Automated Reasoning on Aspects Interactions

Paolo Falcarin, Marco Torchiano
Politecnico di Torino, Corso Duca degli Abruzzi 24, Torino, Italy
Paolo.Falcarin@polito.it, Marco.Torchiano@polito.it

Abstract

The aspect-oriented paradigm allows weaving aspects in different join points of a program. Aspects can modify object fields and method control flow, thus possibly introducing subtle and undesired interactions (conflicts) among aspects and objects, which are not easily detectable.

In this paper we propose a fully automated approach to discover conflicts among classes and aspects directly from Java bytecode. The novelty of this work is the usage of a rule engine for identifying possible conflicts among advices, methods, and fields.

The knowledge base is obtained through static analysis of classes and aspects bytecode. The possible conflicts are represented by means of rules that can be easily extended and customized.

1. Introduction

Aspect-Oriented Programming [1] (AOP) is a powerful methodology, but imprudent use of aspects may complicate development and debugging tasks. Badly designed aspects may interact harmfully with methods and other aspects changing the control flow and modifying shared data, like objects fields.

The work presented in this papers aims at helping developers and maintainers to automatically analyze an aspect-oriented code base for discovering conflicts due to aspects interactions.

We developed JECOM (Java Extensible Conflict Manager), a tool able to extract interactions among aspects and classes, and read-set and write-set of methods and advices.

These data are transformed in a set of "facts" populating the knowledge base of a rule engine. We represent potential conflicts with rules to be matched on the knowledge base executing the rule engine.

The main contributions of this paper are:

- the introduction of a rule engine to detect potential conflicts,
- a method for producing a knowledge base from the code and the relative implementation,
- An approach for expressing potential conflicts through rules, allowing easy extensibility.

We assume the reader is familiar with AOP concepts (non included for space reason).

In next sections the conflict management problem and related work are described; after that we describe our approach, and we draw conclusions.

2. Classification of Conflicts

AOP allows insertion of (aspect) code throughout the application code base. Understanding the behavior of an aspect-oriented application can be difficult because an aspect can interact with several classes and other aspects. While some of these interactions may have been explicitly designed by the developer, others may be unwanted and can be considered as *conflicts*, i.e. side-effects, resulting, for example, from bad wildcards usage, or risky refactoring of the application codebase.

The use of wildcards in pointcut definition is helpful but its real impact cannot be identified when the pointcut is written. Moreover, dynamic pointcuts can match different join-points at different times, depending on data values (e.g. the 'cflow' construct in AspectJ); in addition, using the 'if' construct in AspectJ, the advice execution depends on data in the 'if' condition.

The usage of these powerful pointcuts limit the possibility of modeling all aspect-class interactions, and, as a consequence, the possibility to reason about them using static analysis [9] has to exclude dynamic pointcuts, because the behavior of the corresponding advices cannot be deduced at compile-time.

All of the above problems contribute to the possible unexpected composition of different aspects at the same join-point at compile-time or run-time.

AOP tools like AJDT [8] can discover and visualize such simple interactions, but there are other ones that are not detected. AJDT compiler limits risks of such interactions defining precedence rules among aspects and among advices in the same aspect.

Our approach allows identifying interactions in AspectJ applications and allows developer customizing rules in order to find out more complex interactions on the code base and to evaluate the impact of each aspect on the classes.

Tessier et al [6] classify aspects interactions on different criteria. Depending on involved elements there are intra-aspect interactions (between advices in the same aspect), inter-aspect interaction (e.g. when an aspect's advice acts on the advice of a different aspect), and the typical aspect-class interactions.

Moreover conflicts can be considered 'static' if they can be identified at compile-time and 'dynamic' if they can be detected at run-time, e.g. during testing.

The kind of interaction between an advice and methods has been used by Rinard et al [5] to classify advices in four categories, namely: augmentation, narrowing, replacement, and combination. If, after weaving, the entire body of the method always executes (e.g. with read-only aspects, like logging and monitoring), the advice is an augmentation one. A narrowing advice can decide if a method will be executed or not (e.g. an advice that checks pre-conditions before allowing the method to execute), while a replacement advice substitutes the whole method. All other cases are considered combination advices, i.e. when the method and aspect interact in another way.

Rinard et al [5] also defines *scopes* as set of fields accessed by an object or by an aspect. When an aspect reads data modified by another aspect, the interaction is an 'observation'; when one aspect modifies data read by the other one, then it is an 'actuation', and when both modify some shared data it is a 'combination'. Moreover if aspects both read the same data then they are 'independent', while if no join-points are shared they are 'orthogonal'.

According to this classification, a "conflict-free" application only presents orthogonal, independent and observation interactions, and where advices are all of type augmentation or narrowing.

3. JECOM Approach

JECOM (Java Extensible Conflict Manager) aims at helping developers and maintainers to automatically analyze an aspect-oriented code base for discovering conflicts due to aspects interactions.

The interactions analysis has four sequential phases:

1. *Bytecode Analysis* of the target application to extract interactions among aspects and classes directly form bytecode;
2. *Knowledge Base Creation*: translating the above-mentioned information in a set of "facts" populating the Knowledge Base;
3. *Rule Base creation*: reusing or augmenting the set of rules representing potential undesired interactions;
4. *Conflict Analysis*: running the rule engine for querying the collected data and detecting if some potential conflicts exist in the target application.

In next section we introduce the usage of Bernstein's conditions for reasoning on aspects interactions.

3.1 Bernstein's Conditions in AOP

In operating systems theory it is possible to evaluate if two processes can run in parallel, checking if the input data-sets of two processes are independent of each other's output data-sets, and if their output data-sets are independent. Bernstein [3] formalized these constraints by means of three conditions which can be expressed using set-theory, since they predicate two kinds of sets: the *Read-Set* and the *Write-Set* of a process, which are respectively the set of data read by a process and the set of data written by a process.

In this work we consider the read/write-sets of each method/advice as the fundamental information for detecting interactions between aspect's advices and object's methods. In this work a read/write set is a set of class fields.

If R_A and W_A respectively denote Read-Set and Write-Set of an advice A, and R_M and W_M respectively denote Read-Set and Write-Set of a method M then Bernstein's conditions can be rewritten as follows:

1. $R_A \cap W_M = \emptyset$
2. $W_A \cap R_M = \emptyset$
3. $W_A \cap W_M = \emptyset$

In practice the specified sets must be disjoint in all three conditions in order to guarantee independence between an advice and a method.

Rinard et al [5] defines *scopes* as set of fields accessed by a method or by an advice. Following their terminology, the first Bernstein condition holds in case of observation interaction, the second one holds in case of actuation, and the third one in case of combination. When all Bernstein conditions hold, the advice scope and the method scope are independent, while they are orthogonal if all the Bernstein conditions hold and their read sets are disjoint ($R_A \cap R_M = \emptyset$).

3.2 From Bytecode to Facts

Analyzing an aspect-oriented codebase requires an intermediate step of modeling the knowledge base Understanding the interactions among aspects and classes, requires acquisition of data from the code in order to build a model.

Our tool analyzes the Java bytecode of an application built with the AspectJ compiler.

Tessier et al [6] have developed a model for representing interactions among aspects and classes in the early design phase.

This model is a graph of nodes representing aspects and classes while edges represent a join-point, i.e an interaction between an aspect and a class.

We extended this model to represent relationship between a method and an advice and between advices: our model has a finer granularity because the basic model elements are shared variables (fields), functions (methods or advices), and interactions.

In our model, the interaction between an aspect and a class is decomposed into a set of Link structures representing the interaction between an advice and a join-point in a class or in another aspect. We model the Link structure as a tuple of the following elements:

- aspectID: is the fully-qualified name of the aspect;
- adviceID: is the fully-qualified name of the advice;
- adviceType: can be of three types (before, after, around);
- declarationOrder: is the number (extracted from the bytecode) identifying the position of the advice within the aspect;
- classID: is the fully-qualified name of the class or aspect advised by aspectID;
- methodID: is the fully-qualified name of the method (or advice) advised by adviceID;
- joinPointID: is a unique identifier of the join-point in the advised class, and it is extracted directly form the bytecode of the advised class (or aspect).

The resulting graph obtained by merging all the extracted Links, can have big dimensions on a large codebase. In order to shrink the analysis on aspects of interest, the developer can choose the aspect from which start the reverse engineering activity.

Each link is translated in a fact and inserted in the knowledge base of the rule engine.

This graph is not enough for discovering all kinds of interactions. Some conflicts indeed can be due to Bernstein conditions violation.

In the knowledge base a Read/Write Set is an unordered set of RWS tuples, which represents a read/write access; each tuple is formed by these slots:

- classID: is the fully-qualified name of the class (or aspect) containing the accessed field;
- setType: can be "RS" in case of a read access, or "WS" for a write access;
- methodID: is the name of method or advice accessing the field;
- fieldId: is name of the field accessed by the methodID.

3.3 Reasoning on Code

A rule engine reasons on facts applying pattern-matching techniques, in order to identify which facts satisfies a rule. A rule engine, in the simplest terms, continuously applies a set of if-then statements (*rules*) to a set of assertions (*facts*). In our work a fact is a tuple representing a type of structured information, i.e. the interactions among classes, aspects and their methods, advices and fields.

In our work the rule base can be augmented by the developer and the knowledge base is updated each time the aspect-oriented application is rebuilt, and a new bytecode version has to be inspected.

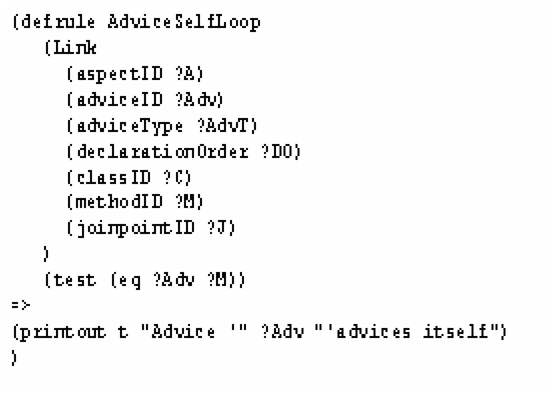
JECOM relies on Jess[1] [7] rule engine, which is developed in Java, it has good performance (improving the typical pattern matching Rete algorithm [2]), and it offers a Java API (compliant with the standard JSR 94 [2]) for adding facts and rules to the working memory, but it can also load the knowledge base from files written in Jess language.

The knowledge base is obtained through static analysis of classes and aspects bytecode. The possible conflicts are represented by means of rules that can be easily extended and customized.

The following figure depicts a rule written in the Jess language. This rule allows detecting all advices which advise themselves recursively in an aspect-oriented codebase. The rule is applied on the knowledge base composed of a set of *Link* facts, each one representing an interaction between an advice and a join-point, by means of a tuple of elements. Each element of this tuple is a couple of strings representing a variable (identified by a starting question mark in its identifier) and its type. For example '*?A*' is the aspect name and *aspectID* means that A can represent one of the aspect names in the application.

Each element of this tuple is a couple of strings representing a variable (identified by a starting question mark in its identifier) and its type. For example '*?A*' is the aspect name and *aspectID* means that A can represent one of the aspect names in the application.

[1] Jess is a registered trademark of Sandia National Laboratories. Jess source code, binary code and all Jess documentation associated with Jess code is owned and under copyright registration by Sandia Corporation.



```
(defrule AdviceSelfLoop
   (Link
      (aspectID ?A)
      (adviceID ?Adv)
      (adviceType ?AdvT)
      (declarationOrder ?DO)
      (classID ?C)
      (methodID ?M)
      (joinpointID ?J)
   )
   (test (eq ?Adv ?M))
=>
(printout t "Advice '" ?Adv "'advices itself")
)
```

Figure 1. Rule detecting recursive advice

Moreover, the variable *Adv* can contain whichever advice name, and *M* can contain whichever method name, among the existing names inserted in the knowledge base after the bytecode analysis phase.

Given this Link structure, the rule matches if the name of the advice *Adv* is equal (see the *eq* operator, in prefix notation in Jess language) to the advised method name *M*, which can be also an advice. The rule engine simply looks for values of Adv and M which are equals among all the Link structures (i.e. the direct interactions) stored in the knowledge base.

A developer can add new customized rules for identifying particular interactions on a subset of the code base; as a consequence it is possible to evaluate the impact of a single aspect insertion detecting the classes (and their fields) which are directly affected by aspect advices.

Ideally, the developer can add rules for discovering indirect interactions of further levels but the performance of this deeper analysis can degrade depending on dimensions of the code base.

4. Conclusions

The detection of interference and conflicts among aspects is an emerging research field.

The existing aspect-oriented tools are recently offering more assistance to programmers in the detection of direct interactions among aspects and classes: for example AJDT visualizes all these interactions in the development environment, but it is not able to understand if this interaction can be harmful (i.e. a conflict) and, as a consequence, raising an appropriate warning.

The novelty of our work is the usage of a rule engine for identifying possible conflicts among advices, methods, and fields; this approach allows developer creating customized rules to identify potential conflicts (direct and indirect) on the whole codebase or on a subset related to an aspect.

JECOM tries to help developers in understanding the aspect-oriented program in case of indirect interactions, and evaluating the impact of a new aspect insertion in a pre-existing codebase.

Further work will consist on validating JECOM on larger codebase and allowing the user to modify the knowledge base, removing useless facts, i.e. removing interactions which have been considered safe during previous analysis (i.e. using the rule engine as an expert system).

Acknowledgments

The authors want to thank Alessandro Di Giovanni for his valuable contribution to this work.

5. References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-oriented programming," presented at 11th European Conference Object-Oriented Programming, 1997.

[2] C. L. Forgy, "Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem". On *Artificial Intelligence* 19 (1982), 17-37.

[3] A. J. Bernstein, "Program analysis for parallel processing". *IEEE Trans. On Electronic Computers.* Ottobre 1966. pp. 757-762.

[4] The Java Rule Engine API (JSR 94) specification. On-line at http://www.jcp.org/en/jsr/detail?id=94.

[5] M. Rinard, A. Salcianu, and S. Bugrara, "A Classification System and Analysis for Aspect-Oriented Programs". In *Proc. 12th Int. Symposium on the Foundations of Software Engineering (FSE-12)*, Newport Beach, USA, Nov. 2004.

[6] F. Tessier, M. Badri, and L. Badri, "A Model-Based Detection of Conflicts Between Crosscutting Concerns: Towards a Formal Approach". *Int. Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, September 2004, Beijing, China.

[7] Jess project homepage. On-line at http://herzberg.ca.sandia.gov/jess/

[8] AJDT project. On-line at http://eclipse.org/ajdt

[9] D. Sereni, and O. de Moor, "Static analysis of aspects". In *Proc. 2nd int. conference on Aspect-Oriented Software Development*, pages 30–39. ACM Press, 2003.