Dynamic Analysis and Association Rules for Aspects Identification

Esteban S. Abait¹, Santiago A. Vidal¹, Claudia A. Marcos¹²

¹Facultad de Ciencias Exactas - Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA)
Campus Universitario (CP 7000) - Tandil - Bs. As. - Argentina

²ISISTAN Research Institute - Facultad de Ciencias Exactas - Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA)
Campus Universitario (CP 7000) - Tandil - Bs. As. - Argentina
Tel/Fax: +54 - 2293 - 440362/3
http://www.exa.unicen.edu.ar/~isistan/
{eabait,svidal}@alumnos.exa.unicen.edu.ar, cmarcos@exa.unicen.edu.ar

Abstract. One of the main problems with current object-oriented systems is the presence of crosscutting concerns, which makes software comprehension and modularization difficult. Aspect mining is the task of identifying the crosscutting concerns present in legacy systems. In this paper, a semi-automatic four-step approach for aspect mining is proposed based on dynamic analysis and association rules. In a preliminary evaluation in two systems, the proposed approach successfully discovered a number of crosscutting concerns (candidate aspects) already identified for these systems by existing approaches.

1. Introduction

Aspect-oriented programming (AOP) [Kiczales et al. 1997] is a paradigm that aims at a better separation of concerns in software systems. Some concerns crosscut the dominant decomposition of the application and are called crosscutting concerns. Examples of such concerns are tracing, persistence or synchronization. Code related to these crosscutting concerns can show two symptoms of bad modularization: it can be scattered over the whole project or it can be tangled with other code [Hannemann and Kiczales 2001]. AOP introduces a new modularization unit, called an aspect, which is intended for the encapsulation of these crosscutting concerns. Since many systems are implemented using object-oriented languages, one of the main goals is to develop mechanisms and techniques to migrate these object-oriented systems to aspect-oriented ones. A first step towards achieving this goal is to discover the different crosscutting concerns present in the code, in order to decide whether they are aspect candidates for a future system. The task of identifying the crosscutting concerns that are amenable for an aspect-oriented implementation is called aspect mining [Kellens et al. 2007].

In this paper, we present a tool approach for aspect mining that is based on the analysis of execution traces of a system through association rules algorithms. As a result of this analysis, our tool is able to generate rules for identifying scattering symptoms.
These rules allow developers to find candidate aspects in the code. The proposed aspect mining process involves four steps: the first two correspond to the dynamic analysis phase, and the other two correspond to the generation and filtering of the association rules. Dynamic analysis (section 2.1) is used to obtain the data that will be input for the association rule mining algorithm (section 2.2). Then, the post-processing step filters out rules that are not interesting for the developer and classifies the remaining rules as scattering indicators (section 3). We have applied our approach to different existing object-oriented applications, such as an implementation of the Observer design pattern (section 3) and JHotDraw (section 4) the de facto benchmark for aspect mining [Ceccato et al. 2005]. Finally, we discuss related works on the subject and present the conclusions (sections 5 and 6).

2. Dynamic Analysis and Association Rules

Our approach is based on the fact that through association rules it is possible to get the most relevant method associations within the execution traces obtained using dynamic analysis. This kind of associations give developers valuable information about the dynamic behavior of the system and allows the identification of scattering symptoms. In this section, the basic concepts of dynamic analysis and association rules are introduced.

2.1. Dynamic Analysis

The basic idea behind dynamic analysis algorithms is to observe run-time behaviours of software systems and to extract information from the execution of the programs [Breu and Krinke 2004]. The approach described here takes two kinds of information as input: execution traces and execution relations. The execution traces and relations are obtained by running the program under given scenarios, where each scenario is like a use case instance as defined in [Booch et al. 1999]. The program trace is the sequence of method invocations during the execution of the program, and the execution relations registers which methods are called from each other.

![Figure 1. UML class diagram for the program under analysis.](image)

Figure 1 shows the UML class diagram of the Observer design pattern [Gamma et al. 1995] in a visual interface application. The intent of the Observer pattern is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [Gamma et al. 1995].
This implementation shows an instance of the pattern where the Point class plays the subject role and Screen plays the roles of both subject and observer (of Point and itself). For example, running the scenario *the point changes its color* generates the trace and execution relations shown in Figure 2.

The resulting trace contains the sequence of method invocations shown by the table atop of Figure 2. The execution relations for this trace are represented by the two columns at the bottom of Figure 2.

Since a trace is made of method calls obtained during program execution, different execution of the same program can yield different traces. This is a consequence of having use cases or scenarios that exercise different parts of the system. In spite of this, many times the same class could collaborate in the realization of multiple use cases, so a method belonging to a class can be in several use case traces. Based on the previous observation, it is possible to identify the recurring patterns present in the traces by applying association rules to them so as to find the most relevant associations among method executions.

### 2.2. Association Rules Concepts

Association rule algorithms find interesting association or correlation relationships among a set of items in a given domain. Hence, applying an association rule algorithm over a set of traces would result in the discovery of interesting execution patterns between methods.

Let \( I = \{i_1, i_2, \ldots, i_m\} \) be a set of items and \( D = \{T_1, T_2, \ldots, T_n\} \) a set of transactions, where each transaction \( T \) is a set of items such that \( T \subseteq I \). An association rule is an implication of the form \( A \Rightarrow B \), where \( A \subseteq I, B \subseteq I \) and \( A \cap B = \emptyset \) [Agrawal and Srikant 1994]. \( A \) is the antecedent of the rule and \( B \) is its consequent. The rule \( A \Rightarrow B \) holds in the transaction set \( D \) with support \( s \), where \( s \) is the percentage of transactions in \( D \) that contains \( A \cup B \). The rule \( A \Rightarrow B \) has confidence \( c \) in the transaction set \( D \) if the percentage \( c \) is the percentage of transactions in \( D \) containing \( A \) which also contain \( B \). Rules that satisfy both a minimum support threshold (\( \text{min_supp} \)) and a minimum confidence threshold (\( \text{min_conf} \)) are called strong [Han and Kamber 2000], and they are the output of an association rule mining algorithm. We use the Apriori algorithm [Agrawal and Srikant 1994] in order to obtain the association rules from a set of program traces. The association rule mining process consists of two well-defined steps:
the first step is to find all the itemsets (set of items) with minimum support \( \text{min\_sup} \), and the second step is to generate the association rules with minimum confidence value \( \text{min\_conf} \) from the set of itemsets obtained.

3. Generation of Association Rules for Aspect Identification

In this section the aspect mining technique is described in detail. First, we explain how to generate the association rules from the dynamic information obtained executing the analysed system. Then, the main steps of the mining process are presented, and the application of an association rule algorithm to the Observer example is discussed. The classification filters are presented and discussed, and its application to the Observer example are shown.

3.1. Association Rules Mining

If each trace of the system under analysis is considered as a transaction \( T \) and the methods contained in all the traces as the set of items \( I \), it is possible to get a dataset \( D \) from which a set of association rules can be generated. For example, the rules for the example shown in Figure 1 will have the following form: \( \text{Point.notifyObservers} \Rightarrow \text{Screen.refresh} (s: 1.0, c: 1.0) \). The support value of the rule indicates the number of traces (transactions) in which both methods are present. In the example above, the support value indicates that the two methods are present together in all the traces. On the other hand, the confidence value points out the stability of the method relation, then a confidence value of 1.0 means that each time the notifyObservers method is called so is the refresh method. For the proposed technique, the generated rules have only one item in its antecedent and one item on its consequent.

3.2. Mining Workflow

In Figure 4, the main steps of the proposed mining technique are shown. The first and second steps (System instrumentation and System executions) correspond to dynamic analysis of the system. The third step (Association rule mining), takes as input the set of traces obtained and uses an association rule algorithm to find interesting associations among methods. The fourth step (Association rule post-processing) classifies rules as scattering indicators, and removes redundant rules as well as rules with utility methods such as main or run. Rules that can not be classified are discarded.

Figure 4. Workflow of the proposed aspect mining technique.
3.3. Association Rules for the Observer Example

In order to characterize the kind of rules that are *interesting* for the aspect mining process, this section shows the results of applying an association rule algorithm to the dynamic information obtained from the Observer example (Figure 1).

First, the execution traces and relations are obtained by means of a tracing aspect that registers all the method invocations and its relations. Two scenarios are used to execute the implementation: a) "a point changes its color", b) "a point changes its position".

Table 1. Some of the rules obtained for the Observer example.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Support</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Main.main ⇒ Point.addObserver</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>Main.main ⇒ Point.setColor</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>Point.setColor ⇒ Point.notifyObservers</td>
<td>0.5</td>
</tr>
<tr>
<td>4</td>
<td>Point.setX ⇒ Point.notifyObservers</td>
<td>0.5</td>
</tr>
<tr>
<td>5</td>
<td>Screen.addObserver ⇒ Point.addObserver</td>
<td>1.0</td>
</tr>
<tr>
<td>6</td>
<td>Point.addObserver ⇒ Screen.addObserver</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Then, running the Apriori algorithm over the traces with support value of 0.1 and confidence value of 0.1 resulted in the generation of 70 rules (partially showed on Table 1). The resulting set of rules demonstrate the importance of the post-processing step, and the need for classification filters providing more information for each rule. For example, rules that include methods like main (rules 1 and 2), toString, hashCode are not interesting for aspect mining purposes. Thus, the first filter must remove rules that include those 'irrelevant methods'. Redundant rules also must be removed from the final list of rules. For example, rules (5) and (6) show the same association between methods and have the same support and confidence value. Hence, the second filter must remove the 'redundant rules'.

The next section describes the classification filters.

3.4. Classification Filters

The classification of the association rules is done by two filters. The first one looks for methods that have the same name and are called together (name filter), whereas the second one looks for rules that share the same consequent (recurrent consequent filter). Next, both filters are discussed.

**Name Filter**

Rules like Screen.addObserver ⇒ Point.addObserver (s: 1.0, c:1.0) could be indicators of a concern that is scattered over two different methods. This is not only because they

---

1 For this experiment we used parameters with very low values in order to maximize the number of rules generated. This is because we wanted to see which kind of rules were of interest for aspect mining purposes and which were not.
share the same name (addObserver), but because they are present together in more than one execution trace (high confidence and support values). This means that both methods were called during the system execution for more than one scenario, thus both methods could correspond to the implementation of the same concern. This latter condition avoids many false positive that could arise if we only consider the syntactic nature of the methods name. For this kind of filter, the confidence value depicts how semantically related both methods are, since the confidence indicates how many times the antecedent method is executed in conjunction with the consequent method. High confidence values enforce the assumption that both methods are semantically related. Then, this 'name filter' is simply defined as follow: given an association rule $A \Rightarrow B$, where $A$ and $B$ are methods, the name of $A$ must be equal to the name of $B$.

**Recurrent Consequent Filter**

When two or more rules share the same consequent (for example, rules (3) and (4) of Table 1), the immediate assumption is that the method of the consequent is consistently invoked from the methods included in the antecedents of the rules. The method of the consequent could be implementing functionality that is required from various places of the system (like a 'log' method). Therefore, the existence of such method is a possible indicator of scattering symptom on the system. The 'recurrent consequent filter' is defined as follow: given an association rule $A \Rightarrow B$, where $A$ and $B$ are methods, the following conditions must hold,

- $A$ and $B$ must be in a execution relation where $A$ is the invoker and $B$ is the invoked method,
- $B$ must be included as a consequent in another association rule $C \Rightarrow B$ that also is in a execution relation where $C$ is the invoker and $B$ is the invoked method.

The application of this two filters along with the redundant rules and the irrelevant methods filters yield the rules shown in Table 2. The concern column of the table must be completed by the user of the technique after manual investigation of the rule on the source code.

**Table 2. Final set of rules for the Observer pattern example.**

<table>
<thead>
<tr>
<th>Concern</th>
<th>Rule</th>
<th>Filter</th>
<th>Supp.</th>
<th>Conf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject-Observer Mapping</td>
<td>Screen.addObserver ⇒ Point.addObserver</td>
<td>Name filter</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Notification mechanism</td>
<td>Screen.notifyObservers ⇒ Point.notifyObservers</td>
<td>Name filter</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Notification mechanism</td>
<td>Point.setColor ⇒ Point.notifyObservers</td>
<td>Recurrent consequent</td>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td>Notification mechanism</td>
<td>Point.setX ⇒ Point.notifyObservers</td>
<td>Recurrent consequent</td>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td>Update logic</td>
<td>Point.notifyObservers ⇒ Screen.refresh</td>
<td>Recurrent consequent</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Update logic</td>
<td>Screen.notifyObservers ⇒ Screen.refresh</td>
<td>Recurrent consequent</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
In [Hannemann and Kiczales 2002], the same implementation of the pattern is analysed and refactored to AspectJ [Kiczales et al. 1997]. The authors point out that the notification mechanism, the subject-observer mapping and the update logic are concerns that crosscut the classes participating in the pattern. The application of our technique to this example allowed us to discover the same crosscutting concerns previously identified by Kiczales et al.

4. Case Study: JHotDraw

This section presents the results of applying our approach to version 5.4b1 of JHotDraw [JHotDraw], a Java object-oriented framework, with approximately 18,000 non-commented lines of code and around 2800 methods. JHotDraw is a framework for drawing structured 2D graphics and was originally developed as an exercise to illustrate good use of object-oriented design patterns [Gamma et al. 1994]. Since its original adoption, JHotDraw has been an application for many aspect mining studies. In order to compare and validate our mining results on JHotDraw, we have compared the crosscutting concerns discovered by [Marin et al. 2007] and [Ceccato et al. 2005] against those discovered by our technique.

In order to analyse JHotDraw, we defined 21 scenarios according to the main functionalities described in the documentation. For example, we created scenarios for drawing a rectangle, for painting a rectangle, for opening a document, etc. When the scenarios were executed they exercised 610 methods (out of 2800 methods). Then, the application of our approach over the 21 traces produced 479 rules, using a minimum support of 0.1 and a minimum confidence value of 0.8. The support value was set in order to obtain rules that are present in at least two traces. Table 3 show some representative rules.

<table>
<thead>
<tr>
<th>Concern</th>
<th>Rule</th>
<th>Supp.</th>
<th>Conf.</th>
<th>Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manage</td>
<td>RelativeLocator.south ⇒ BoxHandleKit.south</td>
<td>0.14</td>
<td>1.0</td>
<td>Name filter</td>
</tr>
<tr>
<td>Observer</td>
<td>AbstractFigure.moveBy ⇒ AbstractFigure.changed</td>
<td>0.14</td>
<td>1.0</td>
<td>Rec. Cons.</td>
</tr>
<tr>
<td></td>
<td>AttributeFigure.setAttribute ⇒ AbstractFigure.changed</td>
<td>0.19</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>Persistence</td>
<td>RectangleFigure.read ⇒ AbstractFigure.read</td>
<td>0.14</td>
<td>1.0</td>
<td>Name filter</td>
</tr>
<tr>
<td>Undo</td>
<td>DragTracker.activate ⇒ AbstractTool.getUndoActivity</td>
<td>0.1</td>
<td>1.0</td>
<td>Rec. cons.</td>
</tr>
<tr>
<td></td>
<td>UndoableTool.deactivate ⇒ AbstractTool.getUndoActivity</td>
<td>0.14</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>

The "bring to front/send to back" functionality reported as a crosscutting concern in [Ceccato et al. 2005] was not identified by the technique, since the scenarios for that functionality were not exercised. Overall, our technique was able to detect almost all the crosscutting concerns that were previously identified by the other two approaches. The only concern not found shows us one of the main limitations present in all the approaches based on dynamic analysis: results depend upon the scenarios used to exercise the system.
5. Related Work

Similar to our work, [Breu and Krinke 2005] analyse the program traces for recurring patterns of methods executions, such patterns are considered aspect candidates if they occur more than once in a uniform way. To ensure that the recurring relations are sufficient crosscutting they should appear in a different 'calling context'. Those patterns are very similar to the "recurrent consequent" filter of the proposed approach. Our technique also allows the application of other filter (such as the name filter). Furthermore, the developer is able to develop his own filters codified with his domain knowledge. Another dynamic approach [Tonella and Ceccato 2004], analyse the execution traces through formal concept analysis. The aspect candidates are extracted from the resulting lattice and corresponds to tangling and scattering symptoms. The main advantage of our approach is that the interpretation of a list of rules is easier than interpreting a lattice of concepts. On the other hand, they can identify tangling symptoms which we can not.

6. Conclusions and Future Work

We have developed a semi-automatic aspect mining technique based on dynamic analysis and association rules. This technique is a four-step process that yields a list of association rules that represents scattering indicators. The main advantages of the technique are: the possibility of automatically identifying scattering symptoms, and of obtaining rules that have a very expressive power. Furthermore, our technique allows the development of new filters facilitating the implementation of better heuristics for aspect identification over a list of rules. For example, the user could use his own domain knowledge to develop a filter that looks for a particular kind of rule. On the other hand, the proposed technique still has some problems and open issues, namely: the analysis is partial in the sense that not all the method involved in an aspect are retrieved. For example, the technique recognized the presence of the Observer pattern in JHotDraw but it did not recognized all of the pattern instances present in the code. Furthermore, the identified aspects depend upon the scenarios exercised on the system, which means that some aspects candidates could be missed (as the 'bring to front/send to back' functionality of JHotDraw). Finally, the case-studies allows us to perform a preliminary validation of the results of the technique, identifying drawbacks and possible improvements for them. For future work, we want to compare the proposed technique with others dynamic approaches in order to improve the existing knowledge on this kind of aspect mining techniques.

References


[JHotDraw] www.jhotdraw.org


