state and returns an altered program state. Since each state is an attributed colored graph, rewriting rules are graph operations. The basic rewriting rules are the following:

- *Addition rule*: add a new node or edge to the program graph;
- *Deletion rule*: deletes an existing node and all its dependencies. Or the deletion of just an edge;
- *Modification rule*: modify the proprieties of a node or an edge of the graph.

The combination of several basic rules will be able to give birth to other rewriting rules, or to rewriting rule sequences. A rewrite rule sequence is a set of rewrite rules applied in a certain order to achieve a specific change request.

4.4. The rule-based Repository

Our proposed rule-based repository contains incremental changes to the AO system under study. The sequence of the rewrite rules that a developer is performing is acquired in real-time using the graph transformation environment, and stored in the repository. Figure 4 shows the overview of our proposed repository. Instead of recording the entire changed graph as a version, we only records the rewriting rule sequences applied on this graph. So a Version is a group of rewriting rule sequences applied to the AspectJ graph formulating a given evolution requests.

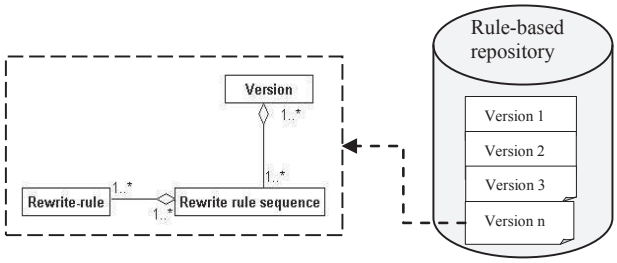

Figure 4. Rule-based Repository

We can reproduce every version of the system by the application of the associated rewrite rule sequences stored in the repository as part of that version.

Changes are stored in a formal format as rewriting rules, which makes the repository more rich and reliable. In contrast to the text format of the change, the rewrite rule is more meaningful, because it contains the full information about the change: pre-condition, post-condition, conditions, action *etc*. This full information facilitates the comprehension of the change, and thereafter storing change in this format makes the repository more accurate for a high quality evolution analysis.

Our repository is change based rather than file-based. So, it breaks the walls between software files and stores the change in its natural format *i.e.*, changes of software entities and their dependencies rather than just changes in the lines of code.

4.5. Discussion

As presented above, every rewrite rule is self explanatory, it contains as much information as possible to formulate the change and control its application. We believe that this format can help to handle the crosscutting nature of AOP. Representing and storing change as rewriting rules can make the AO software dependencies more visible in the repository *i.e.*, the obliviousness effects in the AO source code can be stored explicitly.

For example, if we delete a pointcut, we have to delete their dependencies too; the rewrite rule that formulates this change is depicted in Figure 5. Here we can see that the crosscutting dependency between the pointcut P and the Method M is deleted too. Figure 5 proved again that three entities that belong to different files (method M belongs to the file of the class C, the aspect A and the pointcut P belong to the file of the aspect A) are presented and stored as parts of a single change which is not possible in traditional versioning repositories.

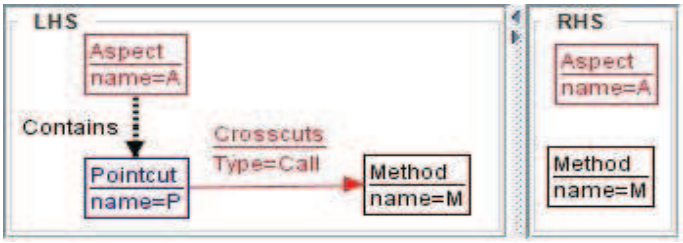

Figure 5. Delete a Pointcut P

5. Our Approach for Change Pattern Detection

Understanding how programs evolve or how they continue to change is a key requirement before undertaking any task in software engineering or software evolution. Extracting change-patterns is important during evolution and maintenance because they provide guidance to maintainers to carry out complete and consistent modifications [5]. We present in this section a change pattern detection approach for AspectJ source code. We define the change patterns to answer the question: given an AspectJ software system and a specific change performed, what others changes must be applied to the system to stay coherent?

5.1. Change Extraction

Most of the change pattern detection approaches use sophisticated tools and techniques to analyze the version repository. These techniques try to extract a suitable representation of changes to be the input for a specific Data Mining algorithm in order to detect change patterns. This is performed by the differentiation between the successive versions stored in the repository *i.e.*, version differencing [25].

The main two steps of this process are: the identification of atomic change sets and grouping these last to transactions. The problem of finding all atomic changes and next the different transactions is not trivial because the performance can be exponential with respect to the number of versions (evolution repository). Thereby, it requires a non-trivial effort; it is an expensive task in term of performance and space memory. It makes up approximately 58 % of run time [31]. Researchers are more interested in gaining convenient access to the extracted data in an easy to process format [17]. So, avoiding this step is very interesting to better enhance the change pattern detection (evolution analysis).

In our approach, changes are stored in the repository while they occur, raising change to a first class concept. There is no need for differencing since the changes are recorded and stored, and thus do not need to be derived later on. Change recording is, in general, more precise and potentially enables to gather more information than version differencing. In contrast to version differencing, recorded change sequences include all intermediate changes. Besides, version differencing does not comprise an order of applied changes, which is, however, usually the case with recorded changes.

So, using our rule-based repository version differencing which is the very costly and difficult task in evolution analysis is not needed and omitted.

Table 1 gives the concepts used in any change pattern detection technique, for traditional approaches. And, it explains the presentation of these concepts in our context. These concepts are more explained in the next sub-sections.

Table 1. Our Approach versus Traditional Approaches

Concept	Traditional approaches	Our approach
Repository	Versions of source code files	Versions of rewrite rule sequences
Changes	Changes in the lines of the source code.	Changes in software entities and their dependencies (Rewrite rules)
Atomic changes	Addition, deletion, modification of source code elements.	Creation, deletion of graph elements (nodes/edges).
Transaction	The set of atomic changes for a specific change request.	The rewrite rule sequence formulated a specific change request.
Change pattern	Atomic changes that happen frequently among the atomic change transaction.	Graph operations (element creation/deletion) that are duplicated enough among the rewrite rule sequences.

5.2. Atomic Change Set

In our proposal we represent change as rewrite rule(s). According to the definition of a rewriting rule, any rule can be easily broken up into a set of creation and/or deletion of source code (graph) elements. Consequently, every rule consists of atomic operations

i.e., creation or deletion of elements (nodes) or dependencies (edges). For example, if we describe the rewrite rule in Figure 6, we distinguish the following atomic changes: deletion of the dependency between A and B, deletion of the node B, creation of node F, creation of node E, creation of a dependency between F and E.

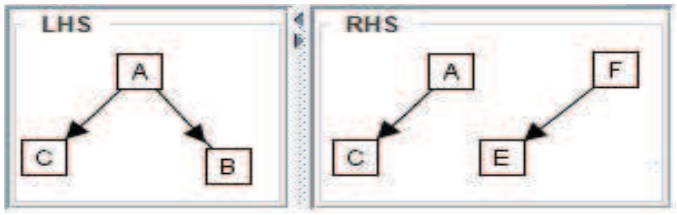

Figure 6. Exemple of a Rewrite Rule

Therefore, we do not have to analyze the rule-based repository to generate the atomic changes (operations) as in traditional techniques. We can define the atomic change as the creation/deletion of any element of our graph (source code). In our repository, every single rewrite rule is recorded directly when it is applied. So, we do not need to use an external tool (*e.g.*, diff) to compare the different versions of a program to detect such rules (changes). The different atomic rewrite rules in our proposal are shown in Table 2.

Table 2. Atomic Rewrite Rules

	The create atomic rewrite rules		The delete atomic rewrite rules	
	Abbreviation	**Atomic rewrite rule**	**Abbreviation**	**Atomic rewrite rule**
Nodes	CC	Create a Class	DC	Delete a Class
	CA	Create an Attribute	DA	Delete an Attribute
	CM	Create a Method	DM	Delete a Method
	CP	Create a Parameter	DP	Delete a Parameter
	CR	Create a Return value	DR	Delete a Return value
	CAS	Create an ASpect	DAS	Delete an ASpect
	CPO	Create a POintcut	DPO	Delete a POintcut
	CAD	Create an ADvice	DAD	Delete an ADvice
	CI	Create an Introduction	DI	Delete an Introduction
Edges	CECA	Create Edge CAlls	DECA	Delete Edge CAlls
	CEIA	Create Edge Introduces Attribute	DEIA	Delete Edge Introduces Attribute
	CEIM	Create Edge Introduces Method	DEIM	Delete Edge Introduces Method
	CECR	Create Edge CRosscuts	DECR	Delete Edge CRosscuts

5.3. Atomic Change Transactions

An atomic change transaction includes prerequisites for a specific change *i.e.*, it is a set of atomic changes for a specific change request.

In our rule-based repository, every version of the program is recorded in the repository as a set of rewriting rule sequences (rewrite rules recorded in a certain order). Every rewrite rule sequence formulates a specific change request i.e. in our context, rule sequences present change transactions. So, we do not need to pre-process the repository to generate such transactions.

On the other hand, every rule sequence is an ordered list of rules. Thus, we already have the semantic dependencies between rules (changes). A rule sequence contains the set of rules for a specific change. That is, rules in a rule sequence are always applied together. As a result, to detect the change-patterns in our approach, we have *just* to analyze the different rule sequences in our rule-based repository.

5.4. Detecting Change Patterns

In this stage, we analyze the rule-based repository in order to identify change patterns. We define change-patterns as common and recurring modifications of software systems in time, during the evolution of such systems. So, we extract atomic sets (atomic rewrite rules) that happen frequently enough among the rule sequences. In our context, such sets, called rule-patterns or change patterns, refer to atomic changes (creation/deletion) that occur (always) together.

We use the traditional Apriori algorithm [1] to detect change patterns in our rule-based repository. Let $R = \{r_1, r_2, \dots, r_m\}$ be a set of atomic rules i.e. creation or deletion of graph elements or dependencies (Table 2), and $X \subseteq R$ a rule-set. We define database (repository) D as a set of rule-sequences: $D = \{s_1, s_2, \dots, s_n\}$, where $s_i = \{s_{i1}, s_{i2}, \dots, s_{ik}\}$ and $s_{ij} \in R$. Also, let $s(X)$ be the set of rule-sequences that contain rule-set X, formally $s(X) = \{Y \in D | Y \supseteq X\}$. Finally, the support of a rule-set X is the fraction of rule-sequences in the database that contain X: $support(X) = \frac{|s(X)|}{|D|}$. Then X is called a frequent rule-set when its support is higher than a given minimum support: $support(X) \geq minsupport$.

In other word, the strength of the pattern $\{r_1, \dots, r_n\}$, where each r_i is an atomic rewrite rule (change), is measured by support which is the number (or percentage) of rule sequences containing $r_1, \dots, r_n$. A frequent pattern describes a set of atomic rules that have support greater than a predetermined threshold called min_support.

6. Validation

6.1. The Repository

The overview of our validation is depicted in Figure 7, which can be resumed in the following parts:

(1) Convertor Tool: to represent the AspectJ source code as an attributed colored graph, we have implemented a convertor tool. This last convert the AspectJ source code to an attributed colored graph in GXL (Graph Exchange Language) [34] format. This graph is imported in the AGG (Attributed Graph Grammar) tool [28] to perform the necessary transformations. For more implementation details, please refer to [7].

(2) Change requests: every change request is formulated as a rewrite rule-sequence *i.e.*, set of rewrite rules applied in a certain order. Then, we use AGG to apply these rules on the attributed colored graph. We can also formulate properties, constraints; analyze the graph…*etc*.

(3) Rule-based Repository: we record every rewrite rule-sequence in the repository when it is applied. Our repository can be defined practically, as a set of GXL documents. Every GXL document presents a version of the program (graph). This is the set of rewrite rule sequences applied in an evolution session. Figure 8 shows the structure of a version. The structure of a rule sequence is depicted in the right hand side of Figure 8; it is constituted of rewrite rules.

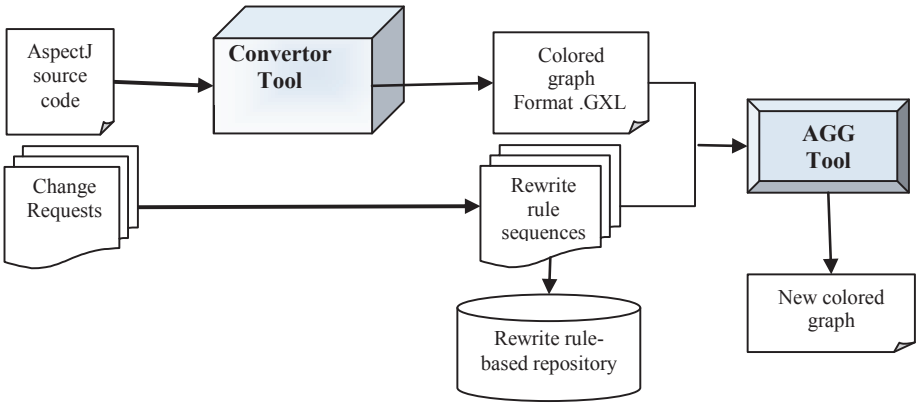

Figure 7. Repository Validation

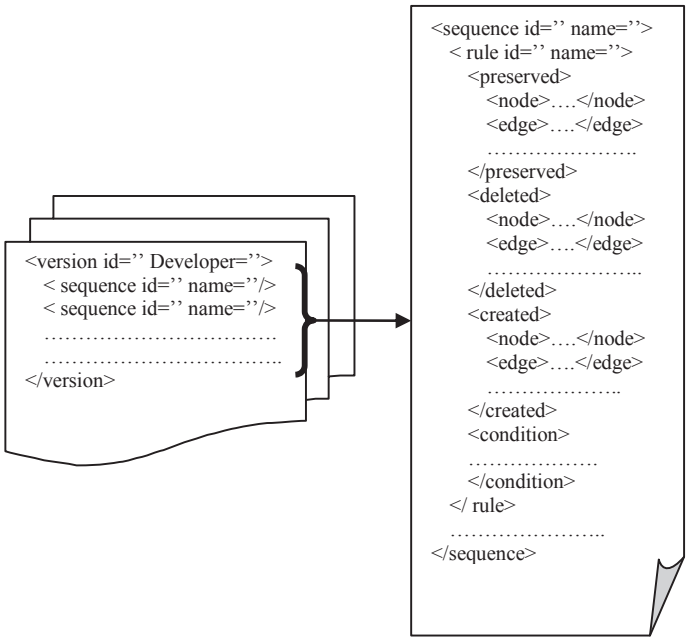

Figure 8. Structure of a Version

Every rule is the combination of preserved, deleted, created and condition, they describe the elements which must be preserved, deleted, and the conditions of the rule respectively.

Note: Graph elements (nodes/edges), rewrite rules and rewrite rule sequences hold unique identifier, in order to keep their identities in the repository. This help in change tracking and repository querying.

6.2. Change Pattern Detection

Since, our repository is a set of GXL documents, the problem of detecting rewrite rule patterns from the rule-based repository is converted to extracting patterns from GXL documents. Every GXL document is an XML (*eXtended Markup Language*) [30]

document. So, we use the XQuery implementation of the Apriori algorithm proposed by Wan *et al.*, [33] to extract such patterns. They propose a set of functions written only in XQuery which implement together the Apriori algorithm. In order to create an appropriate XML document to be the input of the XQuery Apriori algorithm; we follow the pipeline in Figure 9. This last includes 3 steps:

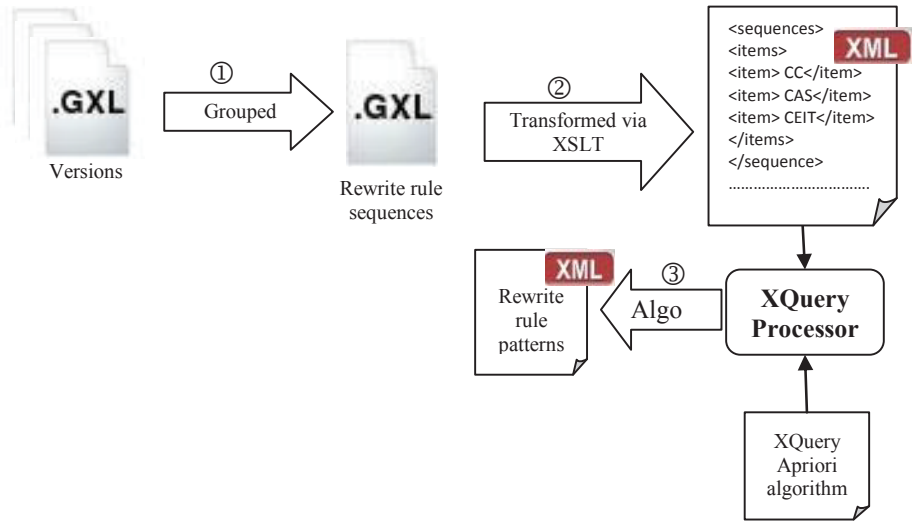

Figure 9. Detecting Rewrite Rule Patterns

Step 1: as depicted in the above sub-section, our rule-based repository is a set of GXL documents (versions). Every GXL document is a set of rule sequences. We regroup all the rule sequences in every version into a single GXL document. This document represents the transactions that must be mined by the change pattern detection algorithm.

Step 2: via the power of XSLT (*XML Stylesheet Language Transformation*) [8], we transform the rule-sequences document produced in the first step to a simple XML document. This last must be as simple as possible, in order to detect rule patterns efficiently. We convert every atomic rewrite rule to a simple tag with the abbreviation of the atomic rewrite rule as value (Table 2). For example, the rewrite rule formulated the creation of a node "Aspect" is represented in GXL format as follows:

```
<created>
 <node id="I173">
   <type name="Aspect"/>
   <attr name="Name">
     <string>A</string>
   </attr>
 </node>
</created>
```

This fragment is converted to the following simple tag :

<item>CAS</item>

Step 3: the XML document produced in the above step is queried with an XQuery processor; according to the XQuery Apriori algorithm (we must fix a specific min support). The output of this step is an XML document representing the rewrite rule patterns in our rule-based repository. These rules represent changes to the AspectJ

source code. Consequently, our approach allows detecting change patterns in AspectJ source code.

7. Experimentation

To validate our claims we need first to gather some data supporting our proposal. Since no change-based versioning system dedicated to AOP is proposed before. Besides, no system has been recording changes in rewrite rule format; we cannot rely on pre-existing software repositories as data sources. So, we choose two AspectJ programs for the experiment: Figure Editor and Tracing. Then, we applied different evolution scenarios to these programs for generating different versions. As a result we have built 5 versions of the Figure Editor program and 3 versions of the Tracing program. Information about the number of Line of Code (LOC), versions and rule sequences for these programs are shown in Table 3.

Table 3. Subject Programs

Programs	LOC	#version	#rule sequence
Figure Editor	393	5	25
Tracing	1059	3	5

The number of the different atomic rewrite rules in every version of the Figure Editor and Tracing programs are shown in Figure 10 and 11 respectively. We predefine min_support threshold to 20% for the both programs. After the application of our rule pattern detection approach, Table 4 summarizes the generated rewrite rule patterns.

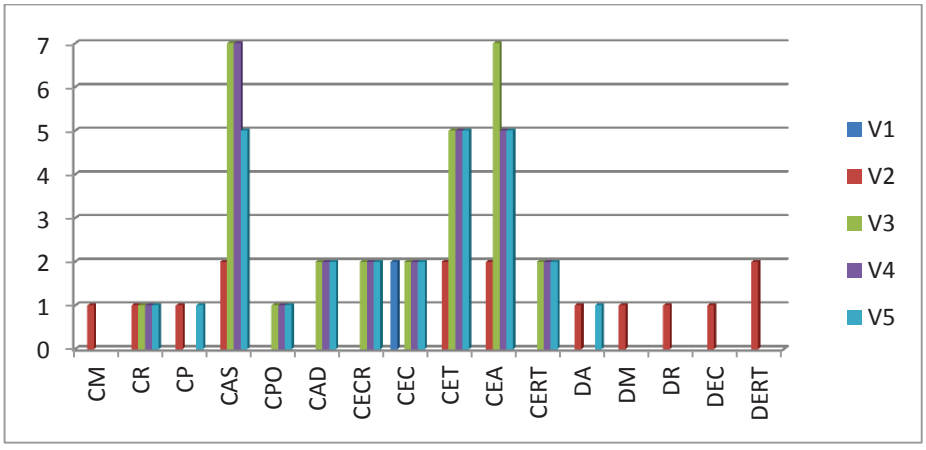

Figure 10. Atomic Rules in Figure Editor Program

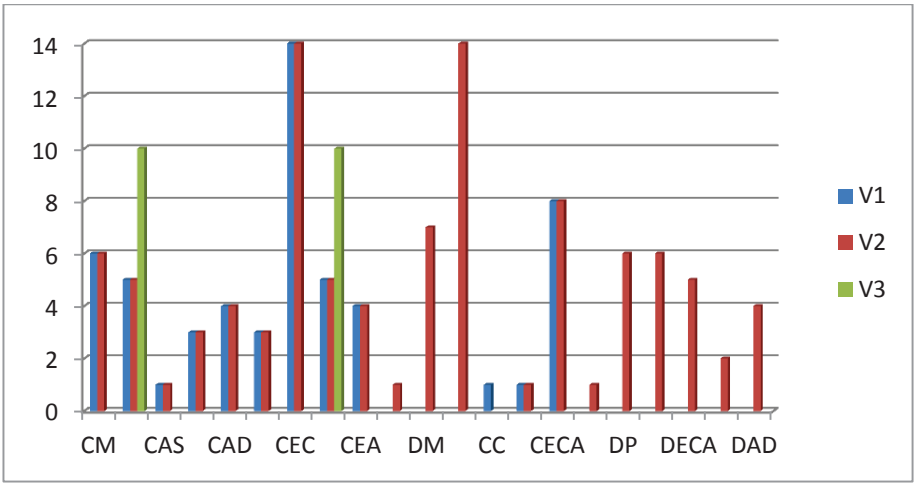

Figure 11. Atomic Rules in Tracing Program

Table 4. Rewrite Rule Patterns

Pattern	Support	Program
CM, CP, CECA	0.33	Tracing
CAS, CPO, CAD, CECR, CECA	0.33	Tracing
CAD, CP	0.24	Figure Editor
CPO, CP, CECR	0.20	Figure Editor

Result Analysis. We have detected two rule patterns for the Tracing program. The first one depicts that frequently, the creation of methods leads to the creation of parameters and edges of the type Calls (calls to other methods). The second one depicts that the creation of an Aspect leads to the creation of pointcuts and advices. And the creation of edges of the type Crosscuts, to specify the Join points. This leads also to the creation of edges calls between advices and methods.

For the Figure Editor program, we have detected two patterns too. The two atomic rules "creation of advice" and the "creation of parameter" are frequently applied together. And the atomic rules: "creation of pointcut", "creation of parameter" and "creation of edge crosscuts" are always applied together. Such rule patterns can help the developer to achieve a complete change i.e. he should not applied a specific change in a pattern without applying the other ones. We believe that the application of our approach to other case studies with large rule-based repository can detect more interesting change patterns.

8. Related Work

This section of the paper presents related work discussing the benefits of our proposal in contrast to the other ones. There are three distinct research areas that are directly related to our work, change-based evolution, AO versioning repositories, and change pattern detection:

Change-based evolution repository: Based on our study of the field, there is a very little work in this research area. The first work that treats this idea for object oriented software is the one of Lanza *et al.*, [24]. They represent a state of a program as an

abstract syntax tree (AST) of its source code. Then, changes to the program are represented as explicit change operations to its AST. Hattori *et al.*, [18] extend this change-based software evolution model [24] into a multi developer context by modelling the evolution of a system as a set containing sequences of changes. Although, the idea of these works is similar to our proposal where the change is treated as a first-class entity, but the use of the AST is not a good choice for software evolution analysis. The AST captures the source code structure but it does not coverage it's semantic, so the change repository is not sufficient for evolution analysis. In contrast, our proposal is promising to better improve and accurate the change repository. We can capture structural as well as semantic information about the change (rewriting rules).

AO versioning repositories: As we presented in Section 3, version repositories of current versioning systems are not satisfactory for AO software evolution analysis. This is why some research works try to adapt current versioning systems to handle the AOP characteristics. For instance, the work in [20] contributes a mechanism that checks-in with the source code versions of crosscutting metadata for tracking the effect of aspects. In [4] the tool TOFRA is presented to address the problem of configuration management in the context of Crosscutting Frameworks (CFs) [11]. However, these works keep the traditional mechanism of classic versioning (file-based, snapshot-based).which do not record the complete information about the AO software evolution. In the other side, the analysis of their repositories becomes a research challenge because the data is unstructured, unlabeled, and noisy. In contrast, our work provides a change-based repository, which stores the complete evolution process, facilitate change extraction, and improve evolution analysis.

Detecting change patterns: There is a plenty of research made on Mining change patterns for procedural or object-oriented programs [21, 17]. Seldom effort is made for AO programs. Qian *et al.*, [26] treat the detection of change patterns in AspectJ programs. They first analyze the successive versions of an AspectJ program, and then decompose their differences into a set of atomic changes. Finally, they employ the Apriori data mining algorithm to generate the most frequent item-sets. However they are based on the repository of current versioning systems, which are not fully adapted to AOP characteristics (Section 3). And need a sophisticated process to extract atomic changes and transactions. Our proposal avoids these problems, where we are based on a rule-based repository. This last stores changes when they occur in more precise and formal format as rewriting rules. This repository is an interesting subject to detect change patterns in AspectJ programs.

9. Conclusion

A sustainable success of an evolution analysis approach depends to a large extent on the version repository used for this analysis. Our research interest is in recording AO software evolution and extracting meta-data from its repository to ease its evolution and predict its future development. In this paper we presented the principles behind a change-based repository for AO software and how they can address some of the problems of AO software evolution.

We proposed a rewriting rule-based repository for AspectJ programs. The evolved AspectJ program is represented as an attributed colored graph, and changes are formulated as rewrite rules. Every rewrite rule is stored directly in the repository when it is applied. Our approach is dedicated to handle the obliviousness characteristic of AOP. It helps to improve program comprehension by making aspect base interaction

more explicit. This does not reduce the obliviousness among system modules, because every module (aspect, class) can be easily observed as an independent module with our representation. Besides, representing and storing changes as rewriting rules preserve the complete information about the change (changed entities, their dependencies, constraints,…*etc.*). This format helps to make the evolution repository more adequate to the crosscutting nature of AO software avoiding the limits of current file-based versioning systems *i.e.*, in contrast to file-based principle of classic versioning tools, our proposal track and store changes in software entities and their dependencies independently of the files they belong to. So, changes in crosscutting dependencies are well stored in our repository.

Besides, we proposed a change pattern detection approach for AspectJ source code. This approach is based on our rule-based repository to extract atomic rewrite rule patterns which are considered as change patterns in AspectJ source code. Those change patterns can be used as measurement aid and fault predication for AspectJ software evolution. This is very important to predict future evolution of AspectJ software, improve the comprehensibility of the software system and consequently decrease the evolution cost.

Using our proposed change pattern detection approach, we proved that it is easier to extract changes from our repository because they are stored in an explicit way. This improves the quality of results of mining efforts. So, we believe that our repository can be an interesting source for a high quality evolution analysis

Finally, we believe that the fundamental approach presented in this paper is generic enough to be adapted to other object oriented or AO programming languages.

References

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases", B. Bocca, M. Jarke, and C. Zaniolo, editors, Proceedings of 20th International Conference on Very Large Data Bases, Santiago, Chile, **(1994)** September 12-15, pp. 487-499.

[2] J. Aldrich, "Open Modules: Reconciling Extensibility and Information Hiding", Proceedings of SPLAT AOSD'04 Workshop, **(2004)**.

[3] R. T. Alexander, J. M. Bieman and A. A. Andrews, "Towards the Systematic Testing of Aspect-Oriented Programs", Report CS-04-105, Colorado State University, Fort Collins-USA, **(2004)**.

[4] M. M. Arimoto, M. I. Cagnin and V. V. de Camargo, "Version control in crosscutting framework-based development", Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC'08), Fortaleza, Ceara, Brazil, **(2008)**, pp. 753-758.

[5] S. Bouktif, Y. G. Guéhéneuc and G. Antoniol, "Extracting Change-patterns from CVS Repositories", Proceedings of 13th Working Conference on Reverse Engineering (WCRE '06), [DOI: 10.1109/WCRE.2006.27], **(2006),** pp. 221-230.

[6] H. Cherait and N. Bounour, "Modeling Software Evolution through Version Control System", Proceedings of 11th African Conference on Research in Computer Science and Applied Mathematics (CARI'12), Algiers, Algeria, **(2012)** October 13-16.

[7] H. Cherait and N. Bounour, "Rewriting Rule-based Model for Aspect Oriented Software Evolution", International Journal of Computer Applications in Technology-Special Issue on Current Trends & Improvements in software Engineering Practices (in press), to appear, **(2013/2014)**.

[8] J. Clark, "XSL, Transformations (XSLT) Version 1.0", Recommandation 16, November edition, http://www.w3.org/TR/xslt, **(1999)**.

[9] A. Colyer and A. Clement, "Large-Scale AOSD for Middleware", Proceedings of 3rd International Conference of Aspect-Oriented Software Development, **(2004)**, pp. 56-65.

[10] A. Corradini, U. Montanari and F. Rossi, "Graph processes", Fundamenta Informaticae, vol. 26, no. 3-4, **(1996)**, pp. 241-265.

[11] V. V. De Camargo and P. C. Masiero, "A pattern to design crosscutting frameworks", Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC'08), Fortaleza, Ceara, Brazil, **(2008)**, pp. 759-764.

[12] H. Ehrig, K. Ehrig, U. Prange and G. Taentzer, "Fundamentals of Algebraic Graph Transformation", EATCS Monographs in Theoretical Computer Science, Springer, ISBN 978-3-540-31187-4, **(2006)**.
[13] F. Ferrari, R. Burrows, O. Lemos, A. Garcia, E. Figueiredo, N. Cacho, F. Lopes, N. Temudo, L. Silva, S. Soares, A. Rashid, P. Masiero, T. Batista and J. Maldonado, "An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs", Proceedings of ICSE '10, Cape Town, South Africa, ACM press [DOI : 978-1-60558-719-6/10/05], **(2010)** May 2-8, pp. 65-74.
[14] R. E. Filman, T. Elrad, S. Clarke and M. Aksit, "Aspect-Oriented Software Development", Addison-Wesley **(2004)**.
[15] R. E. Filman and D. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", In: Aspect-Oriented Software Development, Addison-Wesley, **(2004)**.
[16] B. Fluri, "Change Distilling Enriching Software Evolution Analysis with Fine-Grained Source Code Change Histories", Dissertation for the Degree of a Doctor in Informatics, Department of Informatics, University of Zurich, **(2008)** October.
[17] A. E. Hassan, "The road ahead for mining software repositories", Frontiers of Software Maintenance**, (2008),** pp. 48–57.
[18] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development", Proceedings of 32nd ACM/IEEE International Conerence on Software Engineering, IEEE Computer Society [DOI: 10.1145/1810295.1810339], **(2010)**, pp. 235-238.
[19] R. Heckel, J. M. Kuster and G. Taentzer, "Confluence of Typed Attributed Graph Transformation Systems", In Proceedings of First International Conference, ICGT'02, Barcelona, Spain. Springer-Verlag, LNCS, [DOI: 10.1007/3-540-45832-8_14], vol. 2505, **(2002),** pp. 161-176.
[20] S. Ifrah and D. H. Lorenz, "Crosscutting Revision Control System", Proceedings of ICSE, Zurich, Switzerland, IEEE Computer Society [DOI: 978-1-4673-1067-3/12], **(2012)**, pp. 321-330.
[21] H. Kagdi, M. L. Collard and J. I. Maletic, "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution", Journal of Software Maintenance and Evolution: Research and Practice, vol. 19, no. 2, **(2007)**, pp. 77-131.
[22] C. Kastner, S. Apel and D. Batory, "A Case Study Implementing Features Using AspectJ", Proceedings of SPLC'07, **(2007)**, pp. 223-232.
[23] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. M. Loingtier and J. Irwin, "Aspect-oriented programming", Proceedings of 11th European Conference on Object-Oriented Programming, Springer-Verlag, LNCS, [DOI: 10.1007/BFb0053381], vol. 1241, **(1997)**, pp. 220-242.
[24] M. Lanza and R. Robbes, "A Change-based Approach to Software Evolution", Electronic Notes in Theoretical Computer Science (ENTCS). Elsevier, [DOI: 10.1016/j.entcs.2006.06.015], vol. 166, **(2007)**, pp. 93-109.
[25] T. Mens, "A state-of-the-art survey on software merging", IEEE Trans. Softw. Eng, vol. 28, no. 5, **(2002)**, pp. 449-462.
[26] Y. Qian, S. Zhang and Z. Qi, "Mining Change Patterns in AspectJ Software Evolution", Proceedings of International Conference on Computer Science and Software Engineering, **(2008)**, pp. 108-111.
[27] A. Rashid, T. Cottenier, P. Greenwood, R. Chitchyan, R. Meunier, R. Coelho, M. Südholt and W. Joosen, "Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe", Published by the IEEE Computer Society, **(2010)** February.
[28] T. Schultzke and C. Ermel, "AGG Environnement: A Short Manual", Short manual edition, User Manual, http: //tfs.cs.tuberlin.de/ agg/ShortManual.ps, **(2013)** January.
[29] F. Steimann, "The Paradoxical Success of Aspect- Oriented Programming", Proceedings of OOPSLA'06, **(2006)**, pp. 481-497.
[30] J. Suzuki and Y. Yamamoto, "Managing the software design documents with xml", Proceedings of the 16th annual international conference on Computer documentation, ACM Press: New York [DOI: 10.1145/296336.296366], **(1998)**, pp. 127-136.
[31] G. Taentzer, C. Ermel, P. Langer and M. Wimmer, "A fundamental approach to model versioning based on graph modifications: from theory to implementation", Software and Systems Modeling, Springer-Verlag, [DOI 10.1007/s10270-012-0248-x], **(2012)**.
[32] The AspectJ Team, The AspectJ Programming Guide, Online manual, http://eclipse.org/aspectj/, **(2012)** December.
[33] J. W. W. Wan and G. Dobbie, "Extracting Association Rules from XML Documents using XQuery", Proceedings of WIDM'03, New Orleans, Louisiana, USA, **(2003)** November 7-8, pp. 94-97.
[34] A. Winter, B. Kullbach and V. Riediger, "An overview of the GXL graph exchange language", Proceedings of International Seminar Dagstuhl Castle, Germany, **(2001)**, Springer-Verlag, LNCS, [DOI:10.1007/3-540-45875-1_25], vol. 2269, pp. 324-336.

Authors

Hanane Cherait is a Ph.D student in Complex Software Engineering. She obtained her Master of Science degree in Computer Science from the University of Badji Mokhtar – Annaba (UBMA), Algeria in 2009. Her research interests include software evolution; aspect oriented programming and software reverse engineering.

Dr Nora Bounour received her Doctorate degree in the department of computer science at the University of Badji Mokhtar -Annaba (UBMA), Algeria in the year 2007. She is presently working in the same department as associate professor. She is the head of the research group on reengineering and evolution of complex systems at the Laboratory of complex system engineering (LISCO). Her research interests include software evolution and reverse engineering methodologies, separation of concerns and aspect oriented programming.