Comprehension and change impact analysis of aspect-oriented programs through declarative reasoning

Laleh Mousavi Eshkevari

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada
January 2008

Your file Votre référence
ISBN: 978-0-494-40950-3
Our file Notre référence
ISBN: 978-0-494-40950-3

Abstract

Comprehension and change impact analysis of aspect-oriented programs through declarative reasoning

Laleh Mousavi Eshkevari

In this dissertation, we discuss an approach to support declarative reasoning over aspect-oriented programs, where the AspectJ programming language is deployed as a notable (and representative) technology. The approach is based on i) the transformation of source code into a set of facts, and ii) the definition and implementation of relationships and dependencies between different elements of the system into rules, stored in a Prolog database. Declarative analysis allows us to extract complex information through its rich and expressive mechanisms. Our approach has two contributions. First, it can improve the comprehension of AspectJ programs, and it can be deployed for any AspectJ-like language, like e.g. AspectC#, AspectC++. The second contribution is the provision of change impact analysis for AspectJ programs. Our method is automated and tool support is available. Expected beneficiaries of our approach include system maintainers performing tasks during the "change planning" stage of evolution.

Acknowledgments

I would like to express my deep and sincere gratitude to both my supervisors, Dr. Constantinos Constantinides and Dr. Juergen Rilling. Their understanding, encouragement and personal guidance have been instrumental in the writing of this thesis. I warmly thank Venera Arnaoudova, for her valuable advice and friendly help. I owe my loving thanks to my husband Kianoush Torkzadeh. Without his encouragement and understanding, it would have been impossible for me to finish this work. In addition, I would like to thank my parents, who supported and encouraged me during my studies in Concordia University. At last, I would thank all members of Software Maintenance and Evolution Research Group, for providing valuable comments for my research work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software maintenance is defined as "the modification of a software product after delivery to correct faults, improve performance (or other attributes) or to adapt the product to a modified environment." (ANSI/IEEE standard 1219-1998). The importance of maintenance lies on the fact that i) it prolongs the operability of software once it has been deployed and ii) it tends to consume a significant proportion of the overall software life cycle resources (consequently consuming a large part of the costs) [42]. Bennett and Rajlich [10] divide the maintenance phase into three stages: evolution, servicing and phase out. Evolution is defined as the stage where the software is modified[1] to meet the requirements of the stakeholders. Each change is introduced by a request, classified as correction or enhancement. Corrective and preventive activities under maintenance fall under the correction type , whereas adaptive and perfective activities fall under the enhancement type.

[1]Modification implies adding a new requirement or changing an existing one.

Before implementing a change, problem and modification analysis should be performed [6]. Change impact analysis plays an important role in making a decision whether or not a change should be undertaken. In order to analyze the impact of a change, maintainers need to understand the system. Program comprehension constitutes a significant proportion of the maintenance phase [10], particularly when maintainers are not the ones who developed the system. This proportion increases when comprehension is performed on aspect-oriented systems due to their obliviousness property [19]. In this dissertation we focus on comprehension and change impact analysis of aspect-oriented systems.

1.1 Objective and goals of this dissertation

Our initial objective is to provide a mechanism by which maintainers can obtain comprehension of AspectJ programs. A second objective is to allow maintainers to perform change impact analysis of these programs and a third objective is to provide automation through tool support in an integrated development environment.

To meet these objectives, we set a number of goals: Initially to provide a set of rules by which an AspectJ program will be transformed into a set of Prolog facts, the deployment of various types of rules to support comprehension, the deployment of different types of changes to support change impact analysis and the creation of a query system.

1.2 Organization of the dissertation

The remaining part of this dissertation is organized as follows: In Chapter 2 we provide the necessary theoretical background to change impact analysis, Aspect-Oriented Programing (AOP), and declarative reasoning and logic programming. In Chapter 3 we discuss the problem and motivation behind this research. In Chapter 4 we present our proposal and in Chapter 5 we present our methodology, discussing the types of rules defined for comprehension, dependency and change impact analysis. In Chapter 6 we present a case study to demonstrate how our approach can be deployed for a typical exploration task. In Chapter 7 we discuss automation and tool support. In Chapter 8 we discuss related work, and we evaluate our approach. We conclude our discussion in Chapter 9. Furthermore, Appendix A provides the listings of the core functionality, and the aspectual behavior of the case study and its declarative representation in Prolog facts. Appendix B, provides the installation guide for the tool.

Chapter 2

Theoretical background

In this Chapter, we discuss the necessary theoretical background to this research, starting with software maintenance, followed by background on AOP and on declarative reasoning and logic programming.

2.1 Software maintenance

A typical software lifecycle is composed of five sequential stages [39]: Initial development, evolution, servicing, phase-out, and close-down. The software maintenance phase includes all the activities undertaken from evolution to phase-out. In the literature [6, 34] maintenance is defined as the modification of a software product performed after delivery in order to:

Adaptive maintenance: Adapt to a changing/new environment.

Perfective maintenance: Prevent latent faults[1] to become failures[2], and to implement new requirements, to improve software attributes.

Corrective maintenance: Repair known problems.

Preventive maintenance: Prevent latent faults to become operational faults.

Adaptive and perfective types of maintenance are shown in the literature to consume a significantly large proportion of the overall efforts during the maintenance phase. Corrective maintenance is reported to consume a relatively small proportion of the overall maintenance effort. The rest of the effort is consumed while applying preventive maintenance. It is important to note that the different types of maintenance are not mutually exclusive, rather they can be mutually supportive. For example, during adaptive maintenance one can detect an error and then decide to perform corrective maintenance as well.

Software changes occur during both the evolution and servicing stages, and these changes are initiated due to requirement changes, environment changes, bug manifestation, or prevention from activation of a known fault. Bennett and Rajlich define the change cycle in [10] as follows:

1. **Request for change:** Often it is originated from end-users in the form of bug report, or request for additional functionality (referred to as Modification Request, MR, in the IEEE maintenance standard [6]).

[1]A software error which can cause improper functioning of the software [21].
[2]When a software fault activated it becomes a software failure [21].

2. **Planning phase:** Program comprehension and change impact analysis are performed in order to make a decision on whether a change should be undertaken or not. This step corresponds to the problem and modification analysis activity defined in the IEEE maintenance standard [6].

   (a) **Program comprehension:** A substantial component of the planing phase is to locate application domain concepts in the code and to build a mental model of the system. Studies show that this activity consumes more than half of all maintenance resources.

   (b) **Change impact analysis:** Assessing the extend of the change, i.e. components that will be impacted by the change. It indicates how costly the plan is going to be and whether it is to be undertaken at all.

3. **Change implementation:** If the result of previous step is in favor of change, the particular change will be implemented[3].

   (a) **Restructuring for change:** Restructuring might be required to localize the concern before the change takes place. Restructuring must preserve the overall system behavior.

   (b) **Change propagation:** Once a component is modified, it may no longer properly interact with other components, resulting in changes to be made in dependent components.

[3]Other important factors which influence this decision are system quality and business value [45].

4. **Verification and validation:** Activities performed to check for preserving the semantics and correctness of the system after the change implementation.

5. **Redocumentation:** Change cycle ends with updating the system documentation.

Steps 3-5 correspond to the activities of modification implementation and maintenance review/ acceptance defined in the IEEE maintenance standard [6].

The remaining part of this section elaborates on the planning phase of the change.

2.1.1 Program comprehension during maintenance

Comprehension is an activity which has shown to consume a large proportion of the resources during the overall maintenance phase [42], particularly when maintainers are not the initial designers of the system. Furthermore, design artifacts, if at all present, are often incorrect or incomplete. Comprehension methods rely on the study of the dependencies between program (or software) elements. Program slicing [8] and formal concept analysis [17] are examples of such methods. Another method to achieve comprehension is reverse engineering (sometimes the two terms are used interchangeably), transforming the system to a higher level of abstraction [42].

2.1.2 Change impact analysis during maintenance

Software change impact analysis estimates which parts of a software and its related documents will be affected if a proposed change is to be made [8]. It also provides essential information that can be used to estimate the cost of a modification request and to plan its implementation. Arnold and Bohner [8] divide change impact analysis into two major types, namely dependency analysis and traceability analysis.

Dependency analysis: Identifies potential consequences of a change, and estimates modifications needed in order to accomplish a change.

Traceability analysis: Focuses on concern identification from requirements to design artifacts, followed by concern identification from the design artifacts to the source code or vise versa (i.e. the identification starts from the source code and ends with the requirements artifacts).

In this dissertation we are focusing on dependency analysis. Dependency analysis can be performed on source code, models, execution traces, or a combination of them [22]. An example of dependency analysis of source code is to use the combination of data flow and control flow graphs in order to identify program dependencies. Performing dependency analysis on a class diagram is an example of model based dependency analysis [27, 11]. Dependency analysis on execution traces can be used to locate related features in code [16].

2.2 Aspect-Oriented Programming (AOP) and AspectJ

The principle of separation of concerns [36] refers to the realization of system concepts into separate software units and it is a fundamental principle of software development. The associated benefits include better analysis and understanding of systems, improved readability of code, increased level of reusability, easy adaptability and good maintainability. Despite the success of object-orientation in the effort to achieve separation of concerns, certain properties in object-oriented systems cannot be directly mapped in a one-to-one fashion from the problem domain to the solution space, and thus cannot be localized in single modular units. Their implementation ends up cutting across the inheritance hierarchy of the system. Crosscutting concerns (or "aspects") include persistence, authentication, synchronization and contract checking. Aspect-Oriented Programming (AOP) [29] explicitly addresses those concerns by introducing the notion of aspect, which is a modular unit of decomposition. Currently there exist many approaches and technologies to support AOP. One such notable technology is AspectJ [28], a general-purpose aspect-oriented language, which has influenced the design dimensions of several other general-purpose aspect-oriented languages, and has provided the community with a common vocabulary based on its own linguistic constructs. In the AspectJ, an aspect definition is a new unit of modularity providing behavior to be inserted over functional components. This behavior is defined in method-like blocks called advice blocks. However, unlike a method, an advice is never explicitly called. Instead, it is only implicitly invoked by an associated construct called a pointcut expression. A pointcut expression is a predicate over well-defined points in the execution of the program which are referred to as join points. When the program execution reaches a join point captured by a pointcut expression, the associated advice block is executed. Even though the specification and level of granularity of the join point model differ from one language to another, common join points in current language specifications include calls to - and execution of methods and constructors. Most aspect-oriented languages provide a level of granularity which specifies exactly when an advice block should be executed, such as executing before, after, or instead of the code defined at the associated join point. Furthermore, several advice blocks may apply to the same join point. In this case the order of execution is specified by advice precedence rules defined in the underlying language [30].

2.3 Declarative reasoning and logic programming

Declarative languages describe relationships between variables in terms of functions or inference rules and the language executor (interpreter or compiler) applies some fixed algorithm to these relations to produce a result. Declarative programs allows the programmers to integrate logical statements with programming constructs.

Logic programming languages are declarative languages based on first-order predicate logic where data are presented by Horn clauses and where a logic inferencing process is used to produce results. These programs are non-imperative, that is, they do not state how a result is to be computed. Instead, they provide the results based on relevant information and inference rules.

Originally implemented by Alain Colmerauer and Phillipe Roussel at the University of Aix-Marseille in 1971, the logic programming language Prolog was initially deployed for natural-language processing and has become one of the most widely used languages for artificial intelligence. Prolog statements are made of terms which themselves can be constants, variables or compound terms. A compound term, or functor, is represented as `functor(parameter list)`. A functor is like a predicate in predicate calculus and its parameters can be atoms, variables or other functors. A Prolog database consists of two kinds of statements which are statically declared: facts and rules. Facts are propositions that are assumed to be true and they constitute the statements used to construct the hypotheses. Rules are implications between propositions. A problem domain is therefore defined in terms of queries, and goals are addressed by a built-in search mechanism [43].

Chapter 3

Problem and motivation

In this Chapter, we discuss the motivation of this research and define the scope of this dissertation.

Software change impact analysis involves three challenges [7]:

1. Information source volume: The size of software application is getting larger and larger due to increasingly complex requirements of the users.

2. Change semantics: Methods for describing meaningful software change relationships are limited for the range of software artifacts. There exist a limited number of methods defining meaningful software change relationships.

3. Analysis methods: Methods for analyzing the parts of software to be changed are not fully explored.

Change impact analysis requires program comprehension [39]. That is, information obtained through comprehension can be used to ease the impact analysis.

Aspect-Oriented Programing improves the modularity of systems by allowing programmers to reason about individual concerns in relative isolation. However, the improvement of modularity comes with the cost of overall program comprehension[1] and therefore change impact analysis becomes more difficult. This difficulty originates in a new type of dependency introduced by aspect-oriented systems: aspect-to-component dependency. The obliviousness property in general-purpose aspect-oriented languages[2], such as AspectJ, implies that classes have no visibility over aspects and that the aspect-to-component visibility is strictly unidirectional. Implemented this way, we see that given a piece of component functionality `f`, we need to iterate over all aspect definitions to see which pointcuts refer to `f` and which advice blocks may be combined with `f`. Manual analysis can be tedious and error prone, particularly for medium- to large-scale systems. As a result, the implicit interdependency between aspects and classes[3] demands more careful investigation.

To this end, some tool support is currently available. The Eclipse AspectJ plug-in provides some level of visualization. However, there is certain type of knowledge over an AspectJ program which is neither straightforward to obtain nor can be provided through this plug-in. For example the following information can only be manually extracted:

[1]Maintainers faced a similar problem during the shift from procedural to Object-Oriented Programming (OOP).

[2]In [19] Filman and Friedman have argued that obliviousness is a property which must characterize any AOP system.

[3]We have to note that AOP as well as some of the problems discussed here are not bound to OOP.

1. “Fragile aspects” [46]: Aspects containing pointcuts written in a way which makes them highly vulnerable to any changes in the component code.
2. Aspects that have precedence over a given aspect.
3. Aspects that are advising protected methods only.

The motivation of this research is to provide a fine-grained model for the representation of program elements and their inter-dependencies in aspect-oriented programs in order to obtain comprehension and to perform change impact analysis of aspect-oriented systems, and also to automate this process, as well as to provide an environment where the above can be automated.

Chapter 4

Proposal

In this Chapter, we discuss our research proposal.

4.1 Declarative reasoning of aspect-oriented programs

We propose the adoption of declarative analysis to achieve comprehension and change impact analysis of aspect-oriented programs by deploying AspectJ as a notable representative example of a general-purpose aspect-oriented language. To achieve this goal, we need to perform a transformation from source code to a declarative representation. To obtain comprehension, we plan to adopt strategies from the literature. These strategies will be translated as rules in a Prolog database. In order to perform change impact analysis, we first start with identifying dependencies in an aspect-oriented system, and then we codify them as Prolog rules. Some of these dependencies are adopted from the literature [56], while some others will be introduced. In the same manner, these dependencies will be translated into rules in a Prolog database. We will adopt change types defined in the literature, while refining some and introducing a new one. Comprehension can then be obtained by executing queries on the Prolog database. End-user can perform change impact analysis by querying the database about what parts of the system would be affected should a specific change occur (see Figure 1).

4.2 Why Prolog?

The Prolog language has its foundation in logic programming which allows programmers to define solutions to problems in a logical manner. Its built-in pattern-matching mechanism (unification) makes it possible to bound variables to complex structures which can themselves contain other variables. Moreover, unification provides a mechanism to find multiple solutions for a given problem. In addition to the above, Prolog can be deployed as a query language for a database of simple facts for matching complicated patterns. We believe that Prolog is more suitable than other query languages (e.g the Standard Query Language - SQL) for our approach since our database would contain simple facts, but a lot of complex search rules. For example, the derivation rules of Prolog enable us to define relations between facts. However, with SQL we would need to store facts for each relation (views in relational database) and we cannot build views recursively [31]. Deploying SQL would be more beneficial with a great amount of very complex data and with relatively simple search rules. Prolog makes it relatively straightforward to specify, execute and refine complex queries over facts.

4.3 Expected contributions and benefits

The expected contribution of this proposal is to provide a proof of concept for an automated environment under which one can obtain knowledge over an AspectJ-like program where this knowledge would otherwise have been difficult or impossible to obtain through existing techniques, as well as to perform change impact analysis.

Potential beneficiaries of this approach include system maintainers who can perform change impact analysis by querying the fact-base on what elements of the system would be affected should a specific change occur.

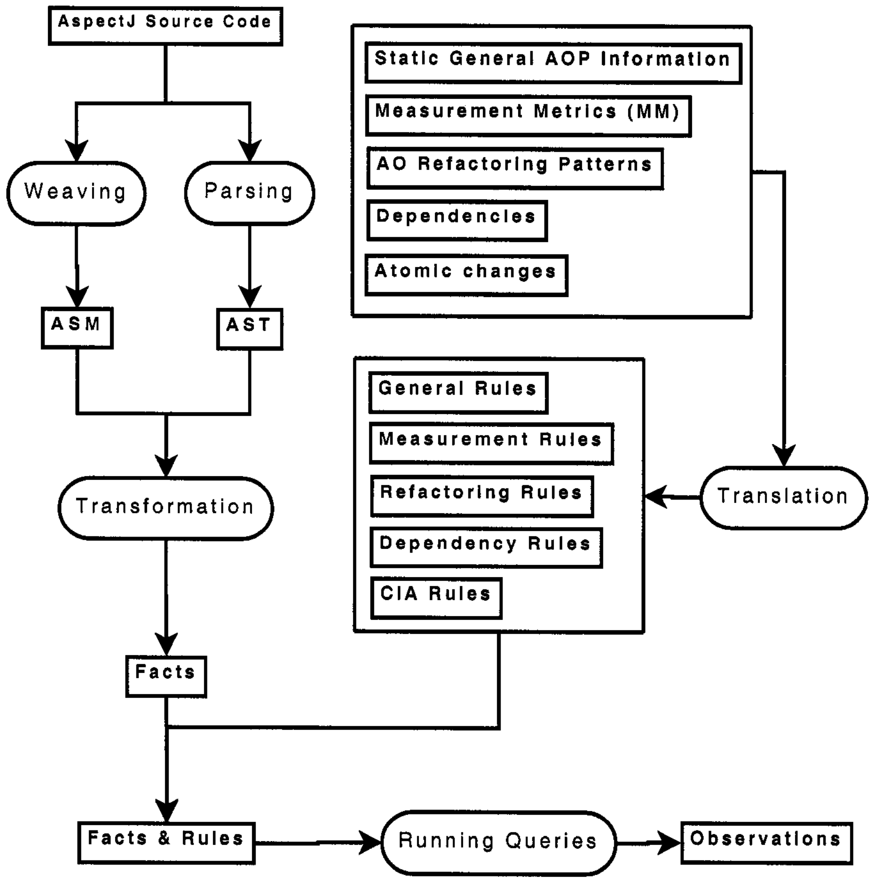


Figure 1: UML activity diagram illustrating our approach.

Chapter 5

Methodology

In this Chapter, we discuss the methodology of our research proposal.

5.1 Model transformation

In order to transform an AspectJ program into a set of Prolog facts, we have defined a set of transformation rules given in Tables 3 and 4. We restrict this approach to method/constructor call and execution. That is, we do not consider object initialization neither attribute accessor/modifier join points during our analysis. However, they are being transformed to a set of Prolog facts for the future adaptation of our sysetm.

The transformation process from source code to facts is broken down into two steps. First, the Abstract Syntax Tree (AST) corresponding to each compilation unit of the program (.java and .aj files) is retrieved and traversed. Second, the

AspectJ Structure Model (ASM) is retrieved and traversed to provide additional information regarding the relationship among pointcuts, methods and advice blocks. The extracted information from these steps is then translated into Prolog facts according to the transformation rules and added to the fact-base and used during the inference process. One such example is shown in Table 1.

```
[introducedMethod](<aspectName>,<typeName>,<methodName>,<visibility>,<returnType>,
                   <listOfParameters>,<methodType>)
```

Table 1: Example of a Prolog fact format.

The above fact reads as follows: `aspectName` introduces a method called `methodName` with the specified `visibility` and `returnType` and `listOfParameters` for the type `typeName`. The last argument in the facts is a number that can be 1, 2, 3, or 0 for abstract methods, final methods, static methods, or non of the previous type respectively. Table 2 shows an example for a transformation rule[1].

5.2 Model comprehension

We have added a set of rules to the database in order to capture relationships between software elements in the system. These rules are context-free, i.e. they are independent from the particular applications in which they are being deployed. The rules are categorized into three types, based on the motivation by which they were created:

[1]This example is taken from the case study discussed in detail in Chapter 6.

```
public aspect CoordinateObserver extends ObserverProtocol {
        //some code
        public void Retailer.notifyOfchange(Subject s,PotentialOrder po){
         //some code
         } // some code
        }


aspect(coordinateobserver,public).
privilegedAspect(coordinateobserver).
extends(coordinateobserver,observerprotocol).
introducedMethod(coordinateobserver,retailer,notifyOfchange,public,void,
               [subject, potentialorder],0).
```

Table 2: Partial code of aspect `CoordinateObserver` and its transformation to Prolog facts.

1. Addressing problems of "bad smells" in code, being the motivation behind refactoring [20, 35] strategies discussed in the literature in the context of object-oriented and aspect-oriented programs.

2. Addressing measurements of program quality through the deployment of metrics.

3. Addressing general issues over a program's static structure (information related to the inheritance hierarchy, the interaction among the entities of the system, etc.).

5.2.1 Rules to identify bad smells

Rules to identify potential bad smells (identifying anomalies where refactoring may be required) are influenced by aspect-oriented refactoring strategies such as those

Transformation rules	Definition
<visibility>:= <public> | <private> | <protected> | <package>	visibility of a feature is <public> or <private> or <protected>
[class](<className>, <visibility>)	class_name is a class with <visibility>
[finalClass](<className>)	className is a final class
[abstractClass](<className>)	className is an abstract class
[interface](<interfaceName>, <visibility>)	interfaceName is an interface with <visibility>
[extends](<subClassName> (| <subInterfaceName> | <subAspectName>),<superClassName> (| <superInterfaceName> | <superAspectName>))	Class subClassName (or subInterfaceName or subAspectName) extends superClassName (or superInterfaceName or superAspectName)
[implements](<className>, <interfaceName>)	Class class_name implements interface interface_name
[aspect](<aspectName>, <visibility>)	aspectName is an aspect with <visibility>
[privilegedAspect](<aspectName>)	aspectName is a privileged aspect
[new](<className1>, <methodName1>, <className2>)	An instance of <className2> is instantiated in method <methodName1> in <className1>
<attributeType>:= 0 | 1 | 2 | 3	<attributeType> can be final (1), or static (2), or final-static (3), and if it is not any of the previous types it is marked 0
[attribute](<className> |<aspectName> | <interfaceName>, <attName>, <type>, <visibility>, <attributeType>)	<attName> of type <type> is an attribute declared in class className or aspect aspectName or interfaceName with <visibility>
<methodType>:= 0 | 1 | 2 | 3	<methodType> can be abstract(1), or static (3), or final (2) and if it is not any of the previous types it is marked 0
[method](<className> | <aspectName> | <interfaceName>, <methodName>, <visibility>, <type>, <listOfParameters>, <methodType>)	<methodName> , with <listOfParameters> parameters and access modifier public or private or protected and return type <type> is declared in <className> or <aspectName> or <interfaceName>
[sendMessage](<className1>, <methodName1>, <listOfParameters1>, <className2>, <methodName2>, <listOfParameters2>)	A message methodName2 is sent to <className2> from methodName1 in <className1>
[used](<aspectName>, <adviceId>, <listOfParameters1>, <className>, <methodName>, <listOfParameters>)	<methodName> in class <className> is invoked by <adviceId> in <aspectName>
[accessFeature](<className1>, <methodName1>, <className2>, <instanceVariableName2>)	<className1> accesses <instanceVariableName2> of <className2> from <methodName1>
[constructor](<className>, <visibility>, <listOfParameters>)	<className> has a constructor with <listOfParameters> parameters
[declareParent](<aspectName>,<type1>, <type2>)	<type2>is declared to be the supertype of <type1> in <aspectName>
[introducedMethod](<aspectName>,<typeName>, <methodName>,<visibility>, <returnType>, <listOfParameters>,<methodType>)	<aspectName> declares a method <methodName> with <visibility>, <returnType>,and <listOfParameters> for class <typeName>
[introducedAtt](<aspectName>,<typeName>, <attName>, <type>, <visibility>, attributeType>)	<aspectName> declares an attribute <attName> with, <type> for a type <typeName>
[pointcutdesig](<pointcutDesignatorId>,<aspectName>, <pointcutName>,<joinpoint>,<listOfParameters>)	<joinpoint> with a unique id <pointcutDesignatorId> defined in aspectName

Table 3: Transformation rules - Part I.

Transformation rules	Definition
`<pointcutType>:= 0 |1 |2`	`<pointcutType>` can be `abstract` (1), `static` (2), and if it is not any of the previous types it is marked 0
`[pointcut](<aspectName>,<pointcutName>,<listOfParameters>, <visibility>,<pointcutType>)`	`<pointcutName>` defined in `<aspectName>`
`<joinpoint>:= call| execution| target| args| this`	`<joinpoint>` can be `call`, `execution`, `target`, `args`, or `this`
`<adviceType>:= before, after, around`	`<adviceType>` is `before` , `after` or `around`
`[triggerAdvice](<aspectName>, <adviceType>, <adviceId>,<listOfParameters>, <returnType> )`	Advice `<adviceType>` belongs to `<aspectName>` aspect
`[advicePointcutMap](<aspectName>, <adviceType>, <adviceId>, <pointcutName> )`	Advice `<adviceType>` defined in `<aspectName>` aspect is related to the pointcut `<pointcutName>`
`[precedence](<aspectName>, <listOfAspects>)`	`<precedence>` rule is defined in aspect `<aspectName>`, and `<listOfAspects>` contains list of aspects according to their precedence
`[advisedBy](<typeName>,<methodName>,<listOfParameters1>, <aspectName>,<adviceType>,<adviceId>,<listOfParameters2>, <pointcutName>)`	`<methodName>` of `<typeName>` with the `<listOfParameters1>` is advised by `<adviceType>` with the `<listOfParameters2>` in `<aspectName>`

Table 4: Transformation rules - Part II.

discussed in [35] where the authors describe typical situations in aspect-oriented programs which can be problematic along with recommended refactoring strategies. In this work we are only interested in the identification of bad smells as they can provide indications to maintainers where refactoring could perhaps be valuable.

Split abstract class into aspect and interface

Problem: *Classes are prevented from using inheritance because they already inherit from an abstract class defining some concrete members* [35].

A candidate to be splitted into an aspect and an interface, is an abstract class which have concrete and abstract methods and which is subclassified. Table 5 shows the corresponding Prolog rule defined to identify such a bad smell. First, one has to check if a given class `ClassName` is an abstract class which is also subclassified. Next, one has to verify if this class contains abstract and concrete methods:

```
is_CandidateForAspect(ClassName):-findall(SuperType,extends(_,SuperType),List),
                                  (select(ClassName,List,Rest),
                                   (is_abstract(ClassName),
                                    has_abstractMethod(ClassName),
                                    has_concreteMethod(ClassName))).
```

Table 5: Finding a class to be refactored as an aspect.

Inline class within aspect

Problem: *A small standalone class is used only by code within an aspect* [35].

A candidate to be inlined within an aspect is a class which is not subclassified, which is not used as an attribute of other classes, which does not receive messages from other classes, and which is referenced in an advice body of only one aspect. Table 6 shows the Prolog rule defined to identify such a bad smell.

```
is_CandidateForInline(Type):-is_class(Type),
                             (get_descendants(Type,L),size(L,0)),
                              not(attribute(_,Type,_,_,_)),
                              not(sendMessage(_,_,_,Type,_,_)),
                             (findall(Aspect,(is_aspect(Aspect),
                                              used(Aspect,_,_,Type,_,_)),List),
                              (size(List,1))).
```

Table 6: Finding a class to be moved into an aspect.

Along the same lines, one can define rules detecting bad smells for the following strategies discussed in [35]: *Replace Implements with Declare Parents*, *Introduce Aspect Protection*, *Replace Inter-type Method with Aspect Method*, *Extract*

Superaspect, *Pull Up Advice*, *Pull Up Declare Parents*, *Pull Up Inter-type Declaration*, *Pull Up Pointcut* and *Push Down Pointcut*.

5.2.2 Rules to deploy measurements

We have defined measurement rules in order to extract information on the quality and the complexity of the program. Often the complexity of a system depends on a number of measurable attributes such as inheritance, coupling, cohesion, polymorphism, and application size. Some of these attributes like coupling and cohesion are also applicable in an aspect-oriented context. In [56] the author defines coupling as the degree of interdependency among aspects and/or classes. The following measurement rules are based on some of the metrics presented in [56].

Attribute-Class dependence

There is an attribute-class dependence between aspect a and class c, if c is a type of an attribute of a. The number of attribute class dependences from aspect a to the class c can formally be represented as $AtC(a, c) = | \{x | x \in A^a(a) \wedge T(x) = c\} |$. This factor can be calculated with the rule in Table 7. The larger the number is (in the above metrics), the stronger the dependency is. However, for a better analysis this factor needs to be calculated for all aspects and classes, and then the the degree of dependency can be inferred by comparing the data.

```
attributeClassDependenceCount(AspectName,TypeName,Count):-
                    is_aspect(AspectName), is_type(TypeName),
                    findall(TypeName,(attribute(AspectName,_,Atype,_,_),
                                  ((Atype=TypeName);
                                    (superType(Atype,TypeName)))),
                          List),
                    size(List,Count).
```

Table 7: Calculating `attribute-class dependence` factor for an aspect.

Pointcut-Class dependence

Let p be a pointcut of aspect a. There is a pointcut-class dependence between a and c, if c is the type of a parameter of p. The number of pointcut-class dependencies from aspect a to class c can formally be represented as:

$$PC(a,c) = \sum_{p \in P(a)} |\{x | x \in Par(p) \wedge T(x) = c\}|$$

Table 8 shows the corresponding Prolog rule for calculating the above measurement. The larger the number is (in the above metrics), the stronger the dependency is. However, for a better analysis this factor needs to be calculated for all pointcuts and classes, and then the the degree of dependency can be inferred by comparing the data.

Along the same lines, one can define rules calculating the metrics discussed in [56] as: *Advice-Class Dependence*, *Intertype-Class Dependence*, and *Aspect-Inheritance Dependence* measures.

```
pointcutClassDependence(AspectName,TypeName,Count):-
          (is_aspect(AspectName),(is_class(TypeName);is_interface(TypeName))),
           count((pointcut(AspectName,_,List,_,_),(member(TypeName,List);
                 (superType(SuperType,TypeName),member(SuperType,List)))),
                Count).


pointcutClassDependence(AspectName,TypeName,Count):-
          (is_aspect(AspectName),(is_class(TypeName);is_interface(TypeName))),
          (Count is 0).

pointcutClassDependenceThroughInheritence(SubAspectName,TypeName,Count):-
          ((is_aspect(SubAspectName),is_aspect(SuperAspectName),
            superType(SuperAspectName,SubAspectName)),
          (is_class(TypeName);is_interface(TypeName))),
          count(((pointcut(SuperAspectName,_,List,_,PointcutType),
                 not(PointcutType=1)),((member(TypeName,List));
                 ((superType(SuperType,TypeName),member(SuperType,List))))),
               Count) .

pointcutClassDependenceThroughInheritence(SubAspectName,TypeName,Count):-
          is_aspect(SubAspectName),(Count is 0).


pointcutClassDependenceCount(AspectName,TypeName,TotalCount):-
        pointcutClassDependence(AspectName,TypeName,Count1),
        pointcutClassDependenceThroughInheritence(AspectName,TypeName,Count2),
        (TotalCount is Count1+Count2).
```

Table 8: Calculating `pointcut-class dependence` factor for an aspect.

5.2.3 General rules

General rules are built in order to extract knowledge about the static structure of a program, like inheritance relationships, dependencies between aspects and classes, etc. This category defines the core rules in our system, and constitutes the basis for change impact analysis. Examples of general rules are : Aspect monitoring features of a class (methods, attributes) with specific modifiers, aspects with precedence over a specific aspect, methods advised by a pointcut, messages to which a class responds.

We illustrate three detailed examples of general rules below.

Declared method in inheritance hierarchy

In Table 9, rule `findDeclaredMethod` identifies all the methods that are defined by an aspect `AspectName` (through the inter type declaration) for all the supertypes of a given type.

```
findDeclaredMethod(AspectName,TypeName,SuperTypeName,MethodName):-
                is_aspect(AspectName),superType(SuperTypeName,TypeName),
                introducedMethod(AspectName,SuperTypeName,MethodName,_,_,_).
```

Table 9: Obtaining methods defined by an aspect for a supertype of a given type.

Messages to which a class responds

There are situations where we want to identify the messages to which an instance can respond. In an object-oriented system an object can respond to a message if it has or inherits a method definition (with access modifier public or protected) with type signature corresponding to this message. However, in an aspect-oriented system inter-type declaration allows to introduce methods or attributes for a class or an interface. This implies that an object o of type T, can also respond to messages introduced by aspects for T, or for supertypes of T. Having a Prolog rule to enforce that an introduced method is itself a method can help to resolve the problem in case of introduction. Table 10 shows the Prolog rule corresponding to this definition. Rule `respondTo` simulates part of the introspective capabilities provided by languages like Smalltalk [23] and Ruby [12].

```
findMessages(Type,ListOfMessagesWithParam):-
            findall(Method,is_declaredMethod(Type,Method), ListOfMessages),
            (getMember(MethodName,ListOfMessages),
             method(Type,MethodName,Visibility,_,ListOfParameters,_),
              \+ Visibility= private,
             append([MethodName],[ListOfParameters],ListOfMessagesWithParam)).

findMessages(Type,ListOfMessagesWithParam):-
            findall(Method,is_inheritedMethod(Type,Method), ListOfMessages),
            (getMember(MethodName,ListOfMessages),
             method(Atype,MethodName,_,_,ListOfParameters,_),
             superType(Atype,Type),
             append([MethodName],[ListOfParameters],ListOfMessagesWithParam)).

findMessages(Type,ListOfMessagesWithParam):-
            findall(Method,(introducedMethod(_,SuperType,Method,_,_,_,_),
                            superInterfaceOfType(SuperType,Type)),
                    ListOfMessages),
            (getMember(MethodName,ListOfMessages),
             method(SuperType,MethodName,Visibility,_,ListOfParameters,_),
              \+ Visibility= private,
             append([Methodname],[ListOfParameters],ListOfMessagesWithParam)).

respondTo(Type,List):-is_type(Type),
                      findall(ListOfMessagesWithParam,
                              findMessages(Type,ListOfMessagesWithParam),
                              List).
```

Table 10: Obtaining all messages to which an object of a given type can respond.

Method monitored by aspects

By the advice construct, aspects can inject functionality to be executed before, after or instead of a method in a class. In this case, the method is said to be monitored by the aspect. In Table 11, rule `aspectMonitoringMethod` identifies the list of aspects monitoring a given method of a given type.

```
aspectMonitoringMethod(TypeName,MethodName,ListOfAspects):-
method(TypeName,MethodName,_,_,_,_),
findall(AspectName,(advisedBy(TypeName,MethodName,_,AspectName,_,_,_,_)),ListOfAspects),
\+ ListOfAspects=[].
```

Table 11: Obtaining all aspects monitoring a given method in a given type.

5.3 Change impact analysis

In this section, we define rules for dependency and change impact analysis.

5.3.1 Dependency relationships between program entities

We have classified dependencies in aspect-oriented systems into three groups:

- Dependencies among aspects (AO specific dependencies).
- Dependencies among classes (OO specific dependencies).
- Dependencies among classes and aspects (OO-AO dependencies).

The presence of these dependencies makes the modification (add/ delete/ change) of the program very difficult. Maintainers of aspect-oriented systems need to consider not only dependencies that exist in aspect-oriented and/or object-oriented parts, but also their inter-dependencies. Some changes in the system lead to errors detected at compile time (for example the deletion of a method definition will lead to compilation errors where this method is called). We exclude these types of changes from our investigation because the programmers will be alerted

Types of dependencies
Inheritance dependency
Precedent dependency
Pointcut-Parameter dependency
Pointcut-Method dependency
Advice-Type dependency
Inter-type dependency
Advice-Method dependency
Pointcut-designator dependency

Table 12: Catalog of dependencies.

by the compiler on the specific points in the system affected by this change. We also exclude the changes that are OO specific since they are already discussed in the literature [40]. Our investigation targets AO and OO-AO dependencies and changes that may cause the system to behave unintentionally as a result of a modification which has affected parts of the system without alerting the programmers, that no compilation error has been produced[2].

In the following subsection we identify different types of dependencies (see Table 12) and for each one we provide a rule to detect them, and then codified them in our database.

Types of AO specific dependencies

In this subsection we discuss types of AO specific dependencies.

[2]It is important to note that we do not identify unintentional behavior. Instead, we identify all parts of the system which are affected by a specific change, in order to allow the user to decide whether or not this impact is intentional.

Inheritance dependency

This defines the dependency between a superaspect and its subaspects. In AspectJ, an aspect cannot be extended unless it is declared to be abstract. An abstract aspect needs to have abstract pointcuts which will then be implemented by concrete subaspects. Normally the advice blocks related to the abstract pointcuts are defined in the superaspect. Detecting the impact of superaspect deletion would not be particularly interesting because this is immediately caught by the compiler. However, it is possible that one would delete the content of the superaspect. In the example of Figure 2, there is a direct dependency between the `before` advice of the `Superaspect` and the abstract pointcut `p` defined in the `Superaspect` (and also to the concrete pointcut `p` defined in `Subaspect`) as the advice knows which pointcut it is related (bound) to. Therefore, deleting the abstract pointcut would lead to a compilation error. On the other hand, a pointcut does not know about the advice blocks which depend on it. This implies that deleting the advice blocks (related to the abstract pointcut) in the superaspect would result in the program loosing the expected functionality (which was supposed to be supported by `before`, `after`, or `around` of the join point match). Therefore the intended behavior of the program will be changed if this dependency is not identified prior to the deletion of advice blocks. This dependency can be detected through the rule in Table 13.

Precedence dependency

Multiple advice blocks may apply to the same pointcut, and the resolution order

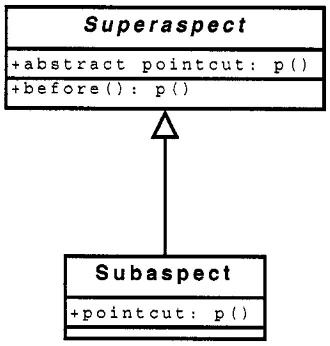


Figure 2: Inheritance dependency.

```
advicePointcutInheritenceDep(SuperAspect,SubAspect,AdviceId,AdviceType,PointcutName):-
                        is_aspect(SuperAspect),is_aspect(SubAspect),
                        superAspect(SuperAspect,SubAspect),
                        triggerAdvice(SuperAspect,AdviceType,AdviceId,_,_),
                        pointcut(SubAspect,PointcutName,_,_,0),
                        pointcut(SuperAspect,PointcutName,_,_,1),
                        advicePointcutMap(SuperAspect,AdviceType,AdviceId,PointcutName).
```

Table 13: Obtaining the advice blocks defined in a superaspect while bound to a given pointcut of a subaspect.

of the advice is based on predefined rules on advice precedence [50] which can be categorized into two types:

1. Precedence rules among advice blocks from different aspects.

2. Precedence rules among advice blocks within the same aspect.

The precedence rules create a dependency between advice blocks related to the same pointcut as the execution of one advice will depend on the advice which executes before it. The example in Table 14 corresponds to the first type. The three advice blocks defined in aspects `AJ1` and `AJ2` are applied to the same join point `call(public void C.m(..)`. According to the precedence rules the `before` advice defined in aspect AJ1 has precedence over the two advice blocks defined in aspect AJ2, and the `before` advice of AJ2 has precedence over its `after` advice. The output in Table 14 shows the execution order of method `C.m(int)` and the advice blocks. Neither of the advice blocks are aware of the precedence defined in aspect AJ2. This implies that there would be no indication about this dependency if one wants to change the `before` advice, of any of the aspects, to `after` or `around` advice. Another type of change can be adding a new advice block for the same join point in aspect AJ1 or deleting either of the advice blocks.

```
public aspect AJ1 {
        pointcut AJ1_P1(): call(public void C.m(..));
        before(): AJ1_P1() { // Display "Before from AJ1"}}

public aspect AJ2 {
        declare precedence: AJ1, AJ2;
        pointcut AJ2_P1(): call(public void C.m(..));
        before(): AJ2_P1() { // Display "Before from AJ2"
        }
        after(): AJ2_P1() { // Display "After from AJ2"
        }}

public class C {
        public void m(int i){
                ... }}

Output:
Before from AJ1
Before from AJ2
 ...
After from AJ2
```

Table 14: Demonstrating precedence dependency.

For certain applications, the correct order of advice and method execution is vital to preserve the semantics of the system. One such example is a system providing a service to concurrent (multiple) clients where a precondition to a service would dictate that authentication would have to be evaluated before synchronization which in turn would have to be evaluated before scheduling. Precedence rules guarantee the correct order, but any changes to the precedence or to the advice should be performed with attention to the dependency that the `declare precedence` creates. We can detect the precedence dependency through the following strategy:

Precedence dependency between advice blocks of the same aspect

For each pointcut defined in an aspect, we need to identify a list of its related advice blocks. If the list contains more than one advice, then according to the precedence rules there would be an order of execution for these advice blocks which implies a dependency (Table 15).

```
advicePrecedenceDepPerAspect(AspectName,PointcutName,ListofAdviceBlocks):-
   is_aspect(AspectName),
   pointcut(AspectName,PointcutName,_,_,_),
   findall(AdviceType,(advicePointcutMap(AspectName,AdviceType,AdviceId,PointcutName),
                  triggerAdvice(AspectName,AdviceType,AdviceId,_,_)),
         ListofAdviceBlocks),
   size(ListofAdviceBlocks,N),N>1.
```

Table 15: Obtaining the list of advice blocks bound to a given pointcut.

Precedence dependency between advice blocks of different aspects

First we need to identify aspects which are listed in `declare precedence` statements. Then for all these aspects, we need to find the list of advice blocks which are advising the same methods in the system. Table 16 shows the Prolog implementation of rule `advicePrecedenceDepForMultipleAspects` to identify list of advice blocks related to a given advice in given aspect.

```
advicePrecedenceDepForMultipleAspects(AspectName,AdviceType,ListOfAdviceParameters,
                                      ListofAdviceIds):–
precedence(_,List),member(AspectName,List),getMember(AnAspect,List),
triggerAdvice(AspectName,AdviceType,_,ListOfAdviceParameters,_).
findall(AdviceId,advisedBy(ClassName,MethodName,ListOfParam,AnAspect,AdviceType,
                           AdviceId,_,_),
        ListOfAdviceIds)).
```

Table 16: Obtaining all advice blocks related to a given advice in a given aspect.

Types of OO-AO dependencies

In this subsection, we discuss types of OO-AO dependencies.

Pointcut-Parameter dependency

Pointcuts can expose part of the execution context of the captured join points, and the values exposed by a pointcut can be used in the body of advice declarations [50]. A context is like a parameter list of a method that is, a list of types (either user-defined or built-in)/identifier pairs. The fact that a pointcut (which has parameter(s)) knows about its parameters indicates a dependency. This means that pointcuts are dependent on the context they expose. According to the inheritance mechanism, an object of a type holds an "is-a" relationship with its supertype, indicating that a pointcut is also dependent on the subtypes of the types, included in its parameter list. Table 17 shows the Prolog rule corresponding to this dependency.

```
pointcutParameterDep(AspectName,PointcutName,ListOfDepTypes):-
    is_aspect(AspectName),pointcut(AspectName,PointcutName,ListOfParameters,_,_),
    findall(TypeName,(getMember(TypeName,ListOfParameters),is_type(TypeName)),
            ListOfDeclaredTypes),
    findall(TypeName,(getMember(X,ListOfParameters),(X = object),(is_class(TypeName);
                    is_interface(TypeName))),ListOfTypes),
    findall(SubType,(getMember(SuperType,ListOfDeclaredTypes),subType(SubType,SuperType)),
            ListOfSubTypes),
    append(ListOfTypes,ListOfDeclaredTypes,ListOfAllTypes),
    append(ListOfAllTypes,ListOfSubTypes,ListOfDepTypes).
```

Table 17: Obtaining the list of user-defined types being members of the parameter list of a given pointcut.

Pointcut-Method dependency

A pointcut expression is built from the composition of the basic pointcut designators [14]. A pointcut monitors a method if it occurs in one of its pointcut designators, and therefore there exists a dependency between the pointcut and this method. A pointcut may be deactivated if the method signature changes in the base code. In the same manner, a change in the signature of a monitored method in pointcut designator may cause a pointcut to monitor other method(s) or nothing at all. Therefore, the dependency between pointcuts and methods needs to be captured. Here we investigate the dependency in the following four cases and we exclude the case where (..) wildcard is used as method parameter in all these cases.

Name of type and method are clearly defined in pointcut designator

In Table 18, rule `pointcutTypeMethodDep` is defined for cases where no wildcard exists in the pointcut designator. The second rule in this figure is defined for cases when there exists an inheritance relationship between aspects.

```
pointcutTypeMethodDep(AspectName,PointcutName,TypeName,MethodName,
                      ListOfMethodParameters):-
            (is_aspect(AspectName),
             pointcutdesig(_,AspectName,PointcutName,_,[_,_,TypeName,MethodName,
                                                     ListOfMethodParameters]),
             (MethodName\= every),(ListOfMethodParameters\= any),(TypeName \= every)),
            method(TypeName,MethodName,_,_,ListOfMethodParameters,_).

pointcutTypeMethodDep(SuperAspectName,PointcutName,TypeName,MethodName,
                      ListOfMethodParameters):-
            (is_aspect(SubAspectName),is_aspect(SuperAspectName),
             superAspect(SuperAspectName,SubAspectName),
             pointcut(SuperAspectName,PointcutName,_,_,1),
             pointcut(SubAspectName,PointcutName,_,_,0),
             pointcutdesig(_,SubAspectName,PointcutName,_,[_,_,TypeName,MethodName,
                                                        ListOfMethodParameters]),
             (MethodName\= every),(ListOfMethodParameters\= any),(TypeName \= every)),
            method(TypeName,MethodName,_,_,ListOfMethodParameters,_).
```

Table 18: Obtaining the pointcuts monitoring a given method of a given type.

Name of type is * (asterisk) and name of method is clearly defined

The Prolog rule `pointcutMethodDep` in Table 19 is defined for cases where the * (asterisk) wildcard is used as type name in pointcut designator. The second rule in this figure is defined for the situation where there exists inheritance relationship between aspects. During the transformation process from the source code to Prolog facts, all the * wildcards are replaced by `every` atom.

```
pointcutMethodDep(AspectName,PointcutName,ListOfTypes,MethodName,
                        ListOfMethodParameters):-
                       (is_aspect(AspectName),
                        pointcutdesig(_,AspectName,PointcutName,_,[_,_,every,MethodName,
                                                                        ListOfMethodParameters]),
                        (MethodName\= every),(ListOfMethodParameters\= any)),
                         findall(TypeName,method(TypeName,MethodName,_,_,
                                                    ListOfMethodParameters,_),ListOfTypes).

pointcutMethodDep(SuperAspectName,PointcutName,ListOfTypes,MethodName,
                        ListOfMethodParameters):-
                       (is_aspect(SubAspectName),is_aspect(SuperAspectName),
                        superAspect(SuperAspectName,SubAspectName),
                        pointcut(SuperAspectName,PointcutName,_,_,1),
                        pointcut(SubAspectName,PointcutName,_,_,0),
                        pointcutdesig(_,SubAspectName,PointcutName,_,[_,_,every,
                                       MethodName,ListOfMethodParameters]),
                        (MethodName\= every),(ListOfMethodParameters\= any)),
                         findall(TypeName, method(TypeName,MethodName,_,_,
                                                    ListOfMethodParameters,_),ListOfTypes).
```

Table 19: Obtaining the pointcuts monitoring a given method.

Name of type is clearly defined and name of method is *

The Prolog rule `pointcutTypeDep` in Table 20 is defined for cases where the * wildcard is used instead of a specific method name in a pointcut designator. The second rule in this table is defined for the situation where there exists inheritance relationship between aspects.

Name of type and method is *

The Prolog rule `pointcutDep` in Table 21 is defined for cases where the * wildcard is used instead of a specific method name and the type in which it is defined in a pointcut designator. The second rule in this table is defined for the situation where there exists inheritance relationship between aspects.

```
pointcutTypeDep(AspectName,PointcutName,TypeName,ListOfMethods,
           ListOfMethodParameters):-
        (is_aspect(AspectName),
         pointcutdesig(_,AspectName,PointcutName,_,[_,_,TypeName,every,
                                             ListOfMethodParameters]),
        (TypeName\= every),(ListOfMethodParameters\= any)),
        findall(MethodName, method(TypeName,MethodName,_,_,
                              ListOfMethodParameters,_),ListOfMethods).

pointcutTypeDep(SuperAspectName,PointcutName,TypeName,ListOfMethods,
           ListOfMethodParameters):-
        (is_aspect(SubAspectName),is_aspect(SuperAspectName),
         superAspect(SuperAspectName,SubAspectName),
         pointcut(SuperAspectName,PointcutName,_,_,1),
         pointcut(SubAspectName,PointcutName,_,_,0),
         pointcutdesig(_,SubAspectName,PointcutName,_,[_,_,TypeName,every,
                                             ListOfMethodParameters]),
         (TypeName\= every),(ListOfMethodParameters\= any)),
        findall(MethodName,method(TypeName,MethodName,_,_,
                              ListOfMethodParameters,_),ListOfMethods).
```

Table 20: Obtaining the pointcuts monitoring all methods of a given type.

Advice-Type dependency

An advice declaration has a parameter list (like a method) that explicitly names all the context that are passed to it. In case the context is a non primitive type, an advice can have access to its features. This creates a dependency between the advice and the types included in advice parameter declaration. It is important to note that, a subaspect inherits all the advice blocks defined in superaspect. As a consequence, an indirect dependency will be created between the subaspect and the types in parameter list of superaspect. Table 22 shows the corresponding Prolog rule identifying this dependency.

```
pointcutDep(AspectName,PointcutName,ListOfTypeMethods,ListOfMethodParameters):-
    (is_aspect(AspectName),
     pointcutdesig(_,AspectName,PointcutName,_,[_,_,every,every,ListOfMethodParameters]),
     (ListOfMethodParameters\= any)),findall(Type,is_type(Type),ListOfAllTypes),
    (getMember(X,ListOfAllTypes),method(X,MethodName,_,_,ListOfMethodParameters,_),
     append([X],[MethodName],ListOfTypeMethods)).

pointcutDep(SuperAspectName,PointcutName,ListOfTypeMethods,ListOfMethodParameters):-
    (is_aspect(SubAspectName),is_aspect(SuperAspectName),
     superAspect(SuperAspectName,SubAspectName),
     pointcut(SuperAspectName,PointcutName,_,_,1),
     pointcut(SubAspectName,PointcutName,_,_,0),
     pointcutdesig(_,SubAspectName,PointcutName,_,[_,_,every,every,ListOfMethodParameters]),
     (ListOfMethodParameters\= any)),findall(Type,is_type(Type),ListOfAllTypes),
    (getMember(X,ListOfAllTypes),method(X,MethodName,_,_,ListOfMethodParameters,_),
     append([X],[MethodName],ListOfTypeMethods)).
```

Table 21: Obtaining the pointcuts monitoring all methods in the system.

Advice-Method dependency

Advice blocks are method-like constructs that provide additional behavior to be executed before, after, or instead of the methods to which they are woven [28]. This creates a dependency between the advice and the methods being advised by it. It is important to note that an advice definition does not have any information about the methods it advises. An advice only knows the pointcut to which it is bound. The pointcut creates a link between the advice blocks which are bound to it and the methods it advises. Therefore, it is difficult to manually identify the dependency between an advice and methods. Table 23 shows the Prolog rule defined in order to identify this dependency `adviceMethodDep`.

Inter-type dependency

There is an inter-type dependency between a class (or interface) and an aspect if

```
adviceTypeDep(AspectName,AdviceType,AdviceId,ListOfDepTypes):-
    triggerAdvice(AspectName,AdviceType,AdviceId,ListOfAdviceParameters,_),
    findall(Type,(getMember(Type,ListOfAdviceParameters),is_type(Type)),
            ListOfDeclaredTypes),
    findall(Type,(getMember(X,ListOfAdviceParameters),(X = object),(is_class(Type);
                  is_interface(Type))),ListOfTypes),
    findall(Subtype,(getMember(SuperType,ListOfAdviceParameters),is_type(SuperType),
                     subType(Subtype,SuperType)),ListOfSubTypes),
    append(ListOfDeclaredTypes,ListOfTypes,ListOfAllTypes),
    append(ListOfAllTypes,ListOfSubTypes,ListOfDepTypes).


adviceTypeDep(SubAspectName,AdviceType,AdviceId,ListOfDepTypes):-
    is_aspect(SubAspectName),is_aspect(SuperAspectName),
    superAspect(SuperAspectName,SubAspectName),
    triggerAdvice(SuperAspectName,AdviceType,AdviceId,ListOfAdviceParameters,_),
    findall(Type,(getMember(Type,ListOfAdviceParameters),is_type(Type)),
            ListOfDeclaredTypes),
    findall(Type,(getMember(X,ListOfAdviceParameters),(X = object),(is_class(Type);
                  is_interface(Type))),ListOfTypes),
    findall(Subtype,(getMember(SuperType,ListOfAdviceParameters),is_type(SuperType),
                     subType(Subtype,SuperType)),ListOfSubTypes),
    append(ListOfDeclaredTypes,ListOfTypes,ListOfAllTypes),
    append(ListOfAllTypes,ListOfSubTypes,ListOfDepTypes).
```

Table 22: Obtaining the list of user-defined types being members of the parameter list of a given advice.

the class (or interface) is the type of a parameter of an inter-type declaration or the return type of an inter-type declaration of the aspect. In this case the inheritance relationship should also be taken into consideration. That is, if a class or interface has subtypes, then the aspect is also dependent on its subtypes. Table 24 shows the corresponding Prolog rule identifying this dependency.

Pointcut-designator dependency

In a `call` or `execution` join point, we may have a complete signature of a method together with the return type and access modifier of the method, or we

```
adviceMethodDep(AspectName,AdviceType,ListOfAdviceParameters,ListOfMethods):-
    is_aspect(AspectName),
    (advicePointcutMap(AspectName,AdviceType,_,PointcutName);
     (advicePointcutMap(SubAspectName,AdviceType,_,PointcutName),
      subAspect(SubAspectName,AspectName))),
    advisedBy(TypeName,MethodName,_,AspectName,AdviceType,_,ListOfAdviceParameters,
              PointcutName),
    append([TypeName],[MethodName],ListOfMethods).
```

Table 23: Obtaining the list of methods being advised by a given advice in a given aspect.

may have wildcards. Having a wildcard increases the level of dependency between the aspects and the base classes, and it also leads to accidental join points [48]. We have defined 32 rules to identify pointcuts which have designators with wildcards. In what follows we elaborate eight of them. As part of the naming convention these rules correspond to the type of information they provide.

- `pointcutMonitorAllTypeMethod(AspectName,ListOfPointcuts).`

  This rule identifies all the pointcuts which have the `call(* * *.*(..))` or `execution(* * *.*(..))` designator. A pointcut with this designator is monitoring all methods with any parameters, return type, and access modifier defined in the system. Therefore, it is dependent to all the classes and interfaces. Table 25 shows the implementation of the Prolog rule `pointcutMonitorAllTypeMethod`.

- `pointcutMonitorMethodWithParam(AspectName,ListOfPointcuts,`
  `ListOfMethodParameters).`

  This rule identifies all the pointcuts which have the `call(* * *.*(ListOfMethod`

```
interTypeDep(AspectName,ListofDepType,IntdMethodName):-
        (is_aspect(AspectName),
         introducedMethod(AspectName,_,IntdMethodName,_,ReturnType,ListOfParam,_)),
        findall(TypeName,(getMember(TypeName,ListOfParam),is_type(TypeName)),
                ListofDeclaredtype),
        findall(TypeName,(getMember(X,ListOfParam),(X = object),(is_class(TypeName);
                          is_interface(TypeName))),Listoftype),
        findall(Subtype,(getMember(SuperType,ListOfParam),
                         is_type(SuperType),subType(Subtype,SuperType)),ListofSubtype),
        append(ListofDeclaredtype,Listoftype,ListofAlltype),
        append(ListofAlltype,ListofSubtype,ListofDepType1),
        findall(TypeName1,((TypeName1 = ReturnType),(is_type(TypeName1))),
                ListofDeclaredtype1),
        findall(TypeName1,((ReturnType = object),(is_class(TypeName1);
                            is_interface(TypeName1))),Listoftype1),
        findall(Subtype1,((SuperType1 = ReturnType),is_type(SuperType1),
                          subType(Subtype1,SuperType1)),ListofSubtype1),
        append(ListofDeclaredtype1,Listoftype1,ListofAlltype1),
        append(ListofAlltype1,ListofSubtype1,ListofDepType2),
        append(ListofDepType1,ListofDepType2,ListofDepType).
```

Table 24: Obtaining the types which are either return type or members of the parameter list of an introduced method.

`Parameter))`, or `execution(* * *.*(ListOfMethodParameter))` designator. A pointcut with this designator is monitoring all methods with the specified list of parameters, but any return type, and access modifier defined in the system. The Prolog rule `pointcutMonitorMethodWithParam` is shown in Table 26.

- `pointcutMonitorSpecificMethodName(AspectName,ListOfPointcuts,MethodName)`.

This rule identifies all the pointcuts which have the `call(* * *.MethodName(..))`, or `execution(* * *.MethodName(..))` designator. A pointcut with this designator is monitoring all methods with the specified name, but any parameter, return type, and access modifier defined in the system. The implementation of the Prolog rule `pointcutMonitorSpecificMethodName` is shown in

```
pointcutMonitorAllTypeMethod(AspectName,ListOfPointcuts):–
pointcut(AspectName,PointcutName,_,_,_),
findall(PointcutName,pointcutdesig(_,AspectName,PointcutName,_,[_,_,every,every,[any]]),
        ListOfPointcuts),
(ListOfPointcuts\= []).
```

Table 25: Obtaining pointcuts monitoring all methods in the system.

```
pointcutMonitorMethodWithParam(AspectName,ListOfPointcuts,ListOfMethodParameters):–
      pointcut(AspectName,PointcutName,_,_,_),
      method(_,_,_,_,ListOfMethodParameters,_),
      findall(PointcutName,pointcutdesig(_,AspectName,PointcutName,_,[_,_,every,every,
              ListOfMethodParameters]),ListOfPointcuts),
      (ListOfPointcuts\= []).
```

Table 26: Obtaining pointcuts monitoring all methods with a given list of parameters.

Table 27.

```
pointcutMonitorSpecificMethodName(AspectName,ListOfPointcuts,MethodName):–
pointcut(AspectName,PointcutName,_,_,_),is_method(MethodName),
findall(PointcutName,pointcutdesig(_,AspectName,PointcutName,_,[_,_,every,MethodName,
                                   [any]]),ListOfPointcuts),(ListOfPointcuts\= []).
```

Table 27: Obtaining pointcuts monitoring all methods with a given name.

- `pointcutMonitorSpecificMethodNameWithParam(AspectName,ListOfPointcuts, MethodName,ListOfMethodParameters).`

This rule identifies all the pointcuts which have the `call(* * *.MethodName (ListOfMethodParameters))`, or `execution(* * *.MethodName(ListOfMethodPara meters))` designator. A pointcut with this designator is monitoring all methods defined in the system with the specified name and list of parameters, any return type, and any access modifier. The implementation of the Prolog rule `pointcutMonitorSpecificMethodNameWithParam` is shown in Table 28.

```
pointcutMonitorSpecificMethodNameWithParam(AspectName,ListOfPointcuts,MethodName,
                                           ListOfMethodParameters):-
pointcut(AspectName,PointcutName,_,_,_),
method(_,MethodName,_,_,ListOfMethodParameters,_),
findall(PointcutName,pointcutdesig(_,AspectName,PointcutName,_,[_,_,every,MethodName,
                                   ListOfMethodParameters]),ListOfPointcuts),
(ListOfPointcuts\= []).
```

Table 28: Obtaining pointcuts monitoring all methods with a given name and list of parameters.

- `pointcutMonitorAllMethodsOfTypeName(AspectName,ListOfPointcuts,TypeName).`

  This rule identifies all pointcuts which have the `call(* * TypeName.*(..))`, or `execution(* * TypeName.*(..))` designator. A pointcut with this designator is monitoring all methods defined for the specified `TypeName`. The implementation of the Prolog rule `pointcutMonitorAllMethodsOfTypeName` is shown in Table 29.

```
pointcutMonitorAllMethodsOfTypeName(AspectName,ListOfPointcuts,TypeName):-
pointcut(AspectName,PointcutName,_,_,_),is_type(TypeName),
findall(PointcutName,(pointcutdesig(_,AspectName,PointcutName,_,[_,_,Atype,every,[any]]),
                      ((Atype=TypeName);superType(Atype,TypeName))),ListOfPointcuts),
(ListOfPointcuts\= []).
```

Table 29: Obtaining pointcuts monitoring all methods of a given type.

- `pointcutMonitorAllMethodsOfTypeNameWithParam(AspectName,ListOfPointcuts,`
  `TypeName,ListOfMethodParameters).`

  This rule identifies all pointcuts which have the `call(* * TypeName.*(ListOf`

`MethodParameters))`, or `execution(* * TypeName.*(ListOfMethodParameters))` designator. A pointcut with this designator is monitoring all methods which have the specified list of parameter and defined for the specified `TypeName`. The implementation of the Prolog rule `pointcutMonitorAllMethodsOfTypeNameWithParam` is shown in Table 30.

```
pointcutMonitorAllMethodsOfTypeNameWithParam(AspectName,ListOfPointcuts,TypeName,
                                             ListOfMethodParameters):-
pointcut(AspectName,PointcutName,_,_,_),
method(TypeName,_,_,_,ListOfMethodParameters,_),
findall(PointcutName,pointcutdesig(_,AspectName,PointcutName,_,[_,_,TypeName,every,
                                   ListOfMethodParameters]),ListOfPointcuts),
(ListOfPointcuts\= []).
```

Table 30: Obtaining pointcuts monitoring all methods of a given type that have a given list of parameters.

- `pointcutMonitorMethodNameOfTypeName(AspectName,ListOfPointcuts,`
  `TypeName,MethodName)`

  This rule identifies all pointcuts which have the `call(* * TypeName.MethodName(..))`, or `execution(* * TypeName.MethodName(..))` designator. A pointcut with this designator is monitoring all methods defined for the specified `TypeName`. The implementation of the Prolog rule `pointcutMonitorMethodNameofTypeName` is shown in Table 31.

- `pointcutMonitorMethodNameOfTypeNameWithParam(AspectName,ListOfPointcuts,`
  `TypeName,MethodName,ListOfMethodParameters).`

  This rule identifies all pointcuts which have the `call(* * TypeName.Method`

```
pointcutMonitorMethodNameOfTypeName(AspectName,ListOfPointcuts,TypeName,
                                    MethodName):-
pointcut(AspectName,PointcutName,_,_,_),
method(TypeName,MethodName,_,_,_,_),
findall(PointcutName,pointcutdesig(_,AspectName,PointcutName,_,[_,_,TypeName,
                                 MethodName,[any]]),ListOfPointcuts),
(ListOfPointcuts\= []).
```

Table 31: Obtaining pointcuts monitoring methods of a given type with a given name.

`Name(ListOfMethodParameters))`, or `execution(* * TypeName.MethodName(ListOf MethodParameters))` designator. A pointcut with this designator is monitoring all methods with the specified list of parameter defined for the specified `TypeName`. The implementation of the Prolog rule `pointcutMonitorMethodNameofTypeNameWithParam` is shown in Table 32.

```
pointcutMonitorMethodNameOfTypeNameWithParam(AspectName,ListOfPointcuts,TypeName,
                                          MethodName,ListOfMethodParameters):-
pointcut(AspectName,PointcutName,_,_,_),
method(TypeName,MethodName,_,_,ListOfMethodParameters,_),
findall(PointcutName,pointcutdesig(_,AspectName,PointcutName,_,
                                 [_,_,TypeName,MethodName, ListOfMethodParameters]),
    ListOfPointcuts),(ListOfPointcuts\= []).
```

Table 32: Obtaining pointcuts monitoring a method with a given signature.

5.3.2 Atomic changes

We have adopted a subset of atomic changes[3] defined in [54] as well as refined some and introduced a new one. A list of these changes is provided in Table 33. In

[3]In object-oriented programming classes, methods, fields and their interrelationships are considered as the atomic units of change [24]. This definition is extended in [54] for aspect-oriented programming.

order to detect the impact of changes in system we need to create a relationship between the changes and the dependencies defined in previous section.

Abbreviation	Atomic Change Name
ANM	Add a New Method
CMS	Change a Method Signature
DIM	Delete an Introduced Method
ANA	Add a New Advice [refined]
CAT	Change Advice Type [introduced]
DAD	Delete an Advice [refined]
ANP	Add a New Pointcut
DPC	Delete a Pointcut

Table 33: A catalog of atomic changes in AspectJ.

Add a new method

A new method can be added for a type directly, or by an aspect through inter-type declaration. We are interested in the identification of possible effects of such a change to a system. If in our system we have a pointcut definition like `pointcut p():call(* *.*(..))` , the newly added method would be captured automatically by this pointcut. Moreover, if we have more than one pointcut with this designator, then more than one advice would be executed. Because having pointcut designators with wildcards may lead to collecting unexpected join points, we should check if there exists any pointcut designator that matches the signature and return type of the newly added method. We have already defined a rule in our Prolog database to treat an introduced method as a regular method (See Table 34).

Table 35 shows all possible cases for matching `(void <typeName>.<methodName> (<params>))` with a pointcut designator.

```
method(TypeName,MethodName,Visibility,ReturnType,ListOfMethodParameters,
        MethodType):-
        is_aspect(AspectName),
        introducedMethod(AspectName,TypeName,MethodName,Visibility,ReturnType,
                                ListOfMethodParameters,MethodType).
```

Table 34: Considering an introduced method as declared method.

Next, we define a rule to detect the affect of adding a new method as `void <typeName>.<methodName>(<params>)` (see Table 36).

Adding a new method may also have other affects in system if this method is introduced by an aspect through inter-type declaration. In [26] the authors discuss cases where introducing a new method through inter-type declaration may lead to conflicts. For example, if a method is defined by an aspect for a type, then all its subtypes inherit this method. A conflict may occur if a method with the same name, signature, and return type is defined for one of its subtypes. In this case the body of the newly defined method for a subtype overrides the method defined in the supertype. However, this may lead to some unexpected behavior if it is done unintentionally. Therefore, we need to identify these cases where the conflict may occur. The Prolog implementation of such a rule **`findConflictForAddMethod`** shown in Table 37.

Change a method signature

Changing a method signature leads to compilation errors where this method

Part of pointcut designator	Definition	Prolog rule
`* void *.*(..)`	All types and all methods with any list of parameter	pointcutMonitorAllTypeMethod(AspectName, ListOfPointcuts)
`* void *.*(<params>)`	All types and all methods with the `<params>` as parameter list	pointcutMonitorMethodWithParam(AspectName, ListOfPointcuts,ListOfMethodParameters)
`* void *.<methodName>(..)`	All types having a method named `<methodName>`	pointcutMonitorSpecificMethodName(AspectName, ListOfPointcuts,MethodName)
`* void *.<methodName>(<params>)`	All types having a method named `<methodName>` with the `<params>` as parameter list	pointcutMonitorSpecificMethodNameWithParam (AspectName,ListOfPointcuts,MethodName, ListOfMethodParameters)
`* void <typeName>.*(..)`	All methods belong to `<typeName>` or its supertypes	pointcutMonitorAllMethodsOfTypeName (AspectName,ListOfPointcuts,TypeName)
`* void <typeName>.*(<params>)`	All methods of `<typeName>` or its supertypes which have `<params>` as parameter list	pointcutMonitorAllMethodsOfTypeName WithParam(AspectName,ListOfPointcuts, TypeName,ListOfMethodParameters)
`* void <typeName>.<methodName>(..)`	All methods of the `<typeName>` or its supertypes which are named `<methodName>`	pointcutMonitorMethodNameOfTypeName (AspectName,ListOfPointcuts,TypeName, MethodName)
`* void <typeName>.<methodName>(<params>)`	Method of the `<typeName>` or its supertypes which is named `<methodName>` and has `<params>` as parameter list	pointcutMonitorMethodNameOfTypeName WithParam(AspectName,ListOfPointcuts, TypeName,MethodName,ListOfMethodParameters)

Table 35: List of pointcut designators that match `void <typeName>.<methodName> (<params>)`.

is called. When one decides to perform such a change, these compilation errors need to be resolved. However, resolving the compilation errors is sometimes not sufficient. Changing a method signature may deactivate some of the pointcuts in the system without any sign of error. When a pointcut is deactivated, the advice blocks bound to it will not be executed. Therefore, before changing a method signature, all its dependent pointcuts need to be identified, and the corresponding modifications should be applied on them. By deploying rule

```
addMethod(AspectName,ListOfPointcuts,TypeName,MethodName,ListOfMethodParameters):-
is_aspect(AspectName),(pointcutMonitorAllTypeMethod(AspectName,ListOfPointcuts);
(pointcutMonitorMethodWithParam(AspectName,ListOfPointcuts,ListOfMethodParameters);
 pointcutMonitorSpecificMethodName(AspectName,ListOfPointcuts,MethodName);
 pointcutMonitorSpecificMethodNameWithParam(AspectName,ListOfPointcuts,
                                        MethodName,ListOfMethodParameters);
 pointcutMonitorAllMethodsOfTypeName(AspectName,ListOfPointcuts,TypeName);
 pointcutMonitorAllMethodsOfTypeNameWithParam(AspectName,ListOfPointcuts,
                                            TypeName,ListOfMethodParameters);
 pointcutMonitorMethodNameOfTypeName(AspectName,ListOfPointcuts,TypeName,
                                    MethodName);
 pointcutMonitorMethodNameOfTypeNameWithParam(AspectName,ListOfPointcuts,
                                             TypeName,MethodName,
                                             ListOfMethodParameters)).
```

Table 36: Obtaining all pointcuts which may monitor a method with a given signature.

```
findConflictForAddMethod(AspectName,TypeName,MethodName,ListOfMethodParameters,
                         ReturnType):-
        findDeclaredMethod(AspectName,TypeName,SuperTypeName,MethodName),
        method(SuperTypeName,MethodName,_,ReturnType,ListOfMethodParameters,_).
```

Table 37: Obtaining the aspect defining a similar method to a supertype of a given type.

`pointcutMonitorMethodNameOfTypeNameWithParam` defined in Section 5.3.1, we can identify the list of pointcuts monitoring a given method.

Delete an introduced method

Deleting a method in the system leads to compilation errors where this method is called. However there are cases where a method deletion does not produce any compilation errors, even though it may affect the behavior of a system. An interface is a group of related methods with empty bodies. In AspectJ an interface

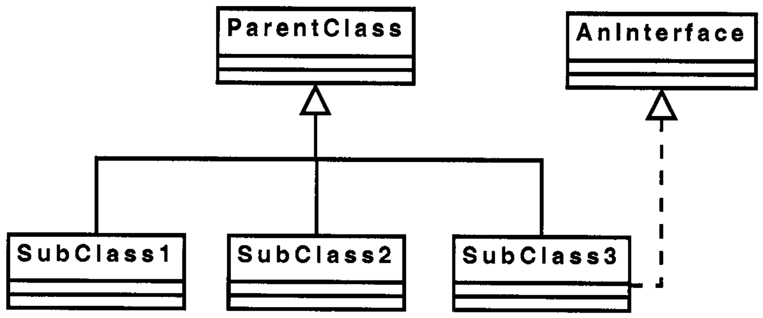


Figure 3: Example of inheritance hierarchy.

can be defined inside the body of an aspect, and it is possible to implement the methods of the interface in the aspect through inter-type declarations. Figure 3 shows an example of an inheritance hierarchy.

In this example, all the subclasses inherit the method `foo()` defined in the superclass. In Table 38, aspect `AnAspect` defines method `foo()` for class `ParentClass`. In this case, all the subtypes of class `ParentClass` inherit this method. However, aspect `AnotherAspect` also defines a method with the same signature and name for class `SubClass3` indirectly through inter-type declaration. If method `foo()` is invoked on instances of class `SubClass3`, the body of the method defined for the interface will be executed. Having such an implementation may be an intentional design decision, with the programmer aiming to provide a different implementation for one of the subclasses. A problem may occur in this case if one deletes the introduced method for the interface `AnInterface`. This would not result in a compilation error, as class `SubClass3` will have the implementation of method `foo()` from its superclass, although it may not be desirable. Table 39 shows the

Prolog rule to find any possible conflict for deleting an introduced method.

```
public aspect AnAspect {
        public void ParentClass.foo(){
                System.out.println("Foo of parent");}
        //.. some code }

public aspect AnotherAspect {
        public void AnInterface.foo(){
                System.out.println("Foo of interface");}
        //.. Some code}
```

Table 38: Partial code of aspects AnAspect and AnotherAspect.

```
findConflictForDeleteMethod(TypeName,ConfList):-
  is_type(TypeName),
  findall(ListOfMethods,listOfIntroducedMethod(TypeName,ListOfMethods),List),
  \+ List=[],(getMember(X,List),Temp1=X, subtract(List,[Temp1],Difference),
              getMember(Y,Difference),Temp2=Y,intersection(Temp1,Temp2,ConfList),
              \+ ConfList=[]) .
```

Table 39: Obtaining methods of a given type overriding an introduced method.

Add a new advice

When an advice is added to an aspect, it should be bound to a named pointcut defined in the aspect (or in its superaspect), or it should be bound to an anonymous pointcut[4] [33]. This implies that an advice block will be executed at well-defined points in the execution of program. In other words, adding a new advice to an aspect would affect the control flow of the system. In Section 5.3.1 we have defined four rules, `pointcutTypeMethodDep`, `pointcutMethodDep`, `pointcutTypeDep`, and `pointcutDep` which capture the dependency between a pointcut and methods. We

[4]In this research, we have assumed that all pointcuts are named.

deploy these rules to find the list of methods being advised by an advice if the advice is to be added to a given aspect. Table 40 shows the Prolog implementation of rule addNewAdvice.

```
addNewAdvice(AspectName,AdviceType,PointcutName,ListOfMethods1,ListOfMethods2,
        ListOfMethods3,ListOfMethods4):-
      (pointcutTypeMethodDep(AspectName,PointcutName,TypeName,MethodName,_),
       append([TypeName],[MethodName],ListOfMethods1));
      (pointcutMethodDep(AspectName,PointcutName,ListOfTypes,MethodName,_),
       append([ListOfTypes],[MethodName],ListOfMethods2));
      (pointcutTypeDep(AspectName,PointcutName,TypeName,ListOfMethods,_),
       append([TypeName],[ListOfMethods],ListOfMethods3));
      (pointcutDep(AspectName,PointcutName,ListOfTypeMethods,_),
       append([ListOfTypeMethods],[ListOfTypeMethods],ListOfMethods4)).
```

Table 40: Obtaining methods being advised by a new advice in a given aspect.

Change an advice type

It may be the case that for various reasons an advice type need to be changed. Often changing the advice type (for example from `before` to `after`, or `around`) modifies the system control flow. Indeed manually detecting all places where this change affects the system control flow, is a tedious and error-prone process specially for a medium to large-scale applications. Therefore, it is useful to have information about the methods which are advised by this advice. Changing an advice type may lead to unintended behavior of the system if there exist advice blocks relevant[5] to this advice. This implies that before modification of an advice type one needs i) information about the methods which are advised by it, and ii) its relevant advice

[5]Two pieces of advice are relevant, if they are defined in different aspects, apply at at least one common join point and are either of the same kind or at least one of them is around-advice [47].

block(s). Table 41 shows the Prolog rule **changeAdviceType** which finds the list of methods advised by a given advice defined in a given aspect and all its relevant advice blocks.

```
changeAdviceType(AspectName,AdviceType,AdviceId,ListofTypeMethod,
                 ListOfRelevantAdvicePerAspect,ListOfRelevantAdvicePerMultipleAspect,
                 ListOfAspectWithPrecedence):-
    is_aspect(AspectName),
    (methodAdviceby(AspectName,AdviceType,AdviceId,ListofTypeMethod);
    findReleventAdvicePerAspect(AspectName,AdviceType,AdviceId,
                                ListOfRelevantAdvicePerAspect);
    findReleventAdvicePerMultipleAspect(AspectName,AdviceType,AdviceId,
                                        ListOfRelevantAdvicePerMultipleAspect);
    getPrecedence(AspectName,ListOfAspectWithPrecedence)).
```

Table 41: Obtaining list of relevant advice blocks for a given advice.

Delete an advice

Deletion of advice affects the control flow of the system, because the system will lose the functionality to be executed at joint points. For example, deleting a `before` advice that checks the precondition of a method may lead to wrong or unexpected results. In Section 5.3.1, we defined rule **adviceMethodDep** which captures the dependency between an advice and methods. Using this rule we can identify list of methods being advised by an advice. Having this list helps to make a better decision before the deletion of an advice.

Add a new pointcut

Adding a pointcut with a name which already exists (for another pointcut) in the same aspect leads to compiler error. However, it is possible to have a pointcut defined in an aspect with the same name as a pointcut in its superaspect. This is referred to as pointcut overriding [49]. If the subaspect does not provide an advice block for the overriding pointcut, the advice block(s) defined in the superaspect will be invoked when the overriding pointcut is captured. However, if there exists an advice definition in the subaspect for the overriding pointcut, this advice block will be invoked together with the advice block(s) defined in the superaspect. In this case, the order of execution will be based on the precedence rules [50]. This implies that adding a new pointcut may lead to unexpected behavior in the system. Table 42 shows the Prolog rule `addNewPointcut` to find the pointcut with the same name as the one to be added.

```
addNewPointcut(SubAspectName,PointcutName,ListOfParameters,List):-
                    is_aspect(SubAspectName),superAspect(SuperAspectName,SubAspectName),
                    pointcut(SuperAspectName,PointcutName,ListOfParameters,_,0),
                    append([SuperAspectName],[PointcutName],List).
```

Table 42: Obtaining all pointcuts having the same name as a given pointcut.

Delete a pointcut

Deleting a pointcut in an aspect leads to a compilation error if there are advice blocks bound to it. However, if one deletes a pointcut which is overriding a pointcut in a superaspect no compilation error will occur, and consequently the advice blocks bound to this pointcut will be deactivated. This indicates that some of the functionalities to be executed after, before, or instead of a method will not be executed. Table 43 shows the Prolog implementation of rule `deletePointcut`.

This rule identifies the list of methods which will not be affected by the advice blocks upon deletion of a given pointcut.

```
deletePointcut(AspectName,PointcutName,ListOfMethods):-
is_aspect(AspectName),is_aspect(SuperAspectName),
superAspect(SuperAspectName,AspectName),
pointcut(AspectName,PointcutName,_,_,_),
pointcut(SuperAspectName,PointcutName,_,_,_),
findall(MethodName,pointcutdesig(_,AspectName,PointcutName,_,
                                  [_,_,TypeName,MethodName,_]),List),
(getMember(X,List),findall(TypeName,pointcutdesig(_,AspectName,PointcutName,_,
                                                   [_,_,TypeName,X,_]),ListOfTypes),
 append(ListOfTypes,[X],ListOfMethods)).
```

Table 43: Obtaining all methods which will not be affected by the advice blocks bound to a given pointcut.

Chapter 6

Case study

As a proof of concept, we deploy our approach over a service-oriented system which we has been developed in the context of another project. Even though there exist large-scale open source aspect-oriented systems[1], they contain a large number of classes but a few aspects (while most of these aspects are performing a simple functionality, as logging). Therefore, we have decided to use the Infomediator system, since it has more complex aspects, for example `ContractChecking`, `Synchronization`, and `ObserverProtocol`. For the complete implementation of the Infomediator system and its corresponding transformation to Prolog facts see Appendix A.

The system allows possibly multiple consumers and retailers to have access to a communication hub which controls and coordinates their interactions in order to implement the reverse auction protocol. In this protocol one consumer may place

[1]http://sourceforge.net/

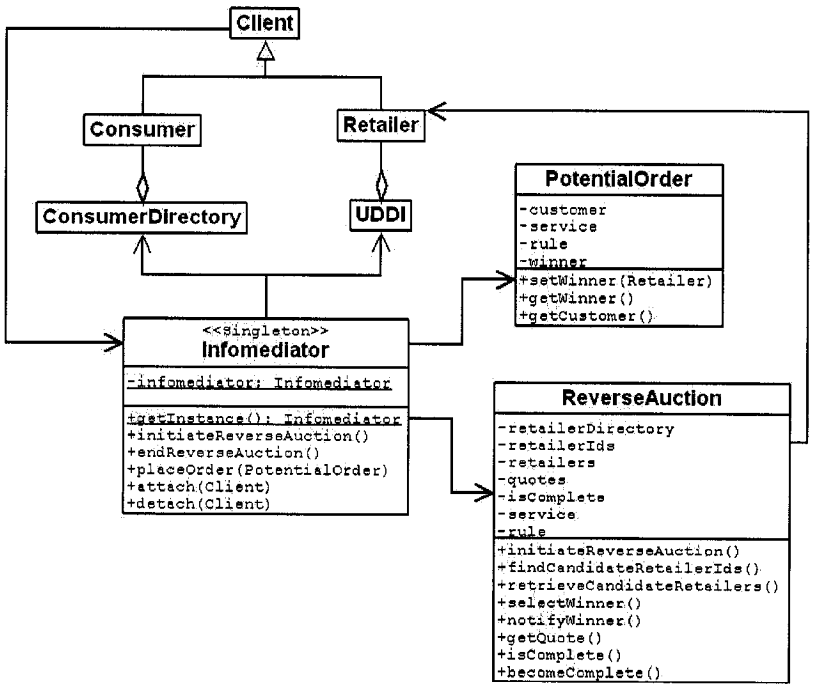


Figure 4: Partial class diagram of the system.

a request to purchase a product. After a collection of sellers is iterated over to find the best price, a message is sent back to the client informing them about the winner retailer and asking for confirmation to place an order. The core functionality is provided by the definitions of classes `Infomediator` and `ReverseAuction`. Figure 4 shows a partial class diagram of the system.

For each reverse auction request, a potential order is created and associated with the consumer who originated the reverse auction and with the winner of the reverse auction (see Figure 5). The system routes the auction result back to the consumer and informs the winner.

Supported semantics and other technical services (such as the subject-observer protocol, contract checking, authentication, synchronization, transaction logging,

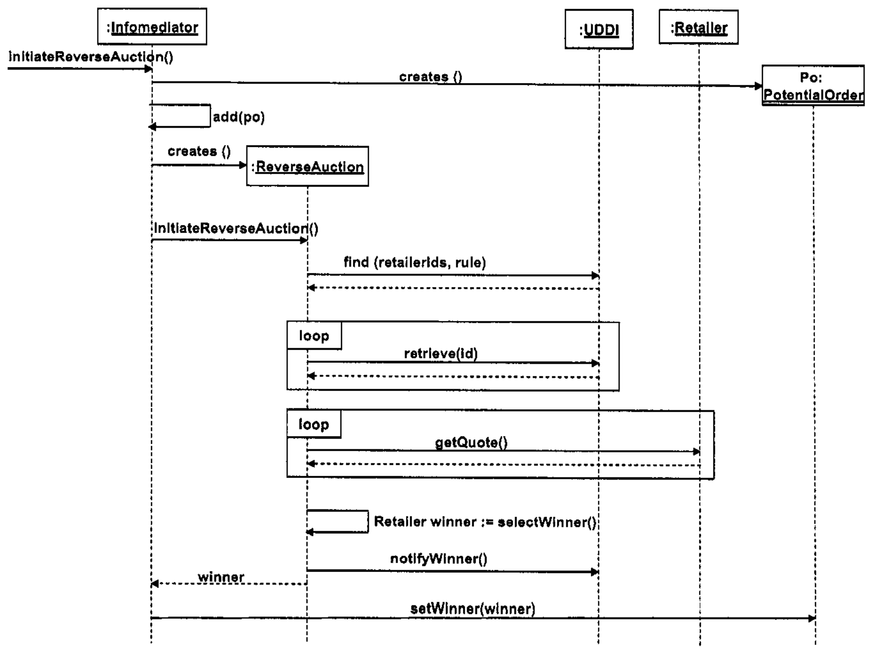


Figure 5: UML sequence diagram for system operation initiateReverseAuction().

throughput and persistence) are provided by a number of aspect definitions. One notable example is the aspectual behavior of the aspect `ObserverProtocol` (see Table 44): This aspect is implemented as an abstract aspect which defines the Observer design pattern [20], where retailers are viewed as observers and customers are viewed as subjects. The definition of aspect `CoordinateObserver` extends aspect `ObserverProtocol` and provides concrete pointcuts. A list of all retailers participating in the reverse auction is created when a reverse auction is initiated. If a customer eventually purchases the service from the winner of the auction, the corresponding retailer will be notified with the information about the number of items sold. A segment of the transformation of aspect `coordinateObserver` to Prolog facts is provided in Table 46. The system has nine classes and eight aspects. Table 45 presents information regarding the number of source lines of code (SLOC)[2], number of methods[3], pointcuts and advice blocks of the system.

```
public abstract aspect ObserverProtocol {
        protected interface Subject {
                public void addObserver(Observer o);
                public void removeObserver(Observer o);
                private List observers = new LinkedList();}
        private synchronized void Subject.notifyObserver(PotentialOrder po){...}
        public interface Observer {
                public void notifyOfchange(Subject s, PotentialOrder po);}
        protected abstract pointcut subjectChange(Subject s, PotentialOrder po);
        after(Subject s, PotentialOrder po): subjectChange(s, po){...}
        protected abstract pointcut findObservers(Infomediator info,
                                        Subject s, String service, String rule);
        after(Infomediator info, Subject s, String service,String rule):
                                    findObservers(info, s, service, rule){..}}

privileged public aspect CoordinateObserver extends ObserverProtocol {
        declare parents : Retailer implements Observer;
        declare parents : Customer implements Subject;
        private int Retailer.NumberSold = 0;
        public void Retailer.notifyOfchange(Subject s, PotentialOrder po)
                                      {NumberSold++;...}
        protected pointcut subjectChange(Subject s, PotentialOrder po):
                execution(* Customer.buy(PotentialOrder))
                && target(s) && args(po);
        protected pointcut findObservers(Infomediator info, Subject customer,
                                            String service, String rule):
                execution (* Infomediator.initiateReverseAuction(Customer,
                                                                  String,
                                                                  String))
                && target(info) && args(customer, service, rule);}
```

Table 44: Partial code of aspects `ObserverProtocol` and `CoordinateObserver`.

In the following subsections, we illustrate examples of general rules, measurement rules, dependency rules, and change impact analysis rules applied on the Infomediator system.

[2]Only non-empty and non-commented lines of code are counted.

[3]Class constructors and methods defined through inter-typed declaration are considered as methods.

Class/Aspect	SLOC	Method	Pointcut	Advice
Client	11	2	NA	NA
Customer	11	2	NA	NA
CustomerDirectory	15	3	NA	NA
Infomediator	49	8	NA	NA
PotentialOrder	23	5	NA	NA
Quote	17	4	NA	NA
Retailer	38	8	NA	NA
ReverseAuction	72	9	NA	NA
UDDI	49	7	NA	NA
Authentication	46	2	2	2
ContractChecking	32	0	2	2
CoordinateObserver	14	1	2	0
ObserverProtocol	39	3	2	2
Persistence	9	0	1	1
Synchronization	24	0	1	1
Throughput	12	0	2	2
TransactionLogging	51	0	4	4
Total	512	54	16	14

Table 45: Statistics about the source code of Infomediator.

6.1 Inter-type declared method in inheritance hierarchy

As shown in Figure 5, class `Customer` extends class `Client`. This implies that objects of type `Customer` inherit the behavior defined for the supertypes. Here we want to obtain *Declared Method in Inheritance Hierarchy* for class `Customer`. Manually, this task would be tedious because one needs to check all aspect definitions in the system in order to obtain this information. According to Section 5.2.3 we run query `findDeclaredMethod(Aspect,customer, SuperType, Method)`, and we show the result in Table 47.

```
aspect(coordinateobserver,package).
privilegedAspect(coordinateobserver).
extends(coordinateobserver,observerprotocol).
introducedMethod(coordinateobserver,retailer,notifyOfchange,public,void,[subject,
                                                              potentialorder],0).
introducedAtt(coordinateobserver,retailer,numbersold,int,private,0).
pointcut(coordinateobserver,subjectchange,[subject,potentialorder],protected,0).
pointcutdesig(1412294,coordinateobserver,subjectchange,execution,
          [public,all,customer,buy,[potentialorder]]).
pointcutdesig(13452612,coordinateobserver,subjectchange,target,[subject]).
pointcutdesig(8287698,coordinateobserver,subjectchange,args,[potentialorder]).
pointcut(coordinateobserver,findobservers,[infomediator,subject,string,string],protected,0).
pointcutdesig(6901522,coordinateobserver,findobservers,execution,
          [public,all,infomediator,initiatereverseauction,[customer,string,string]]).
pointcutdesig(29769356,coordinateobserver,findobservers,target,[infomediator]).
pointcutdesig(3431235,coordinateobserver,findobservers,args,[customer, service, rule]).
...
```

Table 46: Partial Prolog facts resulting from the transformation of aspect `coordinateObserver`.

```
findDeclaredMethod(Aspect,customer,SuperType,Method).

Result:
Aspect = ObserverProtocol, SuperType = Subject, Method = addobserver;
Aspect = ObserverProtocol, SuperType = Subject, Method = removeobserver;
Aspect = ObserverProtocol, SuperType = Subject, Method = notifyobserver;
```

Table 47: Result of query `findDeclaredMethod`.

6.2 Messages to which an object can respond

Inter-type declaration allows aspects to define state/behavior for classes/interfaces outside their definition. Therefore, in order to find the messages to which an object can respond, one needs to check i) the class definition of the declared type, ii) the class definition of the supertypes of the declared type, iii) all aspect definitions for the inter-type declared methods for the static type and its supertypes. As an example, to do this task manually for the class `Customer`, one should check the definition of class `Customer` to find all methods with `public` or `protected` access modifiers. Moreover, the definitions of all supertypes of this class need to be checked in order to find all its inherited methods. Finally, all aspect definitions need to be checked for the inter-type declared methods for the class `Customer` or its supertypes. In order to find all messages to which an object of type `Customer` responds, one should run query `respondTo(customer,List)` defined in Section 5.2.3. The result of this query is shown in Table 48.

```
respondTo(customer,List).

List = [[buy, [potentialorder]], [getname, []], [getid, []], [addobserver, [observer]],
       [removeobserver, [observer]]];
```

Table 48: Result of query `respondTo`.

Class `Customer` has one declared method (`buy(PotentialOrder)`), and it inherits two methods from its superclass `Client` (`getName()`, `getId()`). However, aspect `ObserverProtocol` introduces three methods for interface `Subject`, and this interface is declared to be the supertype of class `Customer` in aspect `coordinateObserver` (see Table 73 in Appendix A). Indeed, an object of type `Customer` has `addobserver(Observer)`, `removeobserver(Observer)`, `notifyobserver(PotentialOrder)` methods, but because the access modifier of method `notifyobserver(PotentialOrder)` is `private`, this method is not listed in Table 48.

6.3 Pointcut-Class dependence factor

Having pointcuts that expose context increases the coupling between aspects and classes. Pointcut-Class dependence factor shows the degree of interdependency between aspects and classes. In the following example, `Pointcut-Class Dependence` factor for the aspect `coordinateObserver` is calculated. According to Section 5.2.2, one should run the following query:

```
pointcutClassDependenceCount(coordinateObserver,TypeName,TotalCount).
```

The result of this query is provided in Table 49. As it is shown in Table 44, none of the parameters of the pointcut definitions in aspect `coordinateObserver` are of type `Customer`. Since class `Customer` is a subtype of interface `Subject`, and both pointcut definitions in aspect `coordinateObserver` have `Subject` in their parameter lists, `pointcutClassDependence` for class `Customer` should be equal to two.

Table 50 shows the Pointcut-Class dependence factor for all classes/ interfaces in the Infomediator system where A1, A2, A3, A4, A5, A6, A7, and A8 correspond to aspects `Authentication`, `ContractChecking`, `CoordinateObserver`, `ObserverProtocol`, `Persistence`, `Synchronization`, `Throughput`, and `TransactionLogging` respectively. The table shows that the pointcuts of aspect `Transactio-`

```
pointcutClassDependenceCount(coordinateobserver,TypeName,TotalCount).
TypeName = client, TotalCount = 0 ;
TypeName = customer, TotalCount = 2 ;
TypeName = customerdirectory, TotalCount = 0 ;
TypeName = demo, TotalCount = 0 ;
TypeName = infomediator, TotalCount = 1 ;
TypeName = potentialorder, TotalCount = 1 ;
TypeName = quote, TotalCount = 0 ;
TypeName = retailer, TotalCount = 0 ;
TypeName = reverseauction, TotalCount = 0 ;
TypeName = uddi, TotalCount = 0 ;
TypeName = observer, TotalCount = 0 ;
TypeName = subject, TotalCount = 2 ;
```

Table 49: Result of query `pointcutClassDependenceCount`.

`nLogging` are dependent to all classes and interfaces in the system. It also shows that aspect `Throughput` is not dependent on any class/ interface. We can also infer from the table that most of the pointcuts are dependent to the class `Infomediator`. Moreover, the table shows that the dependency among aspect `ObserverProtocol` and class `customer` is stronger than the dependency among aspect `ObserverProtocol` and class `Infomediator`. By comparing the data on this table, we can see that the values are very close to each other. However, if we had a relatively larger number (for example 10 or 12), this would be an indication for a high coupling between

Aspect / Class	A1	A2	A3	A4	A5	A6	A7	A8
Client	2	1	0	0	0	0	0	1
Customer	2	1	2	2	0	0	0	1
CustomerDirectory	0	0	0	0	0	0	0	1
Infomediator	0	1	1	1	1	1	0	1
Observer	0	0	0	0	0	0	0	1
PotentialOrder	0	0	1	1	0	0	0	1
Quote	0	0	0	0	0	0	0	1
Retailer	2	1	0	0	0	0	0	1
ReverseAuction	0	1	0	0	0	0	0	2
Subject	0	0	2	2	0	0	0	1
UDDI	0	0	0	0	0	0	0	1

Table 50: Pointcut-Class dependence factor for all classes and interfaces.

the corresponding class and aspect.

6.4 Advice-Type dependency

The obliviousness property of Aspect-Oriented systems creates a unidirectional visibility from aspects to classes. Context passing is one of the mechanisms through which an aspect can have visibility over classes. Context passing increases the coupling between aspects and classes. For example an advice becomes coupled to the class being passed to it through context passing, and hence any change in this class can affect this advice. In order to identify all aspects being coupled to a class through context passing, one should check all aspect definitions in the system. One can run query `adviceTypeDep(AspectName,AdviceType,_,[reverseauction])` defined in Section 5.3.1 in order to find the list of aspects which have visibility over class `ReverseAuction` through context passing. The result of this query is shown in Table 51.

```
adviceTypeDep(AspectName,AdviceType,_,[reverseauction]).

AspectName = contractchecking,
AdviceType = around ;

AspectName = transactionlogging,
AdviceType = after ;
```

Table 51: Result of query `adviceTypeDep`.

6.5 Add method

A pointcut that contains `*.*(..)` monitors all methods in the system. Consequently, if a new method is added to the system, it will be automatically monitored. However, this may not reflect the intentions of the maintainers. Therefore, one should check all aspect definitions in the system in order to find the pointcuts which can capture the newly added method. If one decides to add method `findLoser(PotentialOrder)` to class `ReverseAuction` in the Infomediator system in order to find all losers of an auction, it will be beneficial to know if there exist any pointcut with a designator that matches the signature of this method. In order to obtain this information, one can run query `addMethod(AspectName,ListOfPointcutName,reverseauction,findLoser,[potentialorder])` defined in Section 5.3.2. The result of this query is shown in Table 52.

```
addMethod(AspectName,ListOfPointcutName,reverseauction,findLoser,[potentialorder]).

Result:
AspectName = transactionlogging,
ListOfPointcutName = [captureevent] ;
```

Table 52: Result of query `addMethod`.

Chapter 7

Automation and tool support

This chapter discusses automation through the deployment of an Eclipse plug-in, AJSurf. For the installation of the tool, see Appendix B.

The rationale behind implementing an Eclipse plug-in is to integrate comprehension and change impact analysis activities in an environment already familiar to maintainers. Integrating our automation into Eclipse allows maintainers to edit, understand and perform change impact analysis without having to switching between different applications.

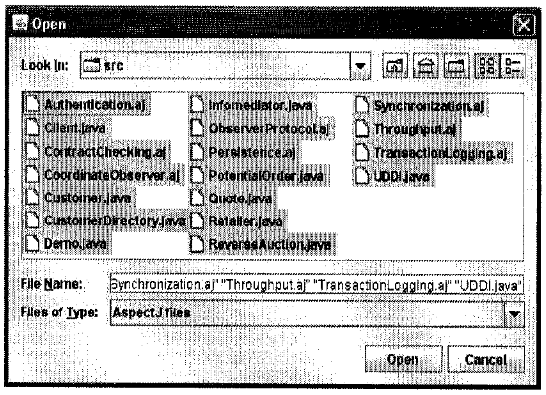
AJSurf consists of three major parts: A transformation component, a query component, and the plug-in interface.

1. The transformation component reads the source code and creates a database composed of a collection of Prolog facts. The transformation process from source code to facts is transparent to the users. The transformation of the

program to the facts is done in three steps. First, the AST corresponding to each compilation unit of the program (.java and .aj files) is retrieved and traversed. Second, the ASM is retrieved and traversed. Third, the extracted information from these steps is translated to facts according to the transformation rules. These Prolog facts are then added to the fact-base and used during the inference process. Traversing the AST is performed by depth-first traversal deploying the Visitor design pattern. This approach needs each compilation unit to be parsed before the traversal.

2. The query component is composed of a set of rules that capture relationships among entities in the system. These rules are context-free, i.e. they are independent from the particular applications in which they are deployed.

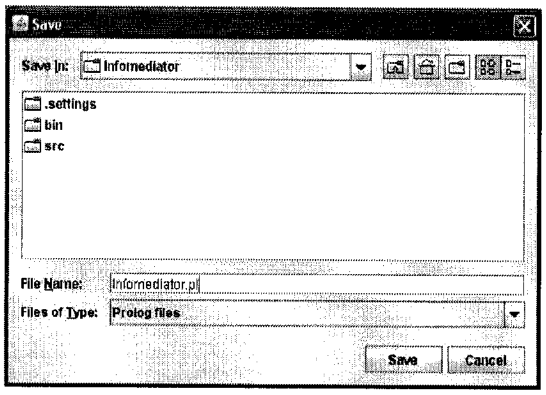
3. The plug-in interface is built using Eclipse Forms which are based on Standard Widget Toolkit (SWT) and JFace (a higher-level toolkit implemented using SWT) [15]. In order to execute the Prolog queries, the plug-in needs JPL - a Java interface to Prolog [2]. The JPL library gives us access to a set of java classes simulating Prolog data structures like Atom, Variable, Query etc. Thus one can run basic queries or parameterized queries and take the result in form of "Yes", "No", or a list of constants satisfying if the query is parameterized.



File Name: Synchronization.aj" "Throughput.aj" "TransactionLogging.aj" "UDDI.java"

Files of Type: AspectJ files

Figure 6: AJSurf: Choosing files to be transformed to Prolog facts.



File Name: Infomediator.pl

Files of Type: Prolog files

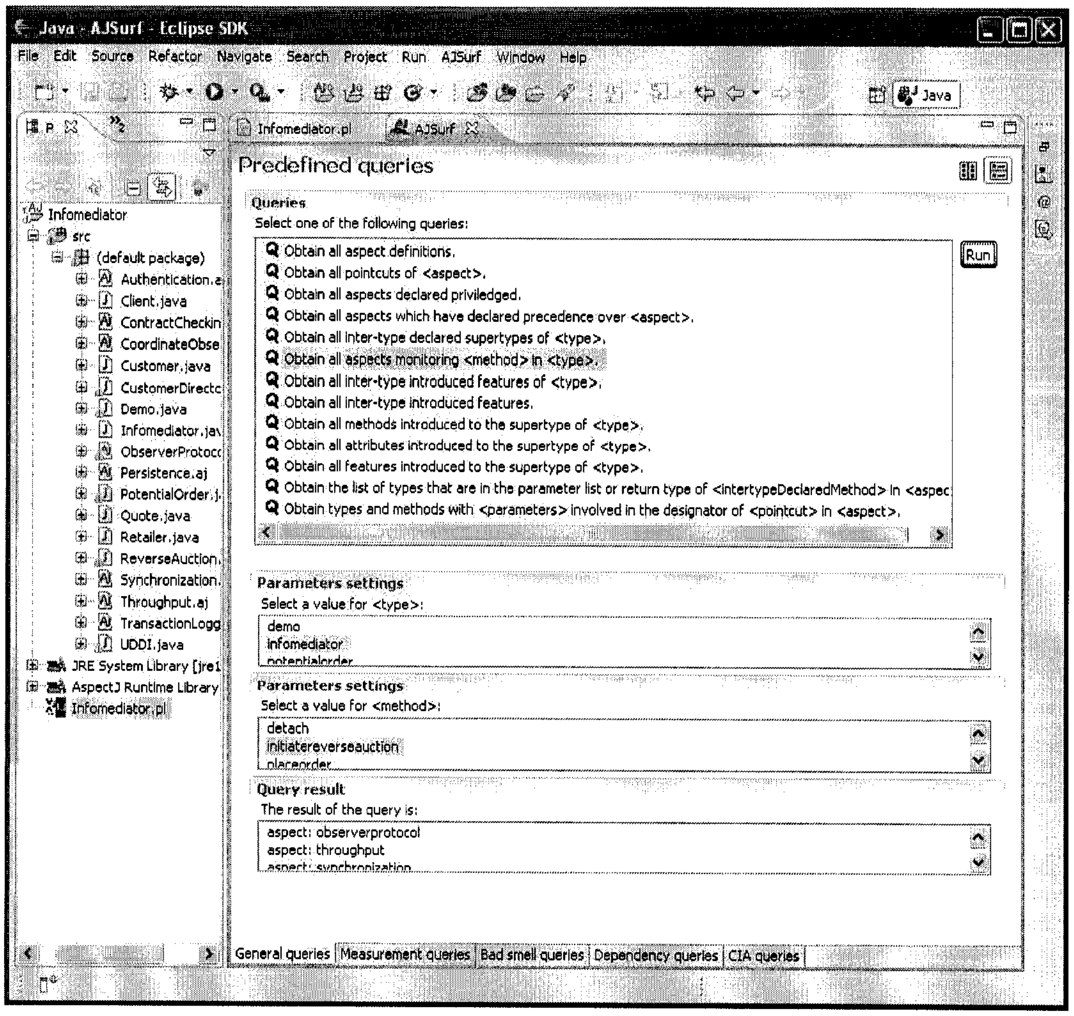
Figure 7: AJSurf: Choosing a destination folder for the Prolog database.

7.1 Deploying AJSurf

In order to deploy AJSurf, the user needs to first invoke Eclipse. Next, the user should select AJSurf. A window which allows possibly multiple .java/.aj files to be selected will pop-up (see Figure 6).

After the files have been selected, the user should validate the selection by clicking on Open button. A new window will pop-up and the user should select the destination folder and the name of Prolog database (see Figure 7).

AJSurf allows end-users to execute predefined parameterized queries. Figure 8



Infomediator
src
(default package)
Authentication.a
Client.java
ContractCheckin
CoordinateObse
Customer.java
CustomerDirectc
Demo.java
Infomediator.ja
ObserverProtoc
Persistence.aj
PotentialOrder.j
Quote.java
Retailer.java
ReverseAuction.
Synchronization.
Throughput.aj
TransactionLogg
UDDI.java
JRE System Library [jre1
AspectJ Runtime Library
Infomediator.pl

Predefined queries

Queries
Select one of the following queries:

Obtain all aspect definitions.
Obtain all pointcuts of <aspect>.
Obtain all aspects declared privileged.
Obtain all aspects which have declared precedence over <aspect>.
Obtain all inter-type declared supertypes of <type>.
Obtain all aspects monitoring <method> in <type>.
Obtain all inter-type introduced features of <type>.
Obtain all inter-type introduced features.
Obtain all methods introduced to the supertype of <type>.
Obtain all attributes introduced to the supertype of <type>.
Obtain all features introduced to the supertype of <type>.
Obtain the list of types that are in the parameter list or return type of <intertypeDeclaredMethod> in <aspec
Obtain types and methods with <parameters> involved in the designator of <pointcut> in <aspect>.

Parameters settings
Select a value for <type>:
demo
infomediator
potentialorder

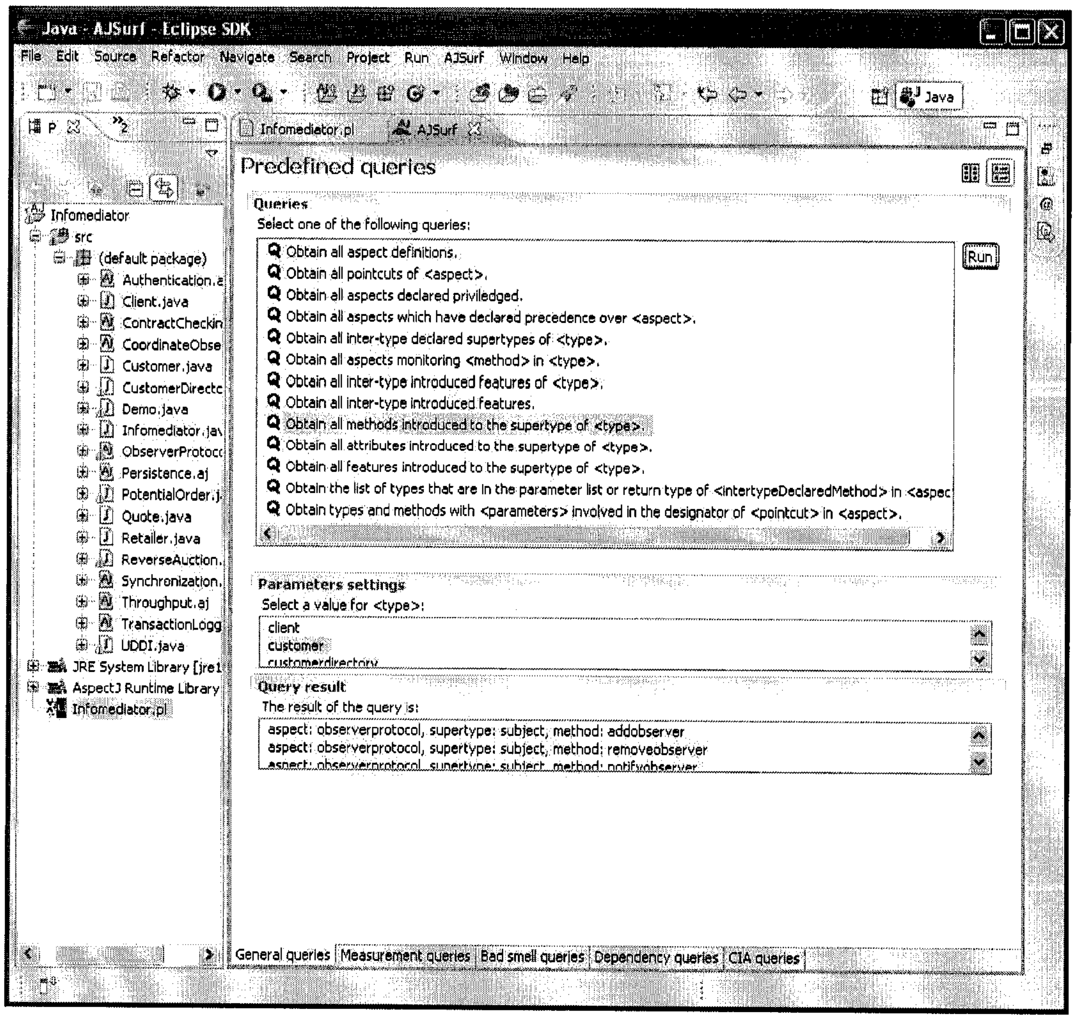
Parameters settings
Select a value for <method>:
detach
initiatereverseauction
placeorder

Query result
The result of the query is:
aspect: observerprotocol
aspect: throughput
aspect: synchronization

Figure 8: AJSurf Example of a parameterized query.

shows a screen capture of AJSurf with an executed parameterized query. The user selects one of the predefined queries, written in plain English, and then he/she should provide the parameters to be passed to the query. Lists of possible parameters are provided under the list of queries. In this example, the user wants to obtain all aspects advising method `initiatereverseauction()` in class `infomediator`.

Figure 9 shows another screen capture with the user obtaining the list of all



Predefined queries

Queries
Select one of the following queries:

- Obtain all aspect definitions.
- Obtain all pointcuts of <aspect>.
- Obtain all aspects declared priviledged.
- Obtain all aspects which have declared precedence over <aspect>.
- Obtain all inter-type declared supertypes of <type>.
- Obtain all aspects monitoring <method> in <type>.
- Obtain all inter-type introduced features of <type>.
- Obtain all inter-type introduced features.
- Obtain all methods introduced to the supertype of <type>.
- Obtain all attributes introduced to the supertype of <type>.
- Obtain all features introduced to the supertype of <type>.
- Obtain the list of types that are in the parameter list or return type of <intertypeDeclaredMethod> in <aspec
- Obtain types and methods with <parameters> involved in the designator of <pointcut> in <aspect>.

Parameters settings
Select a value for <type>:

client
customer
customerdirectory

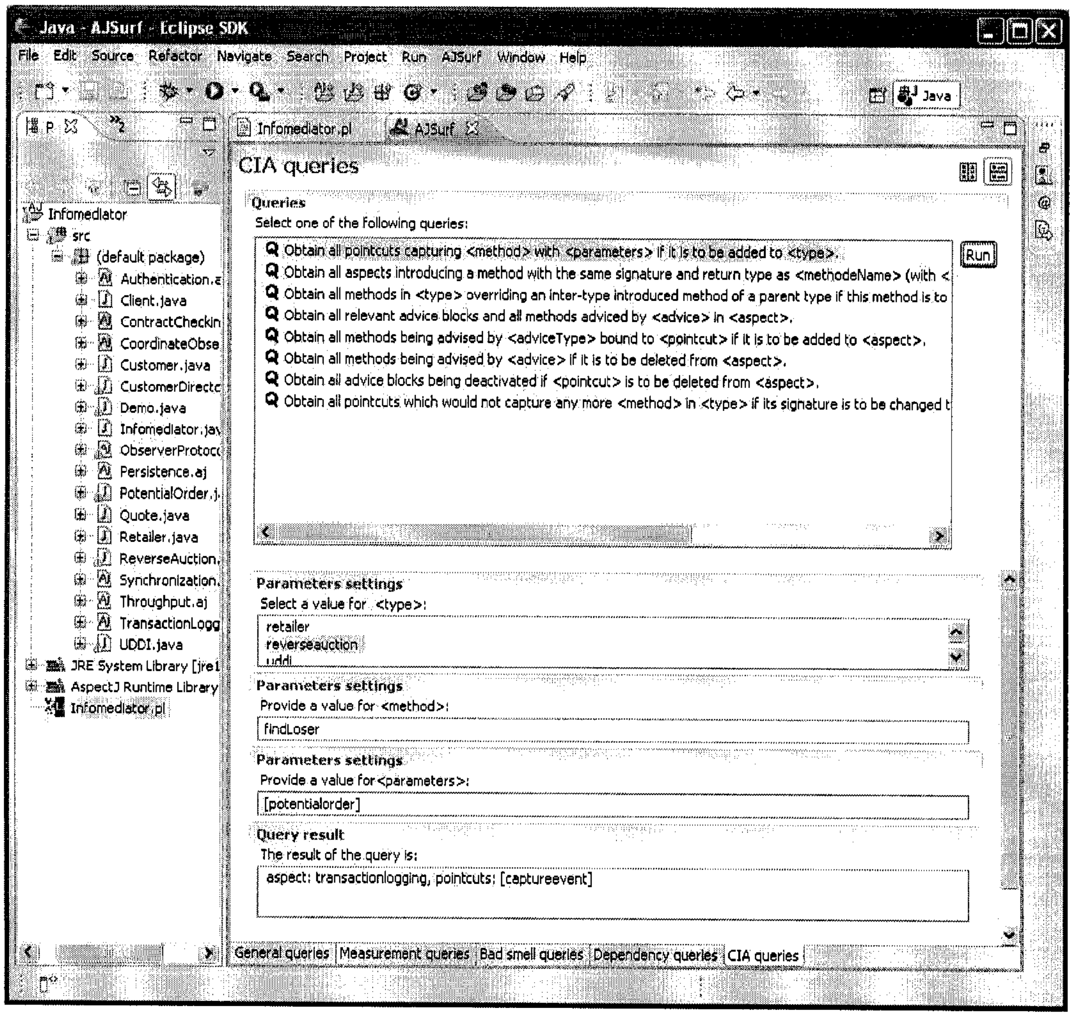
Query result
The result of the query is:

```
aspect: observerprotocol, supertype: subject, method: addobserver
aspect: observerprotocol, supertype: subject, method: removeobserver
aspect: observerprotocol, supertype: subject, method: notifyobserver
```

Figure 9: AJSurf Example of a parameterized query.

methods defined through inter-type declaration to the supertype of class `customer`.

Figure 10 shows a case where a user is performing change impact analysis. The user wants to add a method to the system, but before adding this method he wants to obtain the list of pointcuts that can capture this method.



CIA queries

Queries

Select one of the following queries:

- Obtain all pointcuts capturing <method> with <parameters> if it is to be added to <type>.
- Obtain all aspects introducing a method with the same signature and return type as <methodeName> (with <
- Obtain all methods in <type> overriding an inter-type introduced method of a parent type if this method is to
- Obtain all relevant advice blocks and all methods adviced by <advice> in <aspect>.
- Obtain all methods being advised by <adviceType> bound to <pointcut> if it is to be added to <aspect>.
- Obtain all methods being advised by <advice> if it is to be deleted from <aspect>.
- Obtain all advice blocks being deactivated if <pointcut> is to be deleted from <aspect>.
- Obtain all pointcuts which would not capture any more <method> in <type> if its signature is to be changed t

Parameters settings

Select a value for <type>:

retailer
reverseauction
uddi

Parameters settings

Provide a value for <method>:

findLoser

Parameters settings

Provide a value for<parameters>:

[potentialorder]

Query result

The result of the query is:

aspect: transactionlogging, pointcuts: [captureevent]

Figure 10: AJSurf Example of a parameterized query.

Chapter 8

Related work

In this chapter, we discuss related work and compare our work with similar approaches.

8.1 Related work

There is currently tool support to ease the comprehension process of both procedural [52], and object-oriented programs. For the latter, tools can be categorized according to the type of transformation and the artifacts they provide:

- Reverse engineering of source code to the design model (such as Poseidon-UML [5], and Eclipse-Omondo [4]).

- Reverse engineering of source code to graphs (data dependency graph, control dependency graph, formal concept analysis lattice).

- Knowledge extraction (facts and rules) from the source code to produce structural relations of elements which serve as a knowledge base over which queries can be executed.

Existing tool support for aspect-oriented systems can be categorized in three groups [37]:

Text-based tools They provide different views such as editors, outline, and package explorer.

Visualization-based tools They create aspectual relationship views (e.g calls, advice, advised-by) between aspects and classes.

Query-based tools They can be considered as a subset of text-based or visualization-based tools as they provide the result of a query either in text or in a visualization view.

In [13] the authors present a reverse engineering tool called Ciao. Ciao is a graphical navigating tool, which allows users to specify queries, generate graphs, interact with graph nodes, and to perform various graph analysis tasks in order to extract information from a software repository. The software repository is a collection of source code artifacts with all related documents, configuration management files, modification requests and manuals together with an associated database that describes the components and relationship among them. CQL is used as the query language associated with the repository. Ciao supports repositories which have

AERO style architecture (Attributes, Entity, Relationship, and Operator), and it has been designed for C and C++ program database and program difference database. Each node in the navigation graph corresponds to a query that generates a specific view on the system. The edges of the navigation graph represent historic dependencies between query views. However, the nodes in the navigation graph only indicate the type of query executed and for each query executed the corresponding graph is shown. To reconstruct the structural relationships that connect different queries on a path, one must compare their corresponding views.

In [41] the authors model static and dynamic information of an object-oriented system in terms of Prolog facts. Declarative queries are defined to allow filtering of the collected data and defining new queries. By running these queries, maintainers can have a view of the system at a higher level of abstraction for a better understanding.

SOUL is a logic meta-language based on Prolog which is implemented in Visual Work Smalltalk [53]. It provides a declarative framework that allows reasoning about the structure of Smalltalk programs based on the parse tree representation. This makes facts and rules to be independent from a particular base language. Moreover, facts and rules are collected based on basic relationships in the object-oriented system. High level relationships like design patterns can be expressed and then implemented in code. The declarative framework of SOUL consists of two layers of rules: basic layer and advanced layer. The basic layer includes representational and structural rules. The representational rules define object-oriented elements (classes, methods, and statements) in the logical meta-language using Smalltalk terms. These rules are the only parts of SOUL which are language dependent, the rest of the rules are language independent. Using Smalltalk terms facilitates the translation of object-oriented classes to logical facts, and only the relationships between the classes are formulated in rules on the meta-language. The structural rules are defined over the representational rules and formulate some other relationship in Smalltalk systems. Using these rules one can run basic queries on the system.

Lost [38] is an Eclipse plug-in query tool developed for code navigation and browsing for AspectJ programs, deploying a variant of the Object Query Language (OQL) developed by the authors. For its query language, end-users need to write the queries in the query editor area and an instant error feedback feature of the tool allows users to correct the errors while writing queries. Users of Lost need to know the query language, as there are no predefined queries available. This tool can be also used for navigation of Java programs. Like other Eclipse plug-ins, this tool deploys Eclipse IDE features like syntax highlighting, and auto-compilation.[1].

In [51] the author introduced a Java browser called JQuery as an Eclipse plug-in. The tool creates a database from the source code and provides an interface

[1]Very little else is provided in [38] on the nature of rules and queries and no other information seems to be available.

for the end-users to run queries. The query language used for this tool is a logic programming language called TyRuBa. Using this tool, users can run default (predefined) queries to extract information about their Java programs. Moreover, the tool allows users to write their own queries in order to obtain more information about the given Java code. One of the strengths of this tool is the ability to explore complex combinations of relationships through the declarative configuration interface. Users who only need the basic features do not need to know TyRuBa. However, users would have to know TyRuBa should they want to have more complex queries, as they would need to edit the existing queries or write new ones.

JTransformer [3] is a Prolog-based query and transformation engine for storing, analyzing and transforming fact-bases of Java source code. JTransformer creates an AST representation of a Java project, including the complete AST of method bodies as a Prolog database. Using JTransformer, developers can run powerful queries on the logic fact-base.

CodeQuest [25] is a source code querying tool which uses *safe Datalog* as its query language, mapping Datalog queries to a relational database system.

In [18] the authors present a prototype tool for analysis and performance optimization of Java programs called DeepWeaver-1. This tool is an extension of the abc AspectJ compiler [9] which has a declarative style query language, (Prolog/Datalog-like query language) to analyze and transform code within and across methods.

Eclipse IDE [1] provides different editors and views. Views are visual components allowing navigation through a hierarchy of information. Editors are components that allow editing or browsing a file. Views and editors provide different types of representation of the resources for the developers. AspectJ Development Tools (AJDT) provides support for aspect-oriented software development using AspectJ within the Eclipse IDE. Following is the summary of what is available for comprehension of AspectJ in Eclipse IDE:

Navigator view It provides a tree structure of all resources in project folder.

Package Explorer view It shows only the Java-specific or AspectJ-specific elements which were shown in Navigator view. The source folder and reference libraries of a project is shown in tree structure.

Hierarchy view It provides the inheritance hierarchy of objects in the projects.

Outline view It shows the layout of the file that is open in the editor area. For a Java class, it shows all its features (methods and variables), and for an aspect it shows its pointcuts, advice blocks, and introduced features.

Cross References view It shows how classes are advised by aspects.

Items 1-4 provide tree structures in which developers and maintainers can navigate to find the piece of information they are interested in. These views provide static information of classes and aspects in the projects, however the dependency between these objects is not shown. The Cross Reference view shows all methods that are monitored by an advice in the selected aspect. The same result can also be obtained by right clicking on the marker (the arrow) sign on the scroll bar beside the advice declaration in an aspect, and choosing *Advises* in the drop down menu. Moreover, for a specific method in a class one can find all advice blocks monitoring it by dragging and dropping the method from Outline view in the Cross References view. The same result can also be obtained by right clicking on the marker (the arrow) sign on the scroll bar beside the method declaration in a class, and choosing *AdvisedBy* in the drop down menu.

In [27] the authors propose an approach for change impact analysis of class hierarchy in order to reduce the retesting efforts. They consider changes which can occur at the level of data member, a member function, and inheritance relations. In order to identify the dependency relationship among class members, they introduce a member dependency graph (MDG). In this MDG, nodes correspond to classes and member functions, whereas edges correspond to inheritance relations, control dependencies and data dependencies. They define base cases for changes corresponding to object-oriented features such as : change to scope, changes related to dynamic binding, changes to data definition/use, changes of member functions, changes of data members, and changes of the inheritance hierarchy. For each case they identify the affected part in the MDG by identifying the firewall[2].

In [24] the authors identify the atomic changes for object-oriented programs. By defining partial ordering between the changes, they define dependencies between them. For each test defined in the system, a call graph is created where the nodes represent methods which were tested and the edges represent calling relationship between these methods. By analyzing the call graphs and the atomic changes, they identify: i) test cases that are affected by the atomic changes, ii) the atomic changes that affect a specific test case, iii) changes that do not affect any test, iv) code that was not covered by any test.

Chianti [40] is an Eclipse plug-in for change impact analysis of Java programs. It accepts a set of test suite, original version of code, and edited version of code as inputs and produces the set of changes. Dependencies between the changes are identified in order to predict the the affected part of the tests through analyzing the call graph related to each test suite.

In [55] the author presents a technique for change impact analysis of AspectJ systems based on program slicing while proposing to add arcs, and vertices (call dependence arc, parameter-in dependence arcs, actual-in vertices and formal-in vertices) to a system dependence graph (SDG) in order to present aspect-oriented system dependence graph (ASDG). He discusses the possibility of applying the graph to perform change impact analysis during software maintenance.

[2]Class firewall is a group of classes and their related test cases which are affected due to a change in class with which they have inheritance, aggregation or association relationship [32].

In [44] the authors propose an approach for analyzing the affect of aspect weaving and its propagation in aspect-oriented programs. They define aspect weaving as "possibility of changes in the sequence of statement execution or in the values of variables due to weaving." Based on this definition, they propose a technique similar to slicing which uses SPI (starting point of impact) as a criterion. In this definition, a SPI corresponds to a modification of fields of parameter objects, target objects or return values of a method in the `before` or `after` advice (for `around` advice, the authors also consider the modification of the value passed to the parameters of `proceed()` and the modification of the value returned by `proceed()`). By forward slicing the program dependency graph (which treats aspect weaving like method calls) based on these defined criteria, Shinomi and Tamai identify the affected parts of the program.

8.2 Comparison with similar approaches

Two different query mechanisms are often applied for information storage and extraction:

The first (implemented by Lost [38]) adopts predefined predicates and combines them using relational calculus. In this approach the predicates are stored in named sets and the relational calculus queries are translated to common relational algebra operations. The advantage of this approach is the speed and efficiency of the algorithm and the ease of adapting to a different persistent storage. The disadvantage is the limitation of its expressive power. The second approach (implemented by JQuery [51]) adopts a resolution inference mechanism to find the values of variables as they are resolved during unification, providing more expressiveness and power. There are also disadvantages with this approach including 1) the possibility of running into infinite loops, and 2) time and memory overhead caused by the nature of the reasoning algorithm.

We utilize a logic-based querying approach, to express the complex relationship among different entities in aspect-oriented systems. The transformation rules from AspectJ to Prolog facts can be easily extended in order to adopt a different source (AspectJ-like) language. Our approach provides comprehension, and allows maintainers to perform change impact analysis for aspect-oriented system. The type of knowledge that they inferred manually until now and thus difficult to collect (i.e. all advice blocks, relevant to a given advice, or all messages to which an object can respond) is now explicitly available through AJSurf.

Chapter 9

Conclusion and recommendations

9.1 Summary and conclusion

In this dissertation, we discussed an approach to support declarative (static) analysis of aspect-oriented programs, adopting AspectJ as a representative technology aiming at improving comprehension. Our approach is based on the transformation of source code into a set of facts and data, provided as a Prolog database over which queries can be executed. Declarative analysis allows us to extract complex information through its rich and expressive syntax. The type of knowledge provided through these queries is categorized in three main groups, such as bad smells, measurements, and more general queries that provide static information about the system. In order to perform change impact analysis, we have identified dependencies between software elements, and translated them in form of Prolog rules. Moreover, we have deployed a subset of atomic changes. Change impact analysis is performed by creating a relationship between the atomic changes and the dependencies. We have automated our approach and integrated all activities in a tool (called AJSurf) provided as an Eclipse plug-in. End-users can execute predefined parameterized queries in the form of Prolog goals. The correctness of tool depends on the correctness of the rules. For the case study, we have tested all rules to ensure that the end result of each query is the one expected.

9.2 Recommendations

Future developments may concentrate to address the following issues:

- Formally proofing the correctness of the transformation process from source code to Prolog facts.
- Extending the rules in order to capture changes in more join points including: object initializer, attribute accessor, withincode.
- Extending AJSurf in order to allow users to combine existing queries in order to obtain more complex information.
- Investigating the performance of the tool through benchmarking for very large systems.

Appendix A

Case study: Implementation and transformation

```
public abstract class Client {

    public String name = null;
    public String id = null;
    public Infomediator infomediator;

    public String getName() {
        return name;
    }
    public String getId() {
        return id;
    }
}
```

Table 53: Definition of class `Client`.

```
class(client,public).
abstractClass(client).
attribute(client,att_name,string,public,0).
attribute(client,att_id,string,public,0).
attribute(client,att_infomediator,infomediator,public,0).
method(client,getname,public,string,[ ],0).
method(client,getid,public,string,[ ],0).
```

Table 54: Prolog facts corresponding to class Client.

```
public class Customer extends Client {

        public Customer(String name, String id, Infomediator infomediator) {
                this.name = name;
                this.id = id;
                this.infomediator = infomediator;
                infomediator.attach(this);
        }
        public void buy(PotentialOrder po){
                System.out.println("Customer: " + "'"+ getName()+"'"+ " bought "+
                                po.getService()+" from "+"'"+
                                po.getWinner().getName()+"'"+ " retailer.");

        }
}
```

Table 55: Definition of class Customer.

```
class(customer,public).
extends(customer,client).
constructor(customer,public,[string, string, infomediator]).
method(customer,buy,public,void,[potentialorder],0).
sendMessage(customer,buy,[potentialorder],customer,getname,[]).
sendMessage(customer,buy,[potentialorder],potentialorder,getservice,[]).
sendMessage(customer,buy,[potentialorder],potentialorder,getwinner,[]).
sendMessage(customer,buy,[potentialorder],retailer,getname,[]).
```

Table 56: Prolog facts corresponding to class Customer.

```
import java.util.Vector;
public class Infomediator {
        private UDDI retailerDirectory;
        private CustomerDirectory customerDirectory;
        private Vector<PotentialOrder> potentialOrders;
        private ReverseAuction currentReverseAuction;
        private static Infomediator instance = null;
        private Infomediator(UDDI retailerDirectory, CustomerDirectory customerDirectory) {
           this.retailerDirectory = retailerDirectory;
           this.customerDirectory = customerDirectory;
           this.potentialOrders = new Vector<PotentialOrder>(); }
        public static Infomediator getInstance(UDDI retailerDirectory,
                                               CustomerDirectory customerDirectory) {
          if(instance == null) {
             instance = new Infomediator(retailerDirectory, customerDirectory); }
          return instance;
        }
        public void attach (Client client){
           if (client instanceof Retailer)
                retailerDirectory.attach((Retailer)client);
           else{
                customerDirectory.attach((Customer)client); }
        }
        public void detach (Client client){
           if (client instanceof Retailer)
                retailerDirectory.detach((Retailer)client);
           else
                customerDirectory.detach((Customer)client);
        }
        public synchronized PotentialOrder initiateReverseAuction(Customer customer,
                                                                  String service,
                                                                  String rule) {
           currentReverseAuction = new ReverseAuction(retailerDirectory, service, rule);
           PotentialOrder currentPO = new PotentialOrder(customer, service, rule);
           this.potentialOrders.addElement(currentPO);
           currentPO.setWinner(this.currentReverseAuction.initiateReverseAuction());
           return currentPO;
        }
        public void placeOrder(PotentialOrder po) {
           po.getWinner().placeOrder(po);
           this.potentialOrders.removeElement(po);
        }
        public void endReverseAuction() {
                this.currentReverseAuction.becomeComplete();
        }
        public UDDI getRetailerDirectory() {
                return retailerDirectory;
        } }
```

Table 57: Definition of class `Infomediator`.

```
class(infomediator,public).
attribute(infomediator,att_retailerDirectory,uddi,private,0).
attribute(infomediator,att_customerDirectory,customerdirectory,private,0).
attribute(infomediator,att_potentialOrders,vector,private,0).
attribute(infomediator,att_currentReverseAuction,reverseauction,private,0).
attribute(infomediator,att_instance,infomediator,private,1).
constructor(infomediator,private,[uddi,customerdirectory]).
method(infomediator,getinstance,public,infomediator,[uddi, customerdirectory],3).
method(infomediator,attach,public,void,[client],0).
sendMessage(infomediator,attach,[client],uddi,attach,[retailer]).
sendMessage(infomediator,attach,[client],customerdirectory,attach,[customer]).
method(infomediator,detach,public,void,[client],0).
sendMessage(infomediator,detach,[client],uddi,detach,[retailer]).
sendMessage(infomediator,detach,[client],customerdirectory,detach,[customer]).
method(infomediator,initiatereverseauction,public,potentialorder,[customer,string,string],0).
sendMessage(infomediator,initiatereverseauction,[customer,string,string],potentialorder,
        setwinner,[retailer]).
method(infomediator,placeorder,public,void,[potentialorder],0).
sendMessage(infomediator,placeorder,[potentialorder],potentialorder,getwinner,[]).
sendMessage(infomediator,placeorder,[potentialorder],retailer,placeorder,[potentialorder]).
method(infomediator,endreverseauction,public,void,[],0).
sendMessage(infomediator,endreverseauction,[],reverseauction,becomecomplete,[]).
method(infomediator,getretailerdirectory,public,uddi,[],0).
new(infomediator,getInstance,infomediator).
new(infomediator,initiatereverseauction,reverseauction).
new(infomediator,initiatereverseauction,potentialorder).
```

Table 58: Prolog facts corresponding to class `Infomediator`.

```
public class PotentialOrder {
        private Customer customer;
        private Retailer winner;
        private String service;
        private String rule;
        public PotentialOrder(Customer customer, String service, String rule){
           this.customer = customer;
           this.service = service;
           this.rule = rule;
        }
        public void setWinner(Retailer winner){
           this.winner = winner;
        }
        public Retailer getWinner() {
           return winner;
        }
        public Customer getCustomer() {
           return customer;
        }
        public String getService() {
           return service;
        }
}
```

Table 59: Definition of class `PotentialOrder`.

```
class(potentialorder,public).
attribute(potentialorder,att_customer,customer,private,0).
attribute(potentialorder,att_winner,retailer,private,0).
attribute(potentialorder,att_service,string,private,0).
attribute(potentialorder,att_rule,string,private,0).
constructor(potentialorder,public,[customer, string, string]).
method(potentialorder,setwinner,public,void,[retailer],0).
method(potentialorder,getwinner,public,retailer,[],0).
method(potentialorder,getcustomer,public,customer,[],0).
method(potentialorder,getservice,public,string,[],0).
```

Table 60: Prolog facts corresponding to class `PotentialOrder`.

```
public class Quote {
        private String quote;
        private String id;
        public Quote(String quote, String id){
           this.quote = quote;
           this.id = id;
        }
        public String getQuote() {
           return quote;
        }
        public String getId() {
           return id;
        }
        public String toString() {
                return this.getQuote() + "\t" + this.getId();
        }
}
```

Table 61: Definition of class Quote.

```
class(quote,public).
attribute(quote,att_quote,string,private,0).
attribute(quote,att_id,string,private,0).
constructor(quote,public,[string, string]).
method(quote,getquote,public,string,[ ],0).
method(quote,getid,public,string,[ ],0).
method(quote,tostring,public,string,[ ],0).
sendMessage(quote,tostring,[ ],quote,getquote,[ ]).
sendMessage(quote,tostring,[ ],quote,getid,[ ]).
```

Table 62: Prolog facts corresponding to class Quote.

```
public class Retailer extends Client {
        private String type;
        private boolean notification = false;
        private String service=null;
        public Retailer(String name, String id, String type, String service,
                        Infomediator infomediator) {
           this.name = name;
           this.id = id;
           this.type = type;
           this.service = service;
           this.infomediator = infomediator;
           infomediator.attach(this);
        }
        public String getQuote() {
           return "good quote";
        }
        public void placeOrder(PotentialOrder po) {
           System.out.println("Order placed by: "
                              + ((Retailer)po.getWinner()).getId()
                              + " for customer: "
                              + ((Customer)po.getCustomer()).name + "\n");
        }
        public void inform(){
           this.notification = true;
           System.out.println("\nWinner: " + this.toString());
        }
        public String getType() {
           return type;
        }

        public String getService() {
           return service;
        }

        public String toString() {
           return this.getId() + "\t" + this.getName() + "\t" + this.getType();
        }
        public boolean getNotification() {
           return notification;
        }
}
```

Table 63: Definition of class `Retailer`.

```
class(retailer,public).
extends(retailer,client).
attribute(retailer,att_type,string,private,0).
attribute(retailer,att_notification,boolean,private,0).
attribute(retailer,att_service,string,private,0).
constructor(retailer,public,[string, string, string, string, infomediator]).
method(retailer,getquote,public,string,[],0).
method(retailer,placeorder,public,void,[potentialorder],0).
sendMessage(retailer,placeorder,[potentialorder],potentialorder,getwinner,[]).
sendMessage(retailer,placeorder,[potentialorder],retailer,getid,[]).
sendMessage(retailer,placeorder,[potentialorder],potentialorder,getcustomer,[]).
method(retailer,inform,public,void,[],0).
sendMessage(retailer,inform,[],retailer,tostring,[]).
method(retailer,gettype,public,string,[],0).
method(retailer,getservice,public,string,[],0).
method(retailer,tostring,public,string,[],0).
sendMessage(retailer,tostring,[],retailer,getid,[]).
sendMessage(retailer,tostring,[],retailer,getname,[]).
sendMessage(retailer,tostring,[],retailer,gettype,[]).
method(retailer,getnotification,public,boolean,[],0).
```

Table 64: Prolog facts corresponding to class `Retailer`.

```
import java.util.*;
public class ReverseAuction {
        private UDDI retailerDirectory;
        private Vector retailerIds = new Vector();
        private Vector retailers = new Vector();
        private Vector quotes = new Vector();
        private String service = null;
        private String rule = null;
        private boolean isComplete = false;
        public ReverseAuction(UDDI retailerDirectory, String service, String rule) {
          this.retailerDirectory = retailerDirectory;
          this.service = service;
          this.rule = rule;    }
        public Retailer initiateReverseAuction() {
           Retailer winner = null;
           this.findCandidateRetailerIds(retailerIds, service, rule);
           this.retrieveCandidateRetailers();
           this.getQuote();
           winner = this.selectWinner();
           System.out.println(winner.toString());
           this.notifyWinner(winner.getId()); return winner; }
        public void findCandidateRetailerIds(Vector retailerIds, String service, String rule) {
           retailerDirectory.find(retailerIds, service, rule);    }
        public void retrieveCandidateRetailers() {
           for (int i=0; i< retailerIds.size(); i++) {
                String id = (String)retailerIds.get(i);
                Retailer r = retailerDirectory.retrieve(id);
                retailers.add(r);}        }
        public void getQuote() {
           for (int i=0; i< retailers.size(); i++) {
                Retailer r = (Retailer)retailers.get(i);
                String quote = r.getQuote();
                String id = r.getId();
                Quote q = new Quote(quote, id);
                quotes.addElement(q);} }
        public Retailer selectWinner() {
           String id = null; Retailer result = null;
           for (int i=0; i< quotes.size(); i++) {
                Quote quote = (Quote)quotes.get(i);
                if (quote.getQuote().equals("good quote")) {
                   id = quote.getId(); break;}}
           for (int i = 0; i< retailers.size(); i++) {
                Retailer r = (Retailer)retailers.get(i);
                if (r.getId().equals(id)) {
                   result = r;  break;}} return result;     }
        public void notifyWinner(String id){retailerDirectory.notifyWinner(id); }
        public void becomeComplete() {this.isComplete = true; }
        public boolean isComplete() {return isComplete; }}
```

Table 65: Definition of class `ReverseAuction`.

```
class(reverseauction,public).
attribute(reverseauction,att_retailerDirectory,uddi,private,0).
attribute(reverseauction,att_retailerIds,vector,private,0).
attribute(reverseauction,att_retailers,vector,private,0).
attribute(reverseauction,att_quotes,vector,private,0).
attribute(reverseauction,att_service,string,private,0).
attribute(reverseauction,att_rule,string,private,0).
attribute(reverseauction,att_isComplete,boolean,private,0).
constructor(reverseauction,public,[uddi, string, string]).
method(reverseauction,initiatereverseauction,public,retailer,[],0).
sendMessage(reverseauction,initiatereverseauction,[],reverseauction,
            retrievecandidateretailers,[]).
sendMessage(reverseauction,initiatereverseauction,[],reverseauction,findcandidateretailerids,
            [vector, string, string]).
sendMessage(reverseauction,initiatereverseauction,[],reverseauction,getquote,[]).
sendMessage(reverseauction,initiatereverseauction,[],reverseauction,selectwinner,[]).
sendMessage(reverseauction,initiatereverseauction,[],retailer,tostring,[]).
sendMessage(reverseauction,initiatereverseauction,[],retailer,getid,[]).
sendMessage(reverseauction,initiatereverseauction,[],reverseauction,notifywinner,[string]).
method(reverseauction,findcandidateretailerids,public,void,[vector, string, string],0).
sendMessage(reverseauction,findcandidateretailerids,[vector, string, string],uddi,find,
            [vector, string, string]).
method(reverseauction,retrievecandidateretailers,public,void,[],0).
sendMessage(reverseauction,retrievecandidateretailers,[],uddi,retrieve,[string]).
method(reverseauction,getquote,public,void,[],0).
sendMessage(reverseauction,getquote,[],retailer,getid,[]).
sendMessage(reverseauction,getquote,[],retailer,getquote,[]).
method(reverseauction,selectwinner,public,retailer,[],0).
sendMessage(reverseauction,selectwinner,[],quote,getid,[]).
sendMessage(reverseauction,selectwinner,[],quote,getquote,[]).
sendMessage(reverseauction,selectwinner,[],retailer,getid,[]).
method(reverseauction,notifywinner,public,void,[string],0).
sendMessage(reverseauction,notifywinner,[],uddi,notifywinner,[string]).
method(reverseauction,becomecomplete,public,void,[],0).
method(reverseauction,iscomplete,public,boolean,[],0).
new(reverseauction,getQuote,quote).
```

Table 66: Prolog facts corresponding to class `ReverseAuction`.

```
import java.util.*;
public class UDDI {
        Vector<Retailer> retailers = new Vector<Retailer>();
        public void attach (Retailer retailer){
           retailers.addElement(retailer);
        }
        public void detach (Retailer retailer) {
           retailers.removeElement(retailer);
        }
        public void find(Vector collection, String service, String rule) {
           for (int i=0; i< retailers.size(); i++) {
               Retailer r = (Retailer)retailers.get(i);
               if (r.getType().equals(rule) && r.getService().equals(service)) {
                   collection.addElement(r.getId());
               }  }
        }
        public Retailer retrieve(String id) {
           Retailer result = null;
           for (int i=0; i< retailers.size(); i++) {
                Retailer r = (Retailer)retailers.get(i);
                if (r.getId().equals(id)) {
                   result = r;
                   break;
                }  }
           return result;
        }
        public void notifyWinner(String id){
           for (int i=0; i< retailers.size(); i++) {
                Retailer r = (Retailer)retailers.get(i);
                if (r.getId().equals(id)) {
                   r.inform();
                   break;
                }  }
        }
        public void bind() {
           System.out.println("--bind: request quote" + "\n");
        }
        public void displayAll() {
           System.out.println("\nRegistered Retailers:\n");
           System.out.println("ID" + "\t" + "Name" + "\t\t\t" + "Type");
           for (int i=0; i< retailers.size(); i++) {
                Retailer r = (Retailer)retailers.get(i);
                System.out.println(r.toString());
           }
        }
}
```

Table 67: Definition of class UDDI.

```
class(uddi,public).
attribute(uddi,att_retailers,vector,package,0).
method(uddi,attach,public,void,[retailer],0).
method(uddi,detach,public,void,[retailer],0).
method(uddi,find,public,void,[vector, string, string],0).
sendMessage(uddi,find,[vector, string, string],retailer,gettype,[]).
sendMessage(uddi,find,[vector, string, string],retailer,getservice,[]).
method(uddi,retrieve,public,retailer,[string],0).
sendMessage(uddi,retrieve,[string],retailer,getid,[]).
method(uddi,notifywinner,public,void,[string],0).
sendMessage(uddi,notifywinner,[string],retailer,getid,[]).
sendMessage(uddi,notifywinner,[string],retailer,inform,[]).
method(uddi,bind,public,void,[],0).
method(uddi,displayall,public,void,[],0).
sendMessage(uddi,displayall,[],retailer,tostring,[]).
```

Table 68: Prolog facts corresponding to class `UDDI`.

```
import java.util.*;
public aspect Authentication issingleton(){
        public static int numberOfActiveClients = 0;
        public static int maxAllowableClients = 15;
        private Vector<Client> authenticatedClients = new Vector<Client>();
        private boolean login() {
           boolean result = false;
           if (numberOfActiveClients < maxAllowableClients) {
                System.out.println("authentication OK");
                System.out.println("Active clients: " + numberOfActiveClients + "\n");
                numberOfActiveClients++;
                result = true;}
           else {
                System.out.println("Server full");
                result = false; }
           return result;
        }
        private void logout() {
           numberOfActiveClients--;}
        pointcut attach(Client client): execution (public void Infomediator.attach(..))
                                     &&args(client);
        void around(Client client): attach (client){
           if (login()){
                authenticatedClients.addElement(client);
                proceed(client);
           }
           else{
                System.out.println("There is no space to log in for this auction,
                                    You should try a new auction later...");
                System.exit(0); }
        }
        pointcut dettach(Client client): execution (public void Infomediator.detach(..)) &&
                                     args(client);
        after(Client client): dettach (client){
           for(int i = 0; i< authenticatedClients.size(); i++ ){
                Client currentClient = (Client)authenticatedClients.elementAt(i);
                if(currentClient == client){
                        logout();
                        authenticatedClients.removeElementAt(i);}
                }}
}
```

Table 69: Definition of aspect Authentication.

```
aspect(authentication,public).
method(authentication,login,private,boolean,[],0).
method(authentication,logout,private,void,[],0).
attribute(authentication,att_numberofactiveclients,int,public,1).
attribute(authentication,att_maxallowableclients,int,public,1).
attribute(authentication,att_authenticatedclients,vector,private,0).
pointcutdesig(8469441,authentication,null,persingleton,[null]).
pointcut(authentication,attach,[client],package,0).
pointcutdesig(26789619,authentication,attach,execution,[public,void,infomediator,attach,[any]]).
pointcutdesig(19762893,authentication,attach,args,[client]).
pointcut(authentication,dettach,[client],package,0).
pointcutdesig(4812898,authentication,dettach,execution,[public,void,infomediator,detach,[any]]).
pointcutdesig(24211360,authentication,dettach,args,[client]).
triggerAdvice(authentication,around,5734522,[client],void).
advicePointcutMap(authentication,around,5734522,attach).
triggerAdvice(authentication,after,29782600,[client],void).
advicePointcutMap(authentication,after,29782600,dettach).
advisedBy(infomediator,attach,[client],authentication,around,5734522,[client],attach).
advisedBy(infomediator,detach,[client],authentication,after,29782600,[client],dettach).
```

Table 70: Prolog facts corresponding to aspect `Authentication`.

```
public privileged aspect ContractChecking {
        pointcut RetailerExistanceChecker(ReverseAuction object):
                                execution (* ReverseAuction.selectWinner(..)) &&
                                target(object);

        Retailer around (ReverseAuction object):RetailerExistanceChecker(object){
          if (object.retailerDirectory.retailers.isEmpty()){
              System.out.println("There is no retailer registered for this auction,
                                 You should try a new auction later...");
              System.exit(0);
          }
          Retailer winner = proceed(object);
          if (winner == null){
              System.out.println("There is no winner retailer for this auction,
                                 You should try a new auction later...");
              System.exit(0);
          }
          return winner;
        }
        pointcut PotentialOrderCleaner(Infomediator infomediator, Client client):
                                            execution(* Infomediator.detach(..)) &&
                                            args(client) &&
                                            target(infomediator);
        after(Infomediator infomediator, Client client):
                                     PotentialOrderCleaner(infomediator, client){
          if (client instanceof Customer){
              infomediator.potentialOrders.trimToSize();
              for(int i=0; i< infomediator.potentialOrders.size(); i++){
                     PotentialOrder currentPo =
                     (PotentialOrder)infomediator.potentialOrders.elementAt(i);
                     if(currentPo.getCustomer() == client)
                     infomediator.potentialOrders.removeElementAt(i);
              } }}
}
```

Table 71: Definition of aspect ContractChecking.

```
aspect(contractchecking,public).
privilegedAspect(contractchecking).
pointcut(contractchecking,retailerexistancechecker,[reverseauction],package,0).
pointcutdesig(26625789,contractchecking,retailerexistancechecker,execution,
            [public,every,reverseauction,selectwinner,[any]]).
pointcutdesig(17743384,contractchecking,retailerexistancechecker,target,[object]).
pointcut(contractchecking,potentialordercleaner,[infomediator, client],package,0).
pointcutdesig(14828347,contractchecking,potentialordercleaner,execution,
            [public,every,infomediator,detach,[any]]).
pointcutdesig(11265620,contractchecking,potentialordercleaner,args,[client]).
pointcutdesig(27173235,contractchecking,potentialordercleaner,target,[infomediator]).
triggerAdvice(contractchecking,around,24749215,[reverseauction],retailer).
advicePointcutMap(contractchecking,around,24749215,retailerexistancechecker).
triggerAdvice(contractchecking,after,19333383,[infomediator, client],void).
advicePointcutMap(contractchecking,after,19333383,potentialordercleaner).
used(contractchecking,19333383,[infomediator,client],potentialorder,getcustomer,[]).
advisedBy(infomediator,detach,[client],contractchecking,after,19333383,
        [infomediator,client],potentialordercleaner).
advisedBy(reverseauction,selectwinner,[],contractchecking,around,24749215,
        [reverseauction],retailerexistancechecker).
```

Table 72: Prolog facts corresponding to aspect `ContractChecking`.

```
privileged public aspect CoordinateObserver extends ObserverProtocol {
        declare parents: Retailer implements Observer;
        declare parents: Customer implements Subject;
        private int Retailer.NumberSold=0;
        public void Retailer.notifyOfchange(Subject s, PotentialOrder po){
                NumberSold++;
                System.out.println("' "+po.getWinner().getName()+" '" +" sold "+
                                NumberSold + " "+po.getService()+ " till now");
        }
        protected pointcut subjectChange(Subject s,PotentialOrder po):
                                execution (* Customer.buy(PotentialOrder)) &&
                                target(s)&& args(po);
        protected pointcut findObservers(Infomediator info, Subject customer,
                                                String service, String rule):
        execution (* Infomediator.initiateReverseAuction( Customer , String , String ))

}
```

Table 73: Definition of aspect `CoordinateObserver`.

```
aspect(coordinateobserver,package).
privilegedAspect(coordinateobserver).
extends(coordinateobserver,observerprotocol).
declareParent(coordinateobserver,retailer,observer).
implements(retailer,observer).
declareParent(coordinateobserver,customer,subject).
implements(customer,subject).
introducedMethod(coordinateobserver,retailer,notifyofchange,public,void,
                  [subject, potentialorder],0).
sendMessage(retailer,notifyofchange,[subject, potentialorder],potentialorder,
             getwinner,[]).
sendMessage(retailer,notifyofchange,[subject, potentialorder],retailer,getname,[]).
introducedAtt(coordinateobserver,retailer,numbersold,int,private,0).
pointcut(coordinateobserver,subjectchange,[subject,potentialorder],protected,0).
pointcutdesig(1412294,coordinateobserver,subjectchange,execution,[public,every,customer,
              buy,[potentialorder]]).
pointcutdesig(13452612,coordinateobserver,subjectchange,target,[subject]).
pointcutdesig(8287698,coordinateobserver,subjectchange,args,[potentialorder]).
pointcut(coordinateobserver,findobservers,[infomediator,subject,string,string],protected,0).
pointcutdesig(6901522,coordinateobserver,findobservers,execution,
              [public,every,infomediator,initiatereverseauction,[customer,string,string]]).
pointcutdesig(29769356,coordinateobserver,findobservers,target,[infomediator]).
pointcutdesig(3431235,coordinateobserver,findobservers,args,[customer, service, rule]).
```

Table 74: Prolog facts corresponding to aspect `CoordinateObserver`.

```
import java.util.*;
public abstract aspect ObserverProtocol {
        public interface Observer {
                public void notifyOfchange(Subject s,PotentialOrder po);
        }

        protected interface Subject { }
        private List Subject.observers=new LinkedList();
        public void Subject.addObserver(Observer o){
           this.observers.add(o);
        }
        public void Subject.removeObserver(Observer o){
           this.observers.remove(o);
        }
        private synchronized void Subject.notifyObserver(PotentialOrder po){
           Iterator iter = this.observers.iterator();
           int i=0;
           while ( iter.hasNext() ) {
                Observer temp=(Observer)iter.next();
                if ( ((Retailer)temp).getNotification()== true ){
                        temp.notifyOfchange(this, po);
                        this.removeObserver( ((Retailer)temp) );}}
        }
        protected abstract pointcut subjectChange(Subject s,PotentialOrder po);
        after(Subject s,PotentialOrder po): subjectChange( s, po){
           s.notifyObserver(po);
        }
        protected abstract pointcut findObservers
                        (Infomediator info, Subject s, String service, String rule);
        after(Infomediator info, Subject s, String service, String rule):
                                        findObservers(info, s,  service,  rule ){
           Vector tempRetailer=info.getRetailerDirectory().retailers;
           for(int i=0 ;i < tempRetailer.size();i++ ){
                if (((Retailer)tempRetailer.get(i)).getService()== service
                    && ((Retailer)tempRetailer.get(i)).getType()==rule)
                        s.addObserver((Retailer)tempRetailer.get(i));}
        }
}
```

Table 75: Definition of aspect `ObserverProtocol`.

```
aspect(observerprotocol,public).
abstractAspect(observerprotocol).
interface(subject,protected).
interface(observer,public).
method(observer,notifyOfchange,public,void,[subject, potentialorder],0).
introducedMethod(observerprotocol,subject,addobserver,public,void,[observer],0).
introducedMethod(observerprotocol,subject,removeobserver,public,void,[observer],0).
introducedMethod(observerprotocol,subject,notifyobserver,private,void,
                [potentialorder],0).
sendMessage(subject,notifyobserver,[potentialorder],retailer,getnotification,[]).
sendMessage(subject,notifyobserver,[potentialorder],observer,notifyOfchange,
            [subject,potentialorder]).
sendMessage(subject,notifyobserver,[potentialorder],subject,removeobserver,[observer]).
introducedAtt(observerprotocol,subject,observers,list,private,0).
pointcut(observerprotocol,subjectchange,[subject,potentialorder],protected,1).
pointcut(observerprotocol,findobservers,[infomediator, subject, string, string],
    protected,1).
triggerAdvice(observerprotocol,after,21067408,[subject, potentialorder],void).
advicePointcutMap(observerprotocol,after,21067408,subjectchange).
used(observerprotocol,21067408,[subject,potentialorder],subject,notifyobserver,
    [potentialorder]).
triggerAdvice(observerprotocol,after,13563633,[infomediator, subject, string, string],void).
advicePointcutMap(observerprotocol,after,13563633,findobservers).
used(observerprotocol,13563633,[infomediator, subject, string, string],infomediator,
    getretailerdirectory,[]).
used(observerprotocol,13563633,[infomediator, subject, string, string],retailer,
    getservice,[]).
used(observerprotocol,13563633,[infomediator, subject, string, string],retailer,
    gettype,[]).
used(observerprotocol,13563633,[infomediator, subject, string, string],subject,
    addobserver,[observer]).
advisedBy(infomediator,initiatereverseauction,[customer,string,string],
          observerprotocol,after,13563633,[infomediator,subject,string,string],findobservers).
advisedBy(customer,buy,[potentialorder],observerprotocol,
          after,21067408,[subject,potentialorder],subjectchange).
```

Table 76: Prolog facts corresponding to aspect `ObserverProtocol`.

```
public privileged aspect Persistence {
        pointcut persisted(Infomediator object):(execution (* Infomediator.attach(..)) ||
                                                execution (* Infomediator.detach(..))) &&
                                                this(object);
        after(Infomediator object): persisted(object) {
                System.out.println("<" + thisJoinPoint + ">");
        }
}
```

Table 77: Definition of aspect `Persistence`.

```
aspect(persistence,public).
privilegedAspect(persistence).
pointcut(persistence,persisted,[infomediator],package,0).
pointcutdesig(2443549,persistence,persisted,execution,[public,every,infomediator,
            attach,[any]]).
pointcutdesig(31662978,persistence,persisted,execution,[public,every,infomediator,
            detach,[any]]).
pointcutdesig(19939395,persistence,persisted,this,[object]).
triggerAdvice(persistence,after,13305839,[infomediator],void).
advicePointcutMap(persistence,after,13305839,persisted).
advisedBy(infomediator,attach,[client],persistence,after,13305839,[infomediator],
         persisted).
advisedBy(infomediator,detach,[client],persistence,after,13305839,[infomediator],
         persisted).
```

Table 78: Prolog facts corresponding to aspect `Persistence`.

```
import java.util.*;
public privileged aspect Synchronization {
        public static Vector waitingClients = new Vector();
        public static int activeClients = 0;

        pointcut initiatingAuction(Infomediator object):
                                    call (* Infomediator.initiateReverseAuction(..)) &&
                                    target(object);

        PotentialOrder around(Infomediator object): initiatingAuction(object){
                PotentialOrder result = null;
                if (waitingClients.isEmpty() && activeClients == 0){
                        activeClients++;
                        result = proceed(object);
                        activeClients--;
                }
                else
                  try {
                        wait();
                        proceed(object);
                      }
                  catch (Exception exception) {
                        exception.printStackTrace();
                  }
                return result; }
}
```

Table 79: Definition of aspect `Synchronization`.

```
privilegedAspect(synchronization).
aspect(synchronization,public).
attribute(synchronization,att_waitingclients,vector,public,1).
attribute(synchronization,att_activeclients,int,public,1).
pointcut(synchronization,initiatingauction,[infomediator],package,0).
pointcutdesig(12743356,synchronization,initiatingauction,call,[public,every,
            infomediator,initiatereverseauction,[any]]).
pointcutdesig(33212498,synchronization,initiatingauction,target,[object]).
triggerAdvice(synchronization,around,24480977,[infomediator],potentialorder).
advicePointcutMap(synchronization,around,24480977,initiatingauction).
advisedBy(infomediator,initiatereverseauction,[customer,string,string],synchronization,
          around,24480977,[infomediator],initiatingauction).
```

Table 80: Prolog facts corresponding to aspect `Synchronization`.

```
public aspect Throughput {

      private int requestingCustomers = 0;
      private int terminatingCustomers = 0;
      pointcut requestingCustomers(): call (* Infomediator.initiateReverseAuction(..));

      pointcut terminatingCustomers():
                              execution (* Infomediator.initiateReverseAuction(..));
      before(): requestingCustomers() {
            System.out.println("Requesting auction: " +
                              ++requestingCustomers + "\n");
      }

      after(): terminatingCustomers() {
            System.out.println("Terminating auction: " +
                              ++terminatingCustomers + "\n");
      }
}
```

Table 81: Definition of aspect `Throughput`.

```
aspect(throughput,public).
attribute(throughput,att_requestingcustomers,int,private,0).
attribute(throughput,att_terminatingcustomers,int,private,0).
pointcut(throughput,requestingcustomers,[],package,0).
pointcutdesig(29499086,throughput,requestingcustomers,call,[public,every,infomediator,
        initiatereverseauction,[any]]).
pointcut(throughput,terminatingcustomers,[],package,0).
pointcutdesig(24422114,throughput,terminatingcustomers,execution,[public,every,
        infomediator,initiatereverseauction,[any]]).
triggerAdvice(throughput,before,24668732,[],void).
advicePointcutMap(throughput,before,24668732,requestingcustomers).
triggerAdvice(throughput,after,8818963,[],void).
advicePointcutMap(throughput,after,8818963,terminatingcustomers).
advisedBy(infomediator,initiatereverseauction,[customer,string,string],throughput,
        before,24668732,[],requestingcustomers).
advisedBy(infomediator,initiatereverseauction,[customer,string,string],throughput,
        after,8818963,[],terminatingcustomers).
```

Table 82: Prolog facts corresponding to aspect `Throughput`.

```
import java.util.*;
public privileged aspect TransactionLogging {
    pointcut loggableAuction(ReverseAuction object):
                        execution (* ReverseAuction.initiateReverseAuction(..)) &&
                        target(object);
    pointcut loggable(Object o) : (call (* Retailer.placeOrder(..)) ||
                            execution (* Customer.buy(..))) && target(o);
    pointcut captureCreatobject(Object o):
                    (execution (Customer.new(..))||
                     execution ( Retailer.new(..))||
                     execution( * ReverseAuction.initiateReverseAuction())) && target(o);
    pointcut captureEvent(): withincode( * Infomediator.initiateReverseAuction(..)) &&
                        call(* *.*(..));
    after():captureEvent(){ System.out.println("**<"+ thisJoinPoint+">**"); }
    after(ReverseAuction object): loggableAuction(object) {
      System.out.println("\n\n\nTransaction log:");
      System.out.println("\nCandidate Retailer ids:\n");
      for (int i=0; i< object.retailerIds.size(); i++) {
          String s = (String)object.retailerIds.get(i);
          System.out.println(s);     }
      System.out.println("\nCandidate Retailers:\n");
      for (int i=0; i< object.retailers.size(); i++) {
          Retailer r = (Retailer)object.retailers.get(i);
          System.out.println(r.toString());  }
      System.out.println("\nQuotes:\n");
      for (int i=0; i< object.quotes.size(); i++) {
          Quote q = (Quote)object.quotes.get(i);
          System.out.println(q.toString());  }
      System.out.println(); }
    after (Object o): loggable( o) {
       Calendar cal = Calendar.getInstance(TimeZone.getDefault());
       String DATE_FORMAT = "yyyy-MM-dd HH:mm:ss";
       java.text.SimpleDateFormat sdf = new java.text.SimpleDateFormat(DATE_FORMAT);
       sdf.setTimeZone(TimeZone.getDefault());
       if(o instanceof Retailer )
          System.out.println("Order placed at : " + sdf.format(cal.getTime()));
       if(o instanceof Customer )
          System.out.println("Item purchased at : " + sdf.format(cal.getTime())); }
    after(Object o) : captureCreatobject(o){
      if(o instanceof Customer )
          System.out.println("*** "+((Customer)o).getName()+ "
                              Registerde as a Customer***");
      if(o instanceof Retailer )
          System.out.println("***"+((Retailer)o).getName()+
                            " Registerde as a Retailer***");
          if(o instanceof ReverseAuction )
          System.out.println("*** Auction Started***");}
}
```

Table 83: Definition of aspect `TransactionLogging`.

```
aspect(transactionlogging,public).
privilegedAspect(transactionlogging).
pointcut(transactionlogging,loggableauction,[reverseauction],package,0).
pointcutdesig(16412836,transactionlogging,loggableauction,execution,
        [public,every,reverseauction,initiatereverseauction,[any]]).
pointcutdesig(33114244,transactionlogging,loggableauction,target,[object]).
pointcut(transactionlogging,loggable,[object],package,0).
pointcutdesig(17303670,transactionlogging,loggable,call,[public,every,retailer,
        placeorder,[any]]).
pointcutdesig(6533862,transactionlogging,loggable,execution,[public,every,customer,
        buy,[any]]).
pointcutdesig(24522695,transactionlogging,loggable,target,[object]).
pointcut(transactionlogging,capturecreatobject,[object],package,0).
pointcutdesig(3862294,transactionlogging,capturecreatobject,execution,[public,customer,new,[any]]).
pointcutdesig(19318506,transactionlogging,capturecreatobject,execution,[public,retailer,
        new,[any]]).
pointcutdesig(13452238,transactionlogging,capturecreatobject,execution,[public,every,
        reverseauction,initiatereverseauction,[]]).
pointcutdesig(2011334,transactionlogging,capturecreatobject,target,[object]).
pointcut(transactionlogging,captureevent,[],package,0).
pointcutdesig(15184882,transactionlogging,captureevent,withincode,[public,every,
        infomediator,initiatereverseauction,[any]]).
pointcutdesig(30869925,transactionlogging,captureevent,call,[public,every,every,every,
        [any]]).
triggerAdvice(transactionlogging,after,12217363,[],void).
advicePointcutMap(transactionlogging,after,12217363,captureevent).
triggerAdvice(transactionlogging,after,17666385,[reverseauction],void).
advicePointcutMap(transactionlogging,after,17666385,loggableauction).
triggerAdvice(transactionlogging,after,29321385,[object],void).
advicePointcutMap(transactionlogging,after,29321385,loggable).
triggerAdvice(transactionlogging,after,24145591,[object],void).
advicePointcutMap(transactionlogging,after,24145591,capturecreatobject).
advisedBy(reverseauction,initiatereverseauction,[],transactionlogging,after,
        12217363,[],captureevent).
advisedBy(potentialorder,setwinner,[retailer],transactionlogging,after,
        12217363,[],captureevent).
advisedBy(customer,constructor,[string,string,infomediator],transactionlogging,after,
        24145591,[object],capturecreatobject).
advisedBy(retailer,constructor,[string,string,string,string,infomediator],
        transactionlogging,after,24145591,[object],capturecreatobject).
advisedBy(reverseauction,initiatereverseauction,[],transactionlogging,after,24145591,
        [object],capturecreatobject).
advisedBy(reverseauction,initiatereverseauction,[],transactionlogging,after,17666385,
        [reverseauction],loggableauction).
advisedBy(customer,buy,[potentialorder],transactionlogging,after,29321385,[object],
        loggable).
advisedBy(retailer,placeorder,[potentialorder],transactionlogging,after,29321385,[object],
        loggable).
```

Table 84: Prolog facts corresponding to aspect `TransactionLogging`.

Appendix B

Tool installation

Prerequisites:

1. Prolog should be installed on the user's computer.
2. The environment variable PLBASE should exist and point to the installation folder of Prolog (for example `C:\Program Files\pl`).
3. The environment variables CLASSPATH and Path should be modified in order to point to two more folders. That is, in addition to their values the following should be added: `;%PLBASE%\lib\jpl.jar;%PLBASE%\bin`

Installation:

The installation folder contains four .jar files namely:

1. ajsurf_1.0.0.jar

2. jpl_1.0.0.jar

3. aspectjtools_1.0.0.jar

4. osgi_1.0.0.jar

Place these files in the plugin folder of Eclipse.

Settings:

The first time Eclipse runs after AJSurf has been installed, the following steps are to be performed:

1. Run Eclipse.

2. Go to Window menu and choose Customize Perspective. In tab Commands, check AJSurf and click on OK.

Bibliography

[1] Eclipse website. `http://www.eclipse.org/`

[2] Jpl - a java interface to prolog.
`http://www.swi-prolog.org/packages/jpl/java_api/index.html`

[3] JTransformer Framework website.
`http://roots.iai.uni-bonn.de/research/jtransformer/`

[4] Omondo website. `http://www.eclipsedownload.com/`

[5] Poseidon UML website. `http://jspwiki.org/wiki/PoseidonUML`

[6] International standard. *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 (Revision of IEEE Std 1219-1998)*, 2006.

[7] Shawn A.Bohner. Software change impacts - an evolving perspective. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, 2002.

[8] Robert Arnold and Shawn Bohner. *Software change impact analysis.* Wiley-IEEE Computer Society, 1996.

[9] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible aspectj compiler. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, 2005.

[10] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the International Conference on Software Engineering (ICSE) track on The Future of Software Engineering*, 2000.

[11] Lionel C. Briand, Yvan Labiche, Leeshawn O'Sullivan, and Mike Sówka. Automated impact analysis of uml models. *Journal of Systems and Software*, 2006.

[12] Lucas Carlson and Leonard Richardson. *Ruby Cookbook (Cookbooks (O'Reilly)).* O'Reilly Media, Inc., 2006.

[13] Yih-Farn R. Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: a graphical navigator for software and document repositories. In *Proceedings of the 11th International Conference on Software Maintenance (ICSM)*, 1995.

[14] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-oriented programming with AspectJ and the Eclipse AspectJ Development Tools.* AAddison-Wesley Professional, Boston, MA, 2004.

[15] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *Java(TM) Developer's Guide to Eclipse, The (2nd Edition).* Addison-Wesley Professional, 2004.

[16] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 2003.

[17] Traces Thomas Eisenbarth. Feature-driven program understanding using concept analysis of execution traces. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC)*, 2001.

[18] Henry Falconer, Paul H. J. Kelly, David M. Ingram, Michael R. Mellor, Tony Field, and Olav Beckmann. A declarative framework for analysis and optimization. In *Proceedings of the 16th International Conference on Compiler Construction (ETAPS CC)*, 2007.

[19] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proceedings of the OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2000.

[20] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the design of existing code.* Addison-Wesley Professional, 1999.

[21] Daniel Galin. *Software Quality Assurance: From Theory to Implementation.* Addison-Wesley, 2003.

[22] Vahid Garousi, Lionel C. Briand, and Yvan Labiche. Analysis and visualization of behavioral dependencies among distributed objects based on uml models. Technical report, Software Quality Engineering Laboratory (SQUALL), Carleton University, 2006.

[23] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 2002.

[24] Barbara G.Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, 2001.

[25] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with Datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[26] Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Tools: A graph-based approach to modeling and detecting composition conflicts related to introductions. In *Proceedings of the 6th international conference on*

Aspect-oriented software development AOSD, 2007.

[27] Yoon Kyu Jang, Heung Seok Chae, Yong Rae Kwon, and Doo Hwan Bae. Change impact analysis for a class hierarchy. In *Proceedings of the Fifth Asia Pacific Software Engineering Conference (APSEC)*, 1998.

[28] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[29] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[30] Jörg Kienzle, Yang Yu, and Jie Xiong. On composition and reuse of aspects. In *Proceedings of the AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, 2003.

[31] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of the 3rd AOSD Workshop on Linking Aspect Technology and Evolution (LATE)*, 2007.

[32] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. Class firewall, test order and regression testing of object-oriented programs. *Journal of Object-Oriented Programming (JOOP)*, 1995.

[33] Ramnivas Laddad. I want my AOP! (part 2). Java World, `http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2.html`

[34] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management:A study of the management of computer application software in 487 data processing organizations*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[35] Miguel P. Monteiro and João M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, 2005.

[36] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[37] J.-Hendrik Pfeiffer and John R. Gurd. Visualisation-based tool support for the development of aspect-oriented programs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD)*, 2006.

[38] J.-Hendrik Pfeiffer, Andonis Sardos, and John R. Gurd. Complex code querying and navigation for AspectJ. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, 2005.

[39] Václav T. Rajlic and Keith H. Bennett. A staged model for the software life cycle. *IEEE Computer*, 33(7):66 – 71, July 2000.

[40] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G.Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2004.

[41] Tamar Richner, Stéphane Ducasse, and Roel Wuyts. Understanding object-oriented programs with declarative event analysis. In *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, 1998.

[42] Spencer Rugaber. Program comprehension for reverse engineering. In *Proceedings of the AAAI Workshop on AI and Automated Program Understanding*, 1992.

[43] Robert W. Sebesta. *Concepts of programming languages (7th Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[44] Hideaki Shinomi and Tetsuo Tamai. Impact analysis of weaving in aspect-oriented programming. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, 2005.

[45] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[46] Maximilian Störzer and Christian Koppen. CDiff: Attacking the fragile pointcut problem. In *Proceedings of the Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[47] Maximilian Störzer, Robin Sterr, and Florian Forster. Detecting precedence-related advice interference. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2006.

[48] Francis Tessier, Mourad Badri, and Linda Badri. A model-based detection of conflicts between crosscutting concerns: Towards a formal approach. In *In International Workshop on Aspect-Oriented Software Development*, 2004.

[49] The AspectJ Team. The AspectJ language guide.

[50] The AspectJ Team. The AspectJ programming guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`

[51] Kris De Volder. JQuery: A generic code browser with a declarative configuration language. In *Proceedings of the Eighth International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2006.

[52] Scitools website. `http://www.scitools.com/products/understand/cpp/product.php`

[53] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of the 26th International Conference on Technologies of Object-Oriented Languages and Systems (TOOLS USA)*, 1998.

[54] Sai Zhang and Jianjun Zhao. Change impact analysis for aspect-oriented programs. Technical report, Center for Software Engineering, Shanghai Jiao Tong University, 2007.

[55] Jianjun Zhao. Change impact analysis for aspect-oriented software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, 2002.

[56] Jianjun Zhao. Measuring coupling in aspect-oriented systems. *Proceedings of the 10th International Software Metrics Symposium (Metrics) session on Late Breaking Papers*, 2004.