Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
2008

AN IMPLEMENTATION OF LOGIC POINTCUT LANGUAGES IN METASPIN

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Roberto Sanchez Sanchez

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Advisors: Kris Gybels (Vrije Universiteit Brussel)
Bram Adams (Universiteit Gent)

Abstract

Aspect-oriented programming is a research area that focuses on providing constructs to deal with separation of concerns and modularization. One particular area of interest in aspect-oriented programming, is the use of logic programming as a basis for pointcut languages. It would be interesting to experiment with logic pointcut languages and help the developer in choosing the optimal pointcut in a particular context, like speed measures, memory performance.

This dissertation presents an implementation of a logic pointcut language in the Metaspin environment. The goal of Metaspin is to provide a framework to experiment with programming languages for aspect-oriented programming. Using Metaspin, we experiment with a logic pointcut language and with a temporal logic pointcut language. Metaspin allows to evaluate an aspect language in different contexts, like memory efficiency, evaluation time, etc. We believe the experience of using Metaspin to evaluate and implement a logic pointcut language, helps to better understand how aspect languages work on the inside.

Acknowledgements

First of all, I would like to thank my advisors, Kris Gybels and Bram Adams, for always guiding me, for proofreading my work, and for everything they taught me.

I also want to thank Prof. Dr. Theo D'Hondt for promoting this dissertation, and Dr. Jacques Noye and all the people that make this EMOOSE program a reality.

Many thanks to all my old and new friends, specially to my friends from the EMOOSE program. I appreciate your friendship and all the support during this year.

Finally, I would like to thank my parents, my brother and my sister, for always giving me they support and giving me the opportunity to study abroad.

Table of Contents

List of Figures

Chapter 1

Introduction

In this dissertation, we discuss how to implement a logic pointcut language in the Metaspin environment, which is an environment for experimenting with aspect languages. Modularization and separation of concerns to manage complexity are important goals in computer science [4]. Aspect-oriented programming is a research area that focuses on providing constructs to deal with these problems. Before aspect-oriented programming, there have been several programming techniques for separation of concerns: procedures, modules, and objects. However, it has come to the attention of researchers that not all concerns can be easily separated and easily modularized with those programming techniques. This resulted in aspect-oriented programming, whose goal is to achieve a more advanced level of separation of concerns and to provide better modularization of crosscutting concerns.

A particular interesting area of research in aspect-oriented programming are logic pointcut languages. Logic pointcut languages intend to facilitate expressing pointcuts by using logic expressions (in the form of queries). Some research and different approaches on logic pointcut languages have been done (i.e. CARMA, HALO, ALPHA, etc.).

An interesting framework for experimenting with aspect languages is Metaspin. The framework provided by Metaspin allows to combine aspect languages and to design new ones. Metaspin was developed by the European Network of Excellence on Aspect-Oriented Software Development [25]. The use of the framework is interesting because we can create an implementation of a logic pointcut language that helps to better understand how aspect languages work.

Therefore, in this dissertation we show how to use the Metaspin framework to build a new pointcut language. First of all, we make a CARMA-like [12] implementation on top of Metaspin. We couple the SOUL [29] interpreter to Metaspin, and design a library of logic predicates that allow to write logic pointcuts. However, the achievement of this goal involves linguistic symbiosis, because of the combination of an aspect language for one language (Smalltalk [11], which is object-oriented) with pointcuts written in another language (SOUL, which is a logic language).

The next step is to add more features that will increase the functionality of the logic pointcut language. Therefore, we use HALO logic pointcut language, which is based on temporal logic programming. The combination of HALO with Metaspin adds the possi-

bility to reason about the execution history of join points. HALO uses the Rete network algorithm to evaluate its pointcuts. The Rete implementation evaluates a program executed in Metaspin as a network of facts and obtains results by applying some rules from the existing implementation of HALO.

One important goal of this dissertation, is to have a workbench to experiment with logic pointcuts and temporal logic pointcuts. This allows for example to measure memory usage, evaluation time per join points, and some other experiments for performance. This measurements can be used to detect problems in an implementation, to see if it is possible to make optimizations, etc. Therefore, this dissertation presents the implementation of a logic pointcut language in the Metaspin framework, that helps to understand how aspect languages work. A second important goal is to show the experience of using Metaspin in combination with other technologies to evaluate aspect languages. The use of a framework with a complete structure of an aspect language, gives the possibility to reuse and adapt that framework to learn in detail how aspects languages work in the inside.

1.1 Document structure

The remainder of the thesis is structured as follows. In the next chapter, we briefly explain aspect-oriented software development and logic pointcut languages. We make a review of current research work on logic pointcut languages by presenting the different existing approaches, in order to better contextualize our work.

In chapter three, we present Metaspin, which is the framework we use to build a new logic pointcut language. Metaspin allows to experiment with aspect-oriented languages. We describe in detail how this framework works and the advantages of using it to build a logic pointcut language.

Chapter four explains logic meta programming and the technologies used on top of it for this dissertation. The general concepts involved in logic meta programming are explained. We introduce SOUL, which is a declarative framework in Smalltalk for logic meta programming. Then, we present an overview of temporal logic programming. At the end of the chapter we explain the Rete algorithm for production systems, as a pattern matching solution for the temporal logic pointcut HALO.

The fifth chapter presents an implementation of a new logic pointcut language in Metaspin, which is our approach. The logic pointcut language allows to write logic pointcuts and temporal logic pointcuts by using Metaspin framework in combination with SOUL. This new logic pointcut language uses the different tools presented in chapter three and four to express and evaluate logic pointcuts. This was possible by means of linguistic symbiosis to couple the different tools.

Chapter six presents the experiments to evaluate our approach. It describes a small application built in Smalltalk: a shop application and presents the evaluation criteria for our logic pointcut language. The evaluation criteria measures the memory and timing performance of the implementation of the Rete network.

Finally, in chapter seven, we discuss our conclusions for this dissertation. We also

mention some future work related to this dissertation.

Chapter 2

State of the art

Aspect-oriented software development is a topic that has been studied a lot over the past few years. One of the areas of research in aspect-oriented languages is the use of logic programming to write aspects. Logic programming in combination with aspect-oriented languages are called logic pointcut languages. Researchers have been studying these logic pointcut languages in the past few years, and here we give a review of what has been done so far, as the objective of this dissertation is to build a new temporal logic pointcut language.

In this chapter we first introduce the concept of aspect-oriented software development and the problems that this technology intends to solve. After that, we briefly introduce the concept of logic meta programming and logic pointcut languages, as such concepts will be described in detail in chapter 4. Then, we give a description of some of the existing logic pointcuts languages. Finally, we present a summary of the most important points of this chapter.

2.1 Aspect-Oriented Software Development

The main goals of aspect oriented languages are to deal with crosscutting concerns, scattering, tangling and separation of concerns [15]. In many systems, some concerns crosscut parts of a program which lead to scattering and tangling. The goal of aspect-oriented programming is to modularize crosscutting concerns.

Separation of concerns means to break down a program into small parts to facilitate its understanding [4]. Scattering means that a source element is related to multiple target elements, which result hard to understand or maintain because a concern of that source element is spread over different modules. This means that every time the code has to be modified, it will affect all the elements and will give as a result multiple concerns inside a source code. Tangling is when two or more concerns are implemented in the same body of code, making it more difficult to understand. Therefore, the goals of aspects are to allow good modularity, less tangled code, more reusability, and to provide easy maintenance and evolution. We can define that aspects are used to well-modularized

crosscutting concerns [16].

Aspect-oriented languages are specified by pointcuts, advices and weaving. A join point is a well-defined point in the execution of a program. A pointcut selects a set of join points that exposes some of the values in its execution context. An advice is code that executes at each join point in a pointcut, in other words, it defines what to do at a certain pointcut. And finally, weaving means to apply the advice to all join points defined in the pointcut declaration.

2.2 Existing logic pointcut languages

There are many approaches for logic pointcut languages, but the most important and the first one that used logic meta programming within aspects was CARMA [12]. We make a review of the existing logic pointcut languages here, focusing on the most important characteristics of each of them.

2.2.1 Logic meta programming

Logic meta programming (LMP) uses a logic programming language at meta level to reason about source code. It combines a declarative language at meta level with an object-oriented base language. LMP is a suitable approach and very useful to support aspect-oriented programming. Since base-level programs are expressed as logic terms, facts and rules at the meta level, and meta level programs can manipulate and reason about the structure of the base-level programs.

Therefore, in this dissertation, we use LMP as the underlying technology to build an implementation of a new logic pointcut language in Mestapin.

2.2.2 Logic pointcut languages

A logic pointcut language is an aspect-oriented language that uses logic programming to write pointcuts in a declarative way. Known as logic pointcuts, they are written as logic queries (i.e. like Prolog queries). The logic pointcuts execute over logic facts to obtain information about join points, and those join points are matched in case the pointcut finds a solution based on those known facts.

The benefits of using logic programming within aspect languages are that this programming paradigm is based on a problem domain. It works with a set of rules over known facts or over new facts that can be derived from those known facts. Then, the focus of logic programming is not on *how* a problem needs to be solved, but on *what* the problem is. Using declarative specifications, logic programming is much closer to mathematical intuition.

2.2.3 CARMA

CARMA (formerly Andrew) [12, 26] is an aspect-oriented programming language based on AspectJ, which uses logic meta programming for crosscutting specifications. Carma was the first logic pointcut language and hence it was very important because from that idea researchers began to develop more logic pointcut languages.

Join points are used like in AspectJ, and an advice will match and execute some code before or after the execution of a join point. However, aspects are defined using logic programming to have the possibility to use unification and logic rules to write a predicate. Predicates in CARMA are expressed as logic queries and have Prolog-like syntax because of the usage of the logic language SOUL [28]. Queries are used to express conditions over join points.

The main characteristics of CARMA are: pattern-based crosscuts and the open weaver. The pattern-based crosscuts try to minimize the existing coupling between aspects and classes. The coupling affects the reusability of an aspect and the evolvability of a program. For this reason, CARMA pattern-based crosscuts can include some state changing rules to minimize the coupling. The open weaver gives the possibility to define new predicates and build libraries. It is possible to create specialized join points for specific programs or for the execution of idioms. In a program everything can change, and having the possibility to define specific join points gives the flexibility to match join points by means of the pattern-based crosscuts.

The next example is the observer pattern [9] in CARMA that uses pattern-based crosscuts (taken from [26]).

```
after ?jp matching
  reception(?jp, ?msg, ?args),
  inObject(?jp, ?obj),
  objectClass(?obj, ?class),
  changeState(?class, ?msg),
  not(caller(?jp, ?obj))
do
  observer notify
```

The example shows CARMA's syntax, which is Prolog-like (for the use of SOUL). Logic variables are represented with a question mark (?) at the beginning of its name.

2.2.4 ALPHA

ALPHA [21] is an aspect-oriented extension of an object-oriented core language based on L2[1] calculus. ALPHA supports classes and single inheritance, as well as standard static type system. In this language, aspects become effective after they are deployed.

ALPHA defines pointcuts with Prolog-style to increase the expressiveness of the language. This feature allows to add new pointcuts in a declarative way, turning the language

[1]Lambda 2 calculus is a formal, minimal, imperative, class based, object-oriented language with inheritance, and without overloading

into an open pointcut language. It also supports the idea of pointcuts in a time line (to refer about past join points). This language gives the possibility to create libraries of pointcuts and includes abstraction capabilities (like functional composition and higher order pointcuts).

A library of pointcuts can import other libraries using the module mechanism of Prolog. Currently, ALPHA provides a standard pointcut library, and libraries defined by the user. A programmer needs to understand and be familiar with Prolog semantics, as well as having a good understanding of pointcuts, to be able to compose a predicate in ALPHA.

Now, for a better understanding of this language, the next example shows how pointcuts work in ALPHA.

```
1  class DisplayUpdate extends Object {
2    Display d;
3    after now(ID), set(ID, _, P, _, _), instanceof(P, Point)
4      { this.d.draw(P); }
5  }
6  class Main extends Object {
7    Display d; DisplayUpdate du;
8    void main() {
9      this.d = new Display(); this.du = new DisplayUpdate();
10     this.du.d = this.d;
11     deploy(this.du) { this.doSomething() }
12   }
13   void doSomething() { ... }
14     ...
15 }
```

The example uses a class called *DisplayUpdate*, which updates the display of a figure. In this case, the pointcut will match all the assignments to fields of the objects of type Point. The *now(ID)* method gets the time stamp and binds it to the variable ID (line 3). Then, the advice *after* (lines 3 - 4) only takes effect after one instance of *DisplayUpdate* is deployed (line 11). Finally, that advice will become effective during the execution of the method *doSomething()* (line 13).

2.2.5 GAMMA

Gamma [17] is an aspect-oriented language based on ALPHA, so it is an object-oriented core language based on L2 calculus. It was created in the University of Darmstadt, in Germany. Gamma has Prolog-like syntax, which means a pointcut is declared as a Prolog query.

This language provides support for pointcuts that can refer to future event executions (it differs from ALPHA, which refers to past events) and keep trace of them. This language is expression-oriented, which means a predicate can be created as a Prolog query by the combination of different attributes. However, Gamma has a restriction that indicates that a special variable called *Now* has to be included in at least one of the predicates. The special variable *Now* contains a value that indicates where the advice has to be executed.

The base of Gamma relies on the usage of logic programming, obtaining as a result a flexible and powerful pointcut language. Gamma gives the pointcuts the possibility to keep trace of a program execution which refers to a future event, but this characteristic has some limitations. When Gamma is used in a more specific approach, it only reasons about execution trace and temporal relations between join points on a very abstract level. This means, that it is necessary to improve the efficiency of the language and avoid some existing restrictions. The next example shows a class with aspects in Gamma.

```
1 class main extends Object {
2   bool var;
3   before set(Now,_,Address,_,_) {
4     print(Address)
5   }
6   bool main(bool x) {
7     this.var := true
8   }
9 }
```

The execution trace is represented as a collection of facts in the Prolog database, which represent the steps of the interpreter. This implementation of weaver gives the possibility to have an idea of where the advice invocation is needed. However, to achieve this goal, the program must be run at least once, so the trace can be done. An iterative process (that stops once a fixed-point is reached) has to be performed in order to match all the points where the advice has to be inserted. If more points are matched and gives different results than the first iteration, then the first iteration result is the one that will be executed. The iteration occurs inside the weaver invocation and can remain active until it matches the fixed-point or if the base program is terminated (without advice) as a result of a break condition or other operation that forces the program to stop. The next example shows a display solution using the *before* advice in Gamma.

```
1 before calls(T1,main,_,operation,_),
2   cflow(T1,T2),
3   calls(T2,point,_, setpos),
4   endCall(Now,T1,_) {
5      this.display.update(true)
6   }
```

The example shows a display that has to be updated only if at least one of its elements has been changed. The pointcut crosscuts the end of the execution of *main.operation*. If there is a *point.setpos* call in its control flow, and if it matches the call on *setpos* (line 3), then the display is updated at the end of the method call. This allows us to refer to the execution history of the operation, making the aspect short and also easy to understand.

2.2.6 HALO

HALO [13, 14] stands for History-based Aspects using LOgic, and it is a logic pointcut language based on temporal logic programming for Common Lisp[2]. HALO, unlike the majority of logic pointcut languages, uses Lisp syntax instead of Prolog-like syntax.

HALO allows to reason about past join points (history-based aspects), and predicates of higher-order temporal operators (i.e. since, allpast, etc.). Pointcuts are expressed as logic queries, and advices can be written as logic rules consisting of two parts: a head specifying a Lisp function to be invoked when a pointcut match occurs, and a body specifying a pointcut.

Higher-order predicate temporal operators represent temporal relations between the inside and the outside operator conditions. There are different temporal operators available in HALO: *all-past*, *most-recent* and *since*. HALO uses this operators to reason about past join points and the context in which they occurred.

The *all-past* predicate matches the inner pointcut against all past join points of a join point matched by the outer pointcut. Given a join point that is matched by the outer pointcut, the *most-recent* predicate tries to find the most recent join point which matches the inner pointcut. The *since* predicate, unlike the other predicates, has two inner pointcuts. The first inner pointcut is evaluated against the past join points related to the join points matched by the outer pointcut to select a join point in the past. The second inner pointcut is evaluated against the join points between the two other join points. Finally, there is also a predicate *escape* which enables symbiosis between logic variables that should be previously bound to a value to allow the use of a Lisp program with logic conditions [14].

HALO language gives the programmer the flexibility to define rules for predicates using the *defrule* construct. This feature allows to extend the language when needed by defining new predicates. Another important characteristic found within HALO is the weaver. HALO weaver includes a runtime that intercepts the join points and a query engine in charge of matching the pointcuts to a join point. First, the weaving is done to match the advice code, and then the query engine stores the execution history that allows to reason about past join points.

The next is an example of a simple pointcut that shows the syntax and the use of a temporal predicate in HALO.

```
((gf-call 'checkout ?argsC)
 (most-recent(gf-call 'buy ?argsB)))
```

This example shows an inner pointcut *(gf-call 'buy ?argsB)* evaluated against the most recent join point in the past of the join points matching the outer pointcut *(gf-call 'checkout ?argsC)*. The temporal predicate *most-recent* matches the inner pointcut against the outer pointcut to obtain the most recent join point matched. As the inner pointcut is always evaluated against the outer pointcut, variables can be shared between them.

[2]Commonly known as CL, is a multiparadigm, general-purpose programming language which is a dialect of the Lisp programming language.

2.2.7 LogicAJ

LogicAJ [6] stands for Logic Aspects in Java, and is an extension of the well known AspectJ language [20]. LogicAJ uses logic meta programming to express pointcuts, increasing the expressiveness of the language by using unification and the integration of logic variables into the language (just like the rest of the logic pointcut languages). These characteristics enable the use of meta variables for base program elements. LogicAJ provides two important features for writing aspects: uniform genericity and interference analysis.

Uniform genericity allows the implementation of aspect effects that can vary depending on the values produced for meta variables during the evaluation of pointcuts.

Interference analysis gives an automatic order in which the aspects should be deployed to avoid inconsistency (this is accomplished by determining an interference-free execution). It also makes aspect semantics independent of the textual ordering of advice declarations within an aspect.

LogicAJ provides support for predicate-based meta variable binding, which minimizes restrictions on the use of meta variables. The binding mechanism replaces the manual enumeration of static values by weave-time evaluation of the predicates. The predicates only select the join points that will be applied and bind the meta variables to values from the static context of those join points. List meta variables is another feature provided by this language.

LogicAJ notation for logic meta variables is represented with a question mark (?) at the beginning of its name (i.e. *?class*), and the notation for the list meta variables is represented by a double question mark (i.e. *??args*). This language can be seen as a generic aspect language that allows reuse by its genericity mechanisms.

Properties like constructing higher-level pointcuts, uniform genericity, possibility to express dependencies between multiple join points and base-language code patterns with meta variables, make LogicAJ a good aspect language implementation in logic meta programming according to technical report "Independent Evolution of Design Patterns and Application Logic with Generic Aspects - A Case Study" [23].

To illustrate the power of LogicAJ binding mechanism we show the implementation of a class named *ClassPosing* [10]. This class is a useful technique enabled by the Objective-C[3] runtime. This allows to create all future instances of class A as class B, where class B is a subclass of A. The *ClassPosing* example written in LogicAJ looks as follows.

```
1 abstract aspect ClassPosing {
2
3   // Who poses from whom? May return multiple pair of values:
4   abstract pointcut replace(?class,/*by*/?sub);
5
6   Object around(?sub, ??args):
```

[3]A reflective, object-oriented programming language that adds Smalltalk-style messaging to C. Used primarily on Mac OS X and GNUstep.

```
7       // Determine replaced class and replacing class:
8       replace(?class,/*by*/?sub) &&
9       // Check if ?sub is a subclass of ?super:
10      subtype(?sub,?class) &&
11      // Intercept ?super constructor invocations
12      call(?class.new(..)) &&
13      // Bind ?args to the argument list of the invocation:
14      args(??args)
15      {   // Return instance of posing subclass ?sub (includes
16          // waves time check that the constructor exists):
17          return new ?sub(??args);
18      }
19  }
```

This example is very self explanatory from the comments present inside the code. We have the pointcut definition (line 4), and the advice definition (lines 6 - 18), that executes the aspect for the class *ClassPosing*.

2.2.8 LogicAJ2

LogicAJ2 [6] is a pointcut language designed with the concept of fine-grained generic aspects. The design of this language is based on three built-in fine-grained pointcuts, which makes it different from other aspect-oriented languages. This is the result of code pattern and logic meta variables (LVMs) usage within this language. The join points are classified in three types:

- Statements
- Expressions
- Declarations

LogicAJ2 provides these three types of join points, together with three basic pointcuts (stmt, expr, and decl). The combination of these three pointcuts gives the possibility to create more complex pointcut semantics. Besides this feature, LogicAJ2 provides code patterns that are used to describe join point shadows. Those join points are written with Java-style syntax. Furthermore, the code patterns enable the use of logic meta variables that are needed to substitute all possible program elements, such as types, identifiers, statements, expressions, etc.

The syntax of LogicAJ2 for meta variables is the same as LogicAJ, indicated with a question mark (?) at the beginning of the name (i.e. ?methodBody). The syntax for logic list meta variables is denoted by two question marks (??) at the beginning of the name (i.e. ??parameterList). If there are unnamed logic meta variables, they are represented by an underscore (?_ and ??_).

Finally, another characteristic of LogicAJ2 is the use of an explicit join point variable. This variable is a logic meta variable representing the join point described by the code

pattern, and it is the first argument in a basic pointcut. This variable can be used together with other code patterns to give an easier and organized usage of pointcuts.

The next example shows the definition of a pointcut in LogicAJ2 syntax.

```
pointcut binary_boolean_expressions(?jp, ?lhs, ?rhs):

     expr (?jp, ?lhs == ?rhs) || expr(?jp, ?lhs >= ?rhs)
  || expr (?jp, ?lhs <= ?rhs) || expr(?jp, ?lhs != ?rhs)
  || expr (?jp, ?lhs += ?rhs) || expr(?jp, ?lhs -= ?rhs)
  || expr (?jp, ?lhs <  ?rhs) || expr(?jp, ?lhs >  ?rhs)
  || expr (?jp, ?lhs |  ?rhs) || expr(?jp, ?lhs || ?rhs)
  || expr (?jp, ?lhs &  ?rhs) || expr(?jp, ?lhs && ?rhs);
```

The example above, allow us to write a explicit join point variable that uses the *binary_boolean_expressions* pointcut, and it is shown in the next example.

```
pointcut if_with_bool_condition(?jp, ?condition, ??stmts):

    binary_boolean expressions(?condition, ?_, ?_) &&
    stmt(?jp, if(?condition) {??stmts});
```

2.2.9 Semantics of static pointcuts in AspectJ using Datalog

The use of a Datalog query language to write logic pointcuts is another approach to improve the functionality and the expressive power of a language. This approach focuses on the semantics of static pointcuts in AspectJ using Datalog [1].

Datalog is similar to Prolog, with the difference that Datalog does not allow to construct new data type values like lists. The richness of Datalog, compared with Prolog, is that gives the programmer the possibility to express pointcuts in terms of semantics, rather than syntactic criteria. Hence the semantics of Datalog programs are straightforward and are used to express the semantics of AspectJ pointcuts.

A subset of Datalog called *Safe Datalog* provide conditions that guarantee that every program can be evaluated to a set of relations, and it was used in this approach to develop the idea of expressing aspects as semantics terms. The objective is to write AspectJ pointcuts as Datalog queries. To reach this objective, a set of rules has to be rewritten for each static pointcut in AspectJ to be translated into Datalog predicates. In conjunction with the set of rules, a set of primitive pointcuts has to be supplied to query the structure of the program to evaluate AspectJ pointcuts. This has to be done in order to be able to express pointcuts in Datalog.

The benefits of using Datalog to express the pointcuts consists on the facility to write them as direct expressions (like queries), and the possibility that the rewrite rules can be directly executed to perform some tests. The aim of these rewrite rules using Datalog is to provide a semantic-type of writing for the AspectJ pointcut language, and this semantic-type consists of 90 rules. The details of the rules can be found in the survey "Semantics of Static Pointcuts in AspectJ" [1]. It is also possible the use of wildcards to denote sets of types.

The semantics produced with Datalog queries are executable and can be used directly in an AspectJ implementation. This approach showed that writing pointcuts directly in Datalog in a semantic way is possible, and it also provides the flexibility to write pointcuts in both styles, AspectJ patterns and Datalog semantics.

The idea of the rewrite rules is shown in the next example. It is the rewrite rule for Boolean combinations of pointcuts.

```
[aj2dl(pc1 && pc2, C, S)] -> [(aj2dl(pc1, C, S), aj2dl(pc2, C, S))]
[aj2dl(pc1 || pc2, C, S)] -> [(aj2dl(pc1, C, S)); (aj2dl(pc2, C, S))]
[aj2dl(!pc, C, S)] -> [not(aj2dl(pc, C, S))]
```

In this example, the *aj2dl* constructor is used to rewrite an AspectJ pointcut to Datalog. This Datalog expression means that the pointcut *pc* has the variables C and S. Variable C refers to the class where the pointcut is located and variable S is a condition, if the expression turns out to be true, then S is a shadow in the context of *pc*.

2.3 Summary

Aspect-oriented software development attempts to help programmers in the separation of concerns, specifically crosscutting concerns. In this chapter, we reviewed the concept of logic pointcut languages, which shows how logic meta programming has increased the expressiveness of aspect languages. This increase of expressiveness is due to the fact that writing aspect expressions as logic queries (either Prolog-style, Lisp-style or Datalog-style) give the user a better understanding, as well as making it easier to write expressions because of the use of logic queries, which is a more natural way of writing pointcuts.

The description of the different approaches of logic pointcut languages allow us to have a view on the research that has been done in that area. It is important to remind that CARMA was the first approach and the most important because it was the motivation to write more logic pointcut languages.

CARMA's strong points are the pattern-based crosscuts and the weaver. ALPHA and HALO have the special characteristic of being able to reason about past join points, which means both of them keep trace of the execution history of the pointcuts. Gamma, in contrast with HALO and ALPHA, refers to future join points. LogicAJ is a more generic language compared to the others, and it has two characteristics that barely exist in the other languages, which are uniform genericity and interference analysis. Besides LogicAJ, there is another language which includes fine-grained generic aspect, that lacked in LogicAJ, and that language is known as LogicAJ2 (an extension of LogicAJ). Finally, the use of Datalog to write AspectJ pointcuts as semantics, is an approach which has the possibility to write pointcuts directly on Datalog as semantics and also as AspectJ patterns.

This overview on the state of the art of the thesis shows the characteristics of logic pointcut languages. It also shows the use of logic meta programming in aspect-oriented programming to write aspects, and how this combination increases the expressiveness of a

language. These concepts are used to develop an implementation of a new logic pointcut language, which is the purpose of this dissertation.

In the next chapter, we introduce a framework called *Metaspin* [2], which serves as the base framework to develop an implementation of a new logic pointcut language.

Chapter 3

Metaspin

Metaspin is an environment that provides a workbench to experiment with programming languages for aspect-oriented programming. The workbench provided by Metaspin allows to combine aspect languages and to design new ones. Metaspin was developed by the European Network of Excellence on Aspect-Oriented Software Development [25].

In this chapter we describe Metaspin [2] and its characteristics. First, we give a brief introduction on bytecodes, as Metaspin uses the Smalltalk bytecode interpreter as the base interpreter. Second, we present a detailed description of Metaspin and we explain how it works. After that, we show an example of how Metaspin executes a program. Finally, at the end of the chapter we summarize the most important elements of Metaspin.

3.1 Smalltalk bytecode

Smalltalk code is stored as bytecode instructions, which are sequences of eight-bit instructions. The Smalltalk interpreter executes the bytecode instructions step-by-step. A stack is used during the execution of the bytecode instructions to handle the data that results from executing the bytecode instructions. The bytecodes give better performance than interpreting the source code directly. Even when translating the code into bytecodes introduces a delay before a program is run, the speed of the execution is improved compared to interpretation.

The bytecode instructions of a method are held in a Smalltalk instance of *CompiledMethod*. This method contains the bytecode translation of the Smalltalk code.

The interpreter understands 256 bytecode instructions, but as some of the instructions are larger than one byte, some of these instructions have to take extensions. Those extensions are only part of the instructions and could be one or two bytes added after the bytecode. The bytecode instructions can be classified in five categories:

Push bytecodes add the source of an object to the top of the stack.

Store bytecodes assignments of variables are stored on top of the stack.

Send bytecodes send the selector of a message and the number of arguments needed. The receiver and the arguments are present in the stack.

Return bytecodes return the value of the message that invoked the *CompiledMethod* once the method has been completely executed.

Jump bytecodes indicate which non-consecutive bytecode instruction needs to be executed next.

A complete description of these categories and all of the 256 bytecode instructions can be found in the original Smalltalk-80 book [11]. Another important element to take into account is the state of the interpreter while executing a *CompiledMethod*. The state contains the bytecodes that are being executed, the bytecode instruction pointer, a stack, the temporary variables of the *CompiledMethod*, and the receiver and arguments of that method.

The state of the interpreter is saved in an object called *context*. Several contexts will be present in the execution of a program. Those contexts contain the state of the interpreter, but only one context is active at a time, and is the one which has the current state of the interpreter. The other contexts remain suspended until they are set to active again.

The suspended contexts are linked to the context of the CompiledMethod that interrupted them. This means, that an active context remembers the context that it suspended, so after the active context finishes its execution, the suspended context can be resumed.

The contexts are important for the bytecodes because they keep track of the *CompiledMethods* and the blocks that are executed. All contexts have a stack to store the objects it processes. The classes used to represent contexts inside Smalltalk are the *BlockContext* class and the *MethodContext* class. The interpreter is in charge of manipulating the stack of the contexts during the bytecodes execution. Figure 3.1 shows a diagram of contexts with the possible classes used to represent them.

Here, we present an example of a *CompiledMethod* containing the bytecode translation of the method *test1* and the *CompiledMethod* of the method *test2*, from a class called *Example*. The purpose of this example is to show in more detail how the bytecodes work. To achieve this goal, we will show how the bytecode interpreter executes the bytecode instructions. The methods *test1* and *test2* are:

```
test1
   | x y z |
   x:=1.
   z:=y.
   var:=x.
   ^ self test2
```

```
test2
   | x |
   x:=2.
   ^ 2
```

The bytecode instructions of the method *test1* (on the left) and the method *test2* (on the right) are:

```
17 <76> pushConstant: 1
18 <68> popIntoTemp: 0
```

```
13 <77> pushConstant: 2
14 <68> popIntoTemp: 0
```

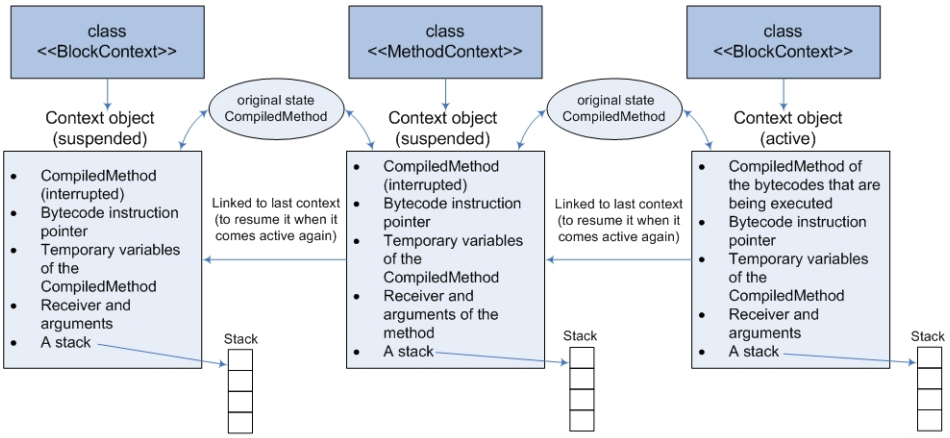

Figure 3.1: Context objects

```
19 <12> pushTemp: 2
20 <69> popIntoTemp: 1
21 <10> pushTemp: 0
22 <60> popIntoRcvr: 0
23 <70> self
24 <D0> send: test2
25 <7C> returnTop
```

```
15 <10> pushTemp: 0
16 <7C> returnTop
```

In this example, each line is a bytecode instruction that contains three elements: the offset of the instruction which is the two digit number before the brackets, the raw bytes of the instruction which are between the brackets in hexadecimal form, and the description of the bytecode instruction. The bytecode interpreter executes each line at a time.

The interpreter uses a *step* method to execute a single bytecode instruction at a time. This method returns the context that would be the active context after the bytecode that is being executed. This means that in a *blockContext*, we can send the message *step* that will increment the pc (program counter) by one allowing to execute the bytecode instructions of the block step-by-step.

First, we will explain the bytecode instructions from *test1*. The offset 17 is a bytecode instruction which pushes a number (one) on the stack. Then, the offset 18 is a bytecode instruction which pops a value off the stack (in this case the number one) and stores it in a temporary location. The temporary location refers to an object zero, which is the reference value of the variable x in our code. This object number zero is stored in a temporary location in the context's stack, and that location is pointing to the temporary variable needed by the *CompiledMethod*.

The next bytecode instruction on the offset 19 pushes a temporary location on the stack. In this case, the temporary location is marked with the number two, which is the reference value of the variable y. Then, the offset 20 pops the variable y off the stack and stores it in a temporary location. The temporary location refers to an object one, which

is the reference value of the variable *z*. As we can see, the context refers three temporary variables (*x, y, z*), and each of them is represented by a number in the context ($0- > x$, $1- > z$, and $y- > 2$). These variables can be referenced through their position in the temporary location's of the context.

The offset 21 pushes a temporary location on the stack (variable x). We already explained that the object number zero refers to this variable. The offset 22 pops a value off the stack (variable x) and stores it in a receiver variable. This receiver variable is the instance variable *var*, which was stored on an object when the class definition was saved.

The offset 23 pushes the receiver on the stack. The receiver is self. The next bytecode instruction on the offset 24 sends the message *test2* as an argument to self. The result of sending this message is pushed on the stack.

The method *test2* is a method in the class *Example* which returns the number two. As this method generates a new *CompiledMethod* to be executed (shown in the code above), when the step message is sent at this bytecode, the active context becomes suspended and a new context is created and set active. In this new active context, the bytecode instructions for the method *test2* are executed. When this method has reached the *returnTop* instruction, the value of the method is returned and the step message will return the value from the suspended context to be resumed and to be set to active again.

Finally, the offset 25 is the bytecode instruction that returns the top of the stack as the value of the message.

The example described how the bytecode interpreter executes bytecode instructions. The explanation of each of the bytecode instructions gave a better idea on how Smalltalk executes a program and this is needed to understand how Metaspin works.

3.2 Metaspin

Metaspin stands for *Metalevel Aspect Interpreter* and is implemented as a small interpreter on top of the bytecode interpretation framework of Squeak Smalltalk. Metaspin works by stepping through the bytecode instructions of a program.

To be able to do this stepping, Metaspin uses a method called *advancedStep* from class *MetaspinContinuation*, which sends a *step* message to the context object. This context object is held in an instance variable called *baseContext*. This is how Metaspin is able to execute a single bytecode instruction at a time.

It is also important to note that when an *advanceStep* is made, there is an instance variable called *currentJoinPoint* which is set to the value *nil*. In other words, it resets the *currentJoinPoint* instance variable to keep it updated at every instruction.

The Metaspin interpreter is composed of elements that combined enable expressing and interpreting aspects. The most important elements are: join points, pointcuts, advices and continuations. These elements are explained in more detail in the next sections of this chapter. Figure 3.2 shows a diagram of these elements and how Metaspin is composed in general.

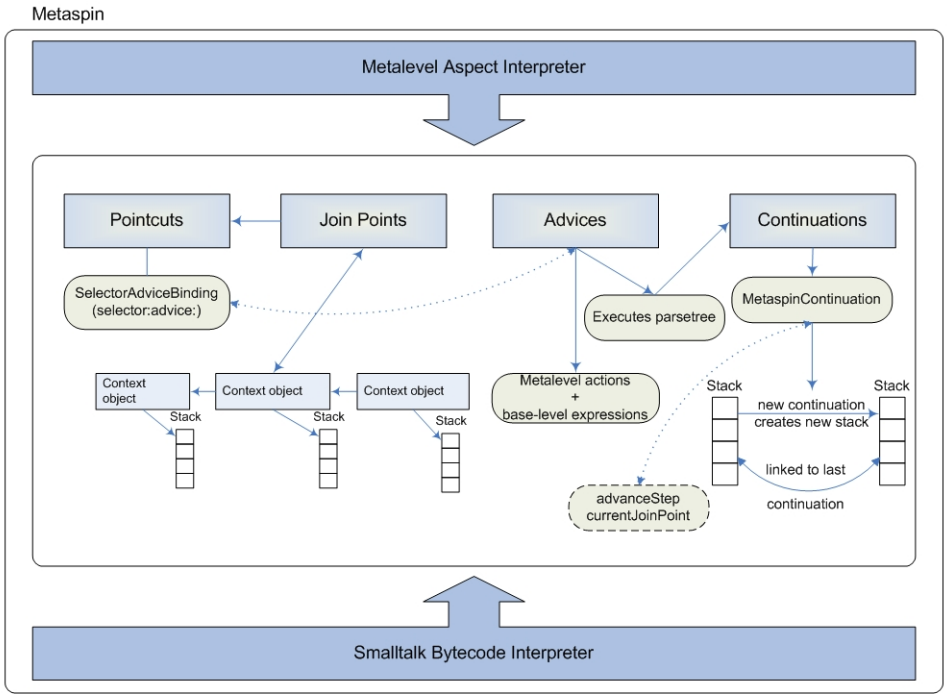

Figure 3.2: Metaspin diagram

3.2.1 Join points

The join points are created during the execution of a program in Metaspin. There are different types of specialized join points in Metaspin:

message send joinpoint: messages sent from an object in a program's execution.

reference joinpoint: makes reference to the object's value whenever it is a variable.

block return joinpoint: returns the value from a block.

assignment joinpoint: assignments of an object in a program's execution.

return joinpoint: returns the value of an object.

This is a general description of join points in aspect-oriented programming, but in Metaspin these join points are modeled as objects that show an AOP view on the state of the context executed in the Smalltalk bytecode interpreter. The different join points have methods that make it easy to access the information that is relevant for this AOP view. These methods get all the necessary information from the contexts, allowing easy use and manipulation of this information.

The join points in Metaspin have a structural part and a behavioral part. The structural part is a location in the source code, while the behavioral part is a continuation (this will be explained in the subsection *continuations*).

In Metaspin the join points implement a combination of the structural part and the behavioral part. The join points are created during the execution of a program. Metaspin uses the Smalltalk bytecode interpreter to execute a program. As the bytecode is executed, the join points can be created at each bytecode instruction. Figure 3.3 shows how a join point is created at a particular bytecode instruction, and the relation that exists between contexts and join points.

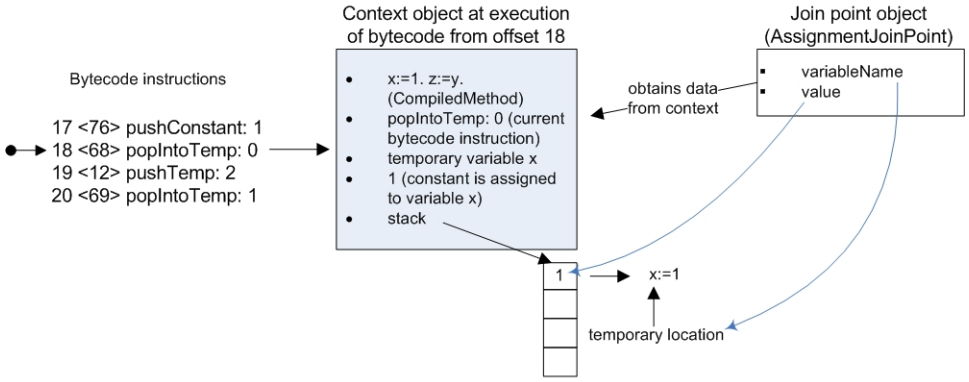

Figure 3.3: Relation between contexts and join points

As we can see from Figure 3.3, a join point is created by taking the necessary information from the context. In the bytecode instruction from offset 18 an assignment was made (*x:=1*), so the join point created was an *AssignmentJoinPoint* which takes two values present in the context: the name of the variable and the value assigned to that variable. Any of the join points in Metaspin is created in the same way, and the information needed to create those join points is obtained from the context.

Figure 3.4 shows a class diagram representing how the join points are structured in Metaspin. The diagram shows the main methods of each specialized join point as well as the classes that inherit from them. For a better understanding of the different join points, we will extend the description given above for each join point.

A *MessageSendJoinPoint* is created when a message is sent, the corresponding bytecodes are from the *send literal selector* (between 131 and 134), the *send arithmetic message* (between 176 and 191) and the *send special message* (between 192 and 207) bytecode instructions. This join point takes the message and the arguments for that message, and it also has a history record that keeps track of the messages sent.

The *ReturnJoinPoint* is created to hold the value that is returned from an object. The bytecode instructions that correspond to this join point are the *return from message* bytecode (which can return receiver, true, false, or nil) and the *return stack top from message*. Both bytecode instructions are held between the range 120 and 124. The return value is taken from the base context, and is always at the top of the stack of that context. This join point is also created when the send of a message has a return value, or when a method returns a value.

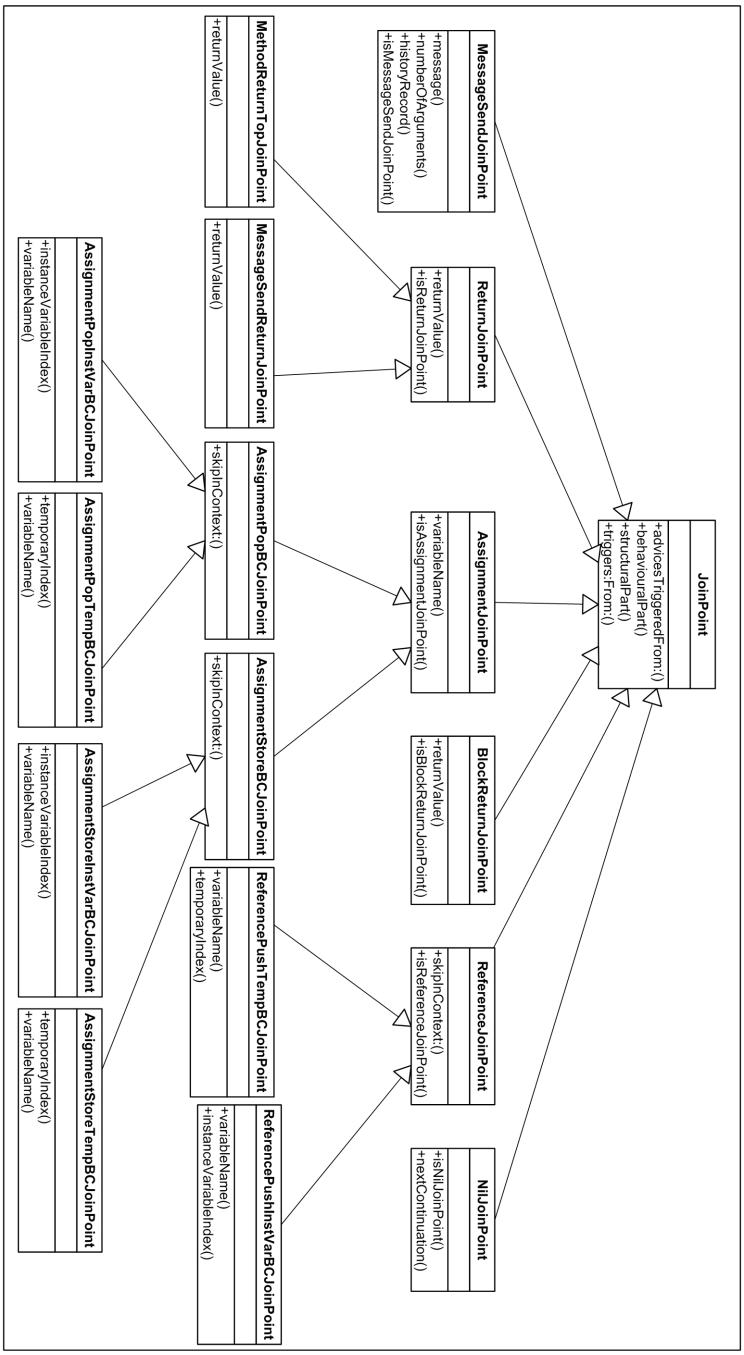

Figure 3.4: Join Point Class Diagram

An *AssignmentJoinPoint* is created everytime an assignment is made in a program through the bytecode instructions *pop and store receiver variable* and *pop and store temporary location*. This join point takes from the context the name of the variable that is being assigned and also the value of that variable. When an instance variable or a temporary variable is assigned, this join point is also created. However, global variables do not create an *AssignmentJoinPoint*. The class diagram from Figure 3.4 shows the different possible assignments for this join point.

The type of *AssignmentJoinPoint* that is created depends on the kind of assignment is made. For instance variables it will depend on the bytecode instruction executed, which could be the case for a pop of an instance variable (*AssignmentPopInstVarBCJoinPoint*)or for a store of an instance variable (*AssignmentStoreInstVarBCJoinPoint*). The same happens with temporary variables.

A *BlockReturnJoinPoint* only returns the value from a block with the bytecode instruction *return top of stack from a block*, which is the bytecode instruction 125. This join point is similar to the *ReturnJoinPoint* because it returns a value, but the only difference is that the *BlockReturnJoinPoint* returns the value of a block. This join point is created when a block has finished its execution.

Finally, the *ReferenceJoinPoint* is a join point that makes a reference to an object's value. The bytecode instruction for this join point is the *push temporary location*, and corresponds to the bytecode instructions in the range 16 to 31. Typically this join point is created when there is a reference to a variable. Because the compiler does not save the variable itself in the context, but its reference to that variable. This join point is created using the name of the variable that is referenced in the base context.

A *NilJoinPoint* is a special join point used only as a return value that implements the necessary behavior to go on with the execution when no other join point is created. The join points are associated to contexts because the latter contain the information on the interpreter state including the information of the join point executed at that moment.

3.2.2 Pointcuts

Pointcuts in Metaspin are normal Smalltalk expressions that are evaluated by a pointcut evaluator which takes a join point as argument.

Metaspin uses a method called *selector:advice:* from the class *SelectorAdviceBinding* to write pointcuts. The pointcuts can be written as simple Smalltalk block expressions. The advices are also written as Smalltalk block expressions. The following code is an example of a pointcut in Metaspin:

```
SelectorAdviceBinding
    selector: [ :jp | jp inAdviceExecution not & jp isAssignmentJoinPoint ]
    advice: (Advice block: [ :jp | 42 ]
```

The example shows a pointcut definition that matches assignment join points. The next section explains how the advices work.

3.2.3 Advices

Advices express the functionality that needs to be invoked at a matched join point. The advices in Metaspin are Smalltalk expressions, but those expressions are represented in a parsetree (advice program). A parsetree is a tree structure representing an advice program. This tree structure can be a combination of primitive and composed metalevel actions and base-level expressions. This means that Metaspin evaluates these metalevel actions by processing the parsetree.

The metalevel actions define how an advice is going to be executed and how specific instructions that can only occur in advices need to execute. Base-level expressions are normal Smalltalk expressions and statements, like message send expressions.

When a join point is created, an advice is executed by executing the program of the parsetree. At the moment, the advice code is contained in Smalltalk blocks, which encapsulate the advices. The next subsection will explain in detail how an advice is executed by creating a continuation and how an advice is triggered at a join point.

3.2.4 Continuations

Metaspin uses the Smalltalk bytecode interpreter as the base interpreter, but at every instruction the bytecode interpreter is halted to check pointcuts. If a pointcut wants to intercept the execution of the bytecode, Metaspin puts the base context object at the top of a stack and it is stored as a continuation. A new continuation is created by creating a new context object with the current state of the interpreter.

The base context represents the state of the interpreter at that moment and it remains on that stack until it becomes active again after executing the new continuation. Then, the advice code is first compiled by Smalltalk and afterwards executed by the bytecode interpreter.

A metaspin continuation encapsulates an execution process and executes this process step-by-step. After each step of the process, a join point can be created and a new continuation can be created at this join point that takes over the execution. The encapsulated process consists of a base context, an advised continuation, a stack of that context and a pointer to the stack.

First of all, each continuation is saved in a base context stack when the bytecode is being executed. Then, the state of the interpreter at that moment is saved for the creation of each continuation.

Each continuation also contains an advised continuation object that links to the continuation it has interrupted. This is how Metaspin constructs a stack for every context. In the case of the first continuation of the program, this link refers to a sentinel continuation which acts as the end of the continuation stack, to deal with the end condition of the stack of continuations. The initial base context object as well as the current join point of the execution are kept in instance variables of the Metaspin class.

3.3 Metaspin bytecode interpreter

In this section, we will show how Metaspin executes a program by using the Smalltalk interpreter as the base interpreter. The code for this example is:

```
aContinuation:=MetaspinContinuation atContext: [ | x y z | x:=1. z:=y.
                                                 Example new test2 ].
```

This code assigns a block expression to the method *atContext:* from the class *MetaspinContinuation*. The class *MetaspinContinuation* is in charge of creating and handling the continuations. An advice is created with the code:

```
MetaSpin add:
  (SelectorAdviceBinding
    selector: [ :jp | jp inAdviceExecution not & jp isAssignmentJoinPoint ]
    advice: (Advice block: [ :jp | 1 ])).
```

This advice will be created after matching an *Assignment join point* and will show us what Metaspin does when an advice is introduced. Now, the bytecode translation of the code containing the class *MetaspinContinuation* is the next one:

```
 33 <41> pushLit: MetaspinContinuation
 34 <89> pushThisContext:
 35 <75> pushConstant: 0
 36 <C8> send: blockCopy:
 37 <A4 08> jumpTo: 47
 39 <76> pushConstant: 1          (Nil join point)
 40 <68> popIntoTemp: 0           (Assignment join point)
 41 <11> pushTemp: 1              (Reference join point)
 42 <6A> popIntoTemp: 2           (Assignment join point)
 43 <43> pushLit: Example         (Nil join point)
 44 <CC> send: new                (Message send join point)
 45 <D2> send: test2              (Message send join point)
 46 <7D> blockReturn              (Block return join point)
 47 <E0> send: atContext:
 48 <81 C4> storeIntoLit: aContinuation
 50 <7C> returnTop
```

In this example, at every instruction the bytecode interpreter is halted. If no pointcut intercepts the execution of that instruction, the interpreter continues to the next instruction and so on. A continuation is created every time a pointcut matches a join point. The example gives a better idea of how Metaspin uses the bytecode interpreter.

The important bytecode instructions for this example are between the offset 39 and 46. The execution of the block expression of our example takes place between those bytecode instructions. This block contains almost the same code as the one from the method *test1* (the assignment of *var:=x* is not used here), which was explained in the first example of this section. For this reason, in this example, we will focus mainly on describing the

creation of continuations, join points and how the advices are triggered when a join point is matched.

Let us assume that Metaspin starts evaluating the offset 39, which is a bytecode instruction that pushes the number one on the stack. To create a join point instance at this bytecode instruction, we need to execute the method *currentJoinPoint* on the *MetaspinContinuation* instance.

The Figure 3.5 presents two inspector windows at the execution of this bytecode instruction. The inspector window on the *MetaspinContinuation* shows the current joint point at this bytecode instruction, which is a *Nil join point* that holds the value for the number one that is pushed on the stack. And the inspector window on the *BlockContext* shows the pc (program counter) in the offset 39, which is the bytecode that is being executed.

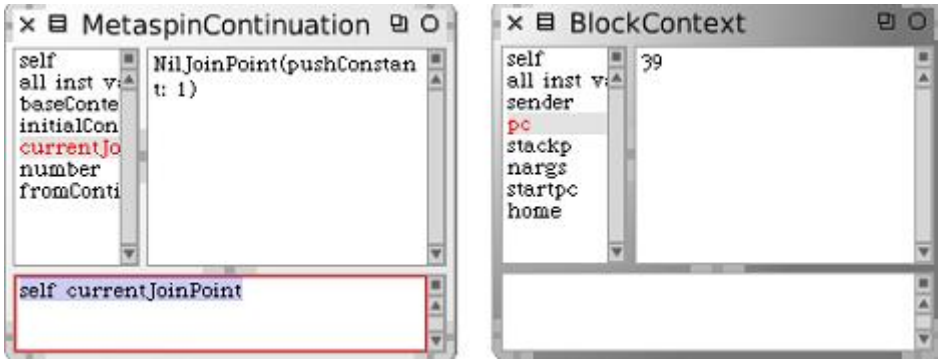

Figure 3.5: Nil join point and block context at offset 39

The next bytecode instruction is the offset 40 which pops the constant one off the stack and stores it in the variable *x*. When the method *currentJoinPoint* is executed at this bytecode instruction, an *Assignment join point* is created.

Figure 3.6 shows this join point in the inspector window on the *MetaspinContinuation*, while in the inspector window on the *BlockContext* appears a new instance variable *1*, just below the home instance, which represents the context of the stack (the constant number one). Then, the *Assignment join point* takes the variable (*x*) from the temporary location, and the value that is at the top of the stack on the base context. This binds the number one to the variable *x*.

At this *Assignment join point* we can show what happens when an *advice* is introduced to be executed. An advice is created after matching the join point with the code showed at the beginning of this section.

This advice will try to intercept the *Assignment join point*. At this moment, the base context is stored as a continuation, and a new continuation containing the execution of the advice is created. Figure 3.7 shows the two instances of *MetaspinContinuation*. The first instance is the interrupted continuation from the code that created the *Assignment join point*, while the second instance is the new continuation created to execute the advice.

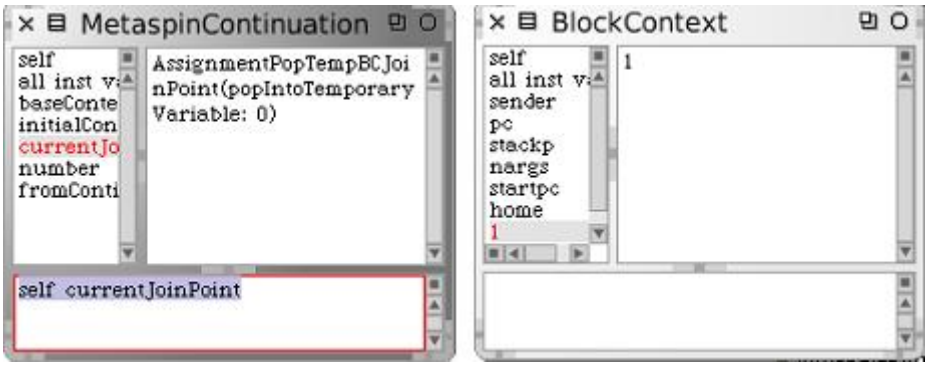

Figure 3.6: Assignment join point and stack value at offset 40

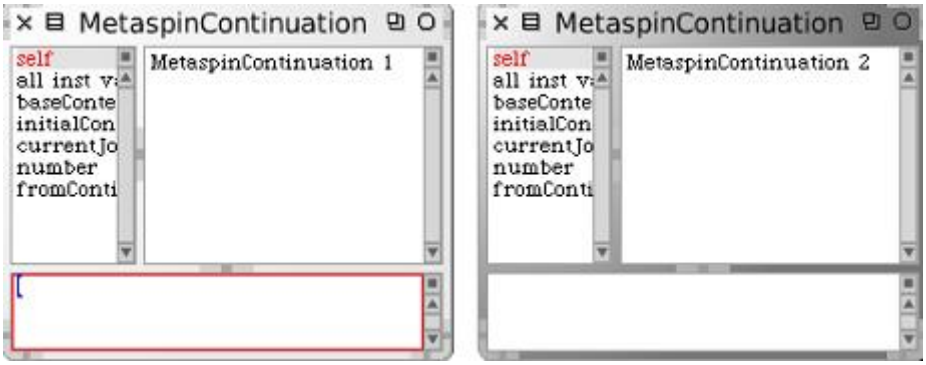

Figure 3.7: Metaspin continuations

The bytecode instructions of the advice are executed by Metaspin to check if there is any matching advice to be triggered. Then, the pointcut is executed. Once the advice has been executed at the join point, the continuation of that advice is over. Then, the continuation that was halted to execute this advice is restarted. If we had two or more advices which matched at the same join point, new continuations would be created for each of those advices. Only after all the advices have been executed, the first continuation will be restarted.

The interpreter now executes the bytecode instruction from the offset 41, which pushes a temporary location (the reference value for the variable y) on the stack. A *Reference join point* is created to save the reference value of the variable. The Figure 3.8 shows in the inspector window on the *MetaspinContinuation* that the join point is holding the reference value pushed onto the stack.

The offset 42 pops the variable y off the stack and stores it in the variable z. This bytecode instruction creates an *Assignment join point* in the same way as the one from the offset 40. The only difference is that this join point binds one variable (y) to the variable z.

The offset 43 pushes a literal variable (*Example*) on the stack, creating a *Nil join point*. The next bytecode instruction on the offset 44 sends the message *new* to the literal variable

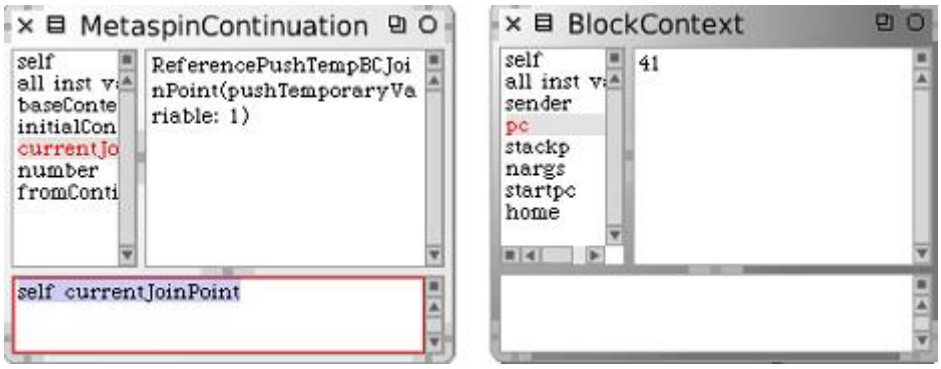

Figure 3.8: Reference join point at offset 41

on top of the stack. After that, the offset 45 sends the message *test2* as an argument to the top of the stack, which now has the value *Example* with the selector *new*. This bytecode instruction creates a *Message send join point*, showed together with the content of the stack in Figure 3.9. For the sake of simplicity, we will not explain the join points that are created from the bytecodes of *new* and *test2*.

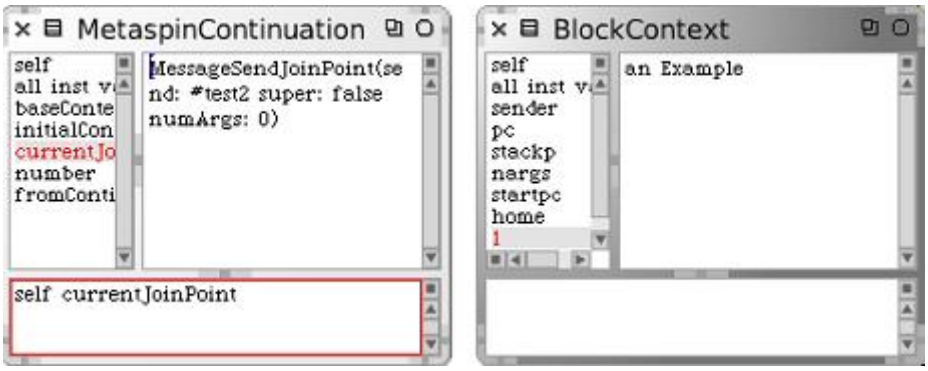

Figure 3.9: Message send join point and stack value at offset 45

Finally, the offset 46 is the bytecode instruction that returns the top of the stack as the value of the block. A *Block return join point* is created after executing this last bytecode instruction. This join point keeps the return value of the block and that value is pushed on another stack, which is the Smalltalk call-stack.

The example above showed that once a pointcut wants to intercept the execution of the bytecode, Metaspin keeps the base context object at the top of a stack and it is stored as a continuation. The advice code is then first compiled by Smalltalk and afterwards executed by the bytecode interpreter. Metaspin creates join points while executing the bytecode instructions. At each bytecode instruction a join point is created. However, a nil join point is used as a return value that implements the necessary behavior to go on every time there is no join point to be created at the current execution bytecode.

3.4 Summary

This chapter gave a detailed description of Metaspin. It explained the different elements that are included in the Metaspin interpreter.

We described the different join points that can be created in Metaspin. We also explained how the advices are executed, which are normal Smalltalk expressions from the combination of parsetrees and metalevel actions. Then, we mention the basic concept behind metaspin: continuations.

An example of how a continuation is made in Metaspin was presented. For a better understanding of how this process is done, we gave a brief explanation on bytecodes. Metaspin works directly with bytecode instructions, so we saw the execution of a *CompiledMethod* and how the interpreter executed the bytecode instructions. It is important to outline that Metaspin works with the Smalltalk bytecode interpreter as the base interpreter, which is the core of the workbench.

The next chapter presents a full description on logic meta programming. It describes the tools that will be used together with Metaspin to achieve the goals of this thesis.

Chapter 4

Logic and Temporal Pointcut Languages

In this chapter we discuss the concepts behind *logic programming* and *logic meta programming* (LMP). The idea behind explaining these concepts is to understand how a logic pointcut language can be built. We also discuss the different technologies that can be used to build such kind of language. It is important to give this explanation, as the goal of this thesis is to build a logic pointcut language by combining these technologies.

First, we give an overview of logic programming. After that, we give a detailed overview of LMP and its principles. Then, we describe SOUL [27, 28] and its functionality, followed by a description of temporal logic meta programming. Finally, we describe the Rete algorithm as a pattern-matching mechanism.

4.1 Logic programming

Logic programming [18] is a programming paradigm based on mathematical logic, which makes it different from other programming paradigms. In logic programming a program is conceived as a logical theory for which the execution of the program searches a proof.

Logic programming is an instance of declarative programming because it focuses on *what* the problem is and not on *how* to solve the problem, like imperative programming does. Another difference between these two paradigms is on the concept of a variable. A variable in imperative programming is a name for a memory location that can store data, while in logic programming it is a variable in the mathematical sense (i.e. a placeholder that can take on any value, just like in a mathematical formula). By means of declarative specifications, logic programming is closer to mathematical intuition than imperative programming.

The basic constructs of logic programming are terms and statements (data structures). There are three basic statements: facts, rules and queries. A term can be a constant, a variable or a compound term. Constants are numbers (integers or floats) and atoms (a general purpose symbol without special meaning). Variables are placeholders that can take

any value. A compound term is composed of a functor (which is an atom) and a sequence of one or more arguments.

Logic programming is based on a problem domain and works with a set of rules over known facts or over new facts that can be derived from those known facts. Facts hold static information that express unconditional truths in the application domain, while rules specify conditional truths by deriving new facts from existing ones. The facts and rules are held as a collection in a database and can be accessed via queries.

In logic programming, a query is computed by an algorithm that is called *resolution*. The *resolution* typically used in logic programming is based on a special variable binding mechanism called *unification*. Unification and backtracking are one of the main ideas behind logic programming. The way a query matches a pattern in a program is called unification. The way it searches for patterns is called backtracking. Unification represents the mechanism of binding variables. In some languages, unification is done with the equality symbol $=$, but it is also used together with other arithmetic operators like: $+$, $-$, $*$, $/$, etc. In other words, unification means that a term can be unified to another term. However, logic programming only finds the first solution of a query and uses unification to bind that solution to a variable. Then, backtracking is the repeated searching for additional solutions of a query, so unification is used on the rest of the solutions to bind the values to a variable.

In a logic program, the predicates define the relations between objects (terms and statements), so that the connection between input data and the information expected as output is well defined. This means that a program in a Logic Programming language expresses a relation between data [24]. A logic program is a collection of horn clauses. A horn clause is of the form:

$$H \leftarrow B_1, ..., B_n$$

A horn clause with at most one positive literal is called a definite clause, while a horn clause with no positive literals is sometimes called a goal clause. Where H is the head of the rule and $B_1, ..., B_n$ is the body of the rule. Then, H is said to be proven if all of B_1 and B_n are proven to be true. There is another case, when H has no body, and then H is unconditionally true. We could say then, that a horn clause with $n = 0$ is a fact, and horn clauses with $n > 0$ are rules.

Prolog [22] is the most popular logic programming language. Prolog stands for Programmation en Logique (French for Programming in Logic). It is commonly associated with Artificial Intelligence. Prolog was created in the seventies at the University of Marseilles and the first compiler was developed at the University of Edinburgh. Examples of facts written in Prolog are:

```
cat(Felix).
cat(Tom).
cat(Garfield).
mouse(Mickey).
mouse(Jerry).
mouse(Speedy)
```

These two facts say that Felix is a cat and Mickey is a mouse. The equivalent rules for this facts would be:

```
cat(Felix) :- true.
cat(Tom) :- true.
cat(Garfield) :- true.
mouse(Mickey) :- true.
mouse(Jerry) :- true.
mouse(Speedy) :- true.
```

Having defined the facts and the rules, we can now apply queries to obtain information from that data. Examples of queries are:

```
?- cat(Felix).   [is Felix a cat?]
yes

?- mouse(X).     [what things are mice?]
X=Mickey
X=Jerry
X=Speedy

?- cat(Mickey).  [is Mickey a cat?]
no
```

Those are just a few examples of the kind of queries we can make to ask for information. The queries search through the facts and rules to work out the answer. First, when the query asks what things are mice, unification is used to bind the first solution: *Mickey* to the variable X. Then, backtracking searches for additional solutions of the query, and it finds a second solution: *Jerry*. The second solution uses unification to bind *Jerry* to the variable X. This process is repeated to find the third solution of the query.

A more detailed description of logic programming can be found in the book "Simply Logical" [5].

4.2 Logic meta programming

Logic meta programming is an instance of declarative meta programming, and a meta program is a program that manipulates other programs as data. For a better understanding, we explain the general terminology of the concepts related to meta programming. These concepts are based on *Computational Reflection* [19].

A *program* is a specification of a computational system that manipulates representations of entities from some problem domain. A program is expressed in a formalism that can be interpreted automatically in order to obtain the computational system it specifies. This formalism is called a *programming language*. Then, programs are built to reason about

almost anything imaginable. It defines representations of the entities or concepts one wants the program to reason about in terms of data structures that are built into the programming language. For this reason, programs can be constructed to reason about other programs as well. Examples of such programs are interpreters, compilers, code generators, etc.

A program that manipulates other programs is called a *meta program* (also called a meta-level program), while the programs that do not manipulate other programs are called *base programs* (also called base-level programs) [3].

Declarative meta programming is defined as the use of a declarative programming language for writing meta programs in terms of a given base language. In logic meta programming, typically the declarative language uses a logic programming language at the meta level, and a programming language (i.e. object-oriented language) at the base level. Meta level programs manipulate and reason about the base level program, while the base level in logic metaprogramming is represented by facts, rules and terms at the meta level.

4.3 SOUL

SOUL stands for *Smalltalk Open Unification Language* and is designed for logic meta programming. SOUL is a logic programming language implemented and integrated into Visual Works Smalltalk and Squeak Smalltalk. It is a symbiosis between a logic language (Prolog) and an object-oriented language (Smalltalk). As SOUL is based on Prolog, it allows to write facts and rules as Prolog queries, but includes extra features for meta programming. LiCoR is the library that provides the logic predicates for processing programs in Smalltalk. LiCoR is used specifically to reason about Smalltalk code, and contains an extensive library of SOUL predicates.

SOUL is used to reason about the structure of object-oriented programs and it provides four layers that contain sets of rules to reason about the base language structure:

logic layer contains the predicates that add core logic-programming functionality (list handling, program control, arithmetic, repository handling, etc).

representational layer reifies some of the concepts from the base language (classes, methods, instance variables and inheritance).

basic layer adds auxiliary predicates to make it easy to reason about implementation. Since the representational layer only provides the most primitive information, this layer is absolutely necessary to interact on a reasonable level of abstraction with the logic meta programming language.

design layer groups all predicates that express particular design notations (programming conventions, design patterns and UML class diagrams).

To get a detailed description on these layers, we refer to the Phd dissertation from Roel Wuyts [29].

SOUL, as any other logic language, has a database where it stores the information that is gathered. All this information is stored in the SOUL system repository, which is the database.

SOUL has Prolog-like syntax, but with some differences. For example, the main difference is that SOUL allows to call Smalltalk code inside a rule or a query. Another difference is the notation for the variables. In Prolog, variables are capitalized symbols, while in SOUL variables start with a question mark (?). A second notable difference is the explicit use of the statement *if* in SOUL, while in Prolog this statement is represented with the symbol $:-$. Finally, another difference is found in the use of the Boolean *and*, which in SOUL is represented with a comma.

We will show now an example program in SOUL. The example shows the *class* predicate which has two rules that are the logical representation of a class:

```
Rule
   class(?C) if
       atom(?C),
       [Smalltalk includes: ?C].

Rule
   class(?C) if
       var(?C),
       generate(?C,[Smalltalk allClasses]).
```

The first rule describes what happens when a query is launched with as head the *class* predicate and the logic variable $?C$ is bound to a value, and then the Smalltalk term checks if $?C$'s value is a class that is included or not in Smalltalk.

The second rule describes what happens when the logic variable is unbound (only applied if $?C$ is just a variable), and then the *generate* predicate binds each of the values generated with the Smalltalk term (*Smalltalk allClasses*) to the variable $?C$.

With these two rules defined, we can now make some queries in SOUL to ask for information. For example we could ask SOUL to list all its classes with the next query:

```
Query class(?X)
```

The query will return as a result all the solutions for $?X$, which will be all the classes of the system. We can make another query to ask if Dictionary is a class of the system:

```
Query class([Dictionary])
```

In this case, the query will return the boolean value true as Dictionary is indeed a class of the system.

SOUL is a very useful language used for various applications of declarative meta programming, and in this thesis we will show how it can be combined with other tools to implement a logic pointcut language that uses temporal logic predicates on top of Metaspin.

In the next section we introduce the concept of temporal logic meta programming, which is a form of LMP that uses temporal logic, and that is used to reason about the execution history of a program.

4.4 Temporal logic meta programming

Temporal logic meta programming is a form of logic meta programming where the logic programming language is based on a temporal logic programming language. Temporal logic [8] is used to describe any system of rules and reasoning about it in terms of time, i.e. as a sequence of states.

Temporal logic uses logical operators and modal operators. The logical operators are truth-functional (i.e. *and*, implication, etc.) and the modal operators are temporal operators (i.e. *until*, *release*, *sometime*, etc.), which can be used in rules to specify at what time a rule applies.

Temporal logic meta programming can handle the temporal aspects of the execution of a program. It can be used on logic pointcut languages to add the functionality to reason about the join points executed in a program. This can be used to describe an execution history of join points as a sequence of events and reason about it.

HALO [13] (explained in chapter 2) is a pointcut language based on temporal logic programming for Common Lisp[1]. HALO refers to past join points by using temporal logic, which gives this language the possibility to reason about a program execution and its state. Its implementation is heavily based on an enhanced Rete network.

The next section explains the Rete algorithm used to evaluate logic predicates in HALO.

4.5 Rete algorithm

The Rete algorithm [7] is an efficient pattern matching algorithm used for implementing production systems (which are a kind of expert systems). Rete means "network" in Latin and the algorithm builds a network of nodes. It was designed by Dr. Charles L. Forgy at the Carnegie Mellon University in 1974. This algorithm was created to provide an efficient implementation of an expert system. An expert system is a software system with a set of rules that attempts to reproduce the performance of one or more humans, usually in a specific problem domain.

Before the Rete algorithm, a naive implementation of an expert system checked each rule against known facts in a knowledge base, firing that rule if necessary, then moving to the next rule and looping back to the first rule when finished. The performance of this naive implementation was very slow.

A production system is a program that consists of a set of rules ("productions"), and those rules provide some form of artificial intelligence. Productions have two parts: a

[1]Commonly known as CL, is a dialect of the Lisp programming language. CL is a multiparadigm, general-purpose programming language.

precondition (*If* statement) and an action (*Then*). The *If* part of a production is called left-hand side (LHS), while its *Then* part is called right-hand side (RHS). A production system has a database called working memory, which keeps the knowledge (the data operated by the productions). An example of a production looks like:

$$IF\ swims(X)\ AND\ has\_gills(X)\ AND\ lay\_eggs(X)\ THEN\ (add\ fish\ X)$$

The LHS of a production is a sequence of patterns, while the RHS consists of an unconditional sequence of actions. When a precondition matches the knowledge, the production is *triggered*. If an action is executed, it means that the rule *fired*.

If we compile the production given above, it gives as a result the Rete network shown in Figure 4.1. The rule compiled for this network says that *if something swims, has gills and lay eggs, then that something is a fish*. Figure 4.1 shows the root node at the top, which is represented with an ellipsis shape. Then, we have the input nodes represented with diamonds shapes (which can also be called pattern nodes or filter nodes). Those pattern nodes match facts, and the facts that pass the tests are sent to the output nodes. Then, we have the trapezoid shape which represents join nodes. These join nodes are the result of the matching patterns that came from the upper left of the network. Finally, the oval shape at the bottom node of the network is the terminal node or production node. The working memory uses memory tables containing the data (the facts) of the network. Those memory tables are associated with the nodes, and are represented at each node as a box with the label "memory".

Every node in the network represents one or more tests on the LHS of a rule, and each of the nodes has one or two inputs and can have any number of outputs. At the same time, every node processes the facts that have to be added or removed from the working memory. Input nodes are the top of the network, while output nodes are at the bottom. There are multiple paths from the root node to a leaf, but the path from the root node to a specific leaf node defines a complete LHS of a rule.

A set of facts filters through the network from the top to the bottom. When those facts pass the tests on a LHS of a rule, the rule will be *fired*, which means the action of the rule is executed and may assert new facts.

An important feature of Rete implementation in HALO, is the memory table garbage collection. Each of the entries of a memory table have a life time assigned and contains the facts stored for each node. The life time of a memory table entry consists of the time when an entry was created and when an entry can be removed. In fact, an entry can be removed once it cannot be used to derive more conclusions in a node. The decision of removing a memory table entry depends on the input data of the node which has the memory table.

4.6 Summary

This chapter gave a complete description of logic programming and logic meta programming. It also explained in detail the technologies used to build a logic pointcut language

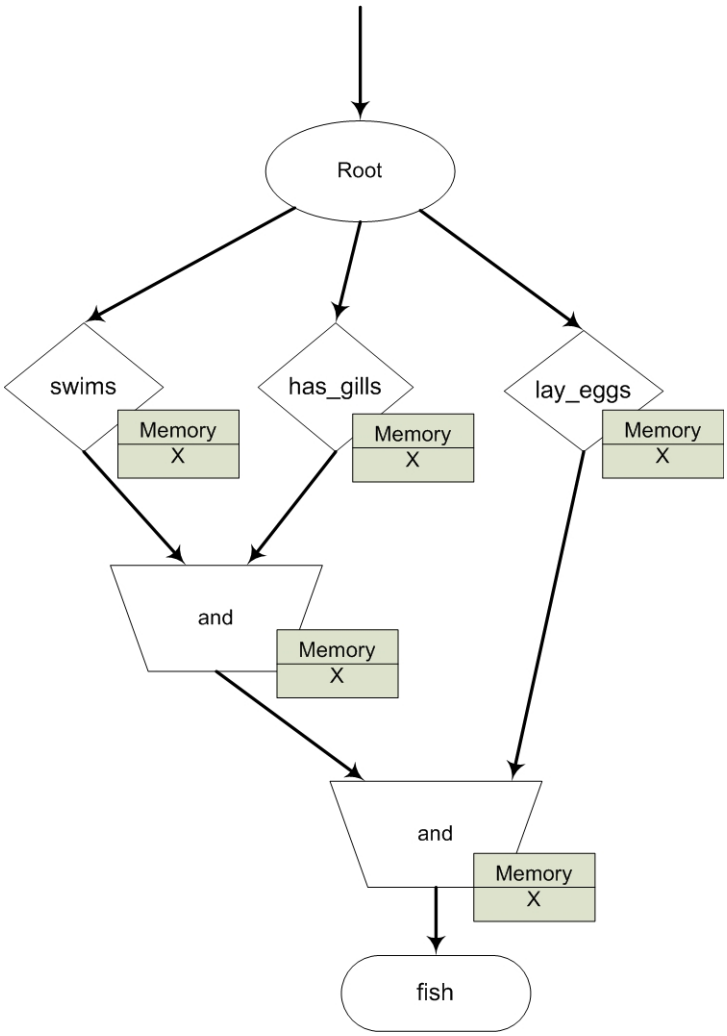

Figure 4.1: Example of a Rete network

and a temporal logic pointcut language. Defining these terms helped to understand better what can be accomplished by using logic programming as the programming paradigm, instead of another programming paradigm like imperative programming.

SOUL provides the declarative framework to work with logic pointcut languages. The examples provided showed some of the features of this language. We explained the Rete algorithm which allows to build networks for pattern matching. Rete is used in HALO to evaluate temporal predicates.

In the next chapter we explain how we combine the different technologies given in this chapter to build a new logic pointcut language.

Chapter 5

A logic pointcut language

A logic pointcut language is a pointcut language that makes use of logic meta programming. The goal of a logic pointcut language is to obtain more expressiveness of a language and to demonstrate that the use of logic meta programming allows to express more features in a language than AspectJ-based pointcut languages.

In this chapter, we give a description of the logic pointcut language we implemented and how we combined different tools like Metaspin, SOUL and Rete to achieve this goal.

Finally, we explain how we use the Rete algorithm in our implementation of the logic pointcut language.

5.1 Introduction

This chapter presents the design and implementation of our logic pointcut language. The components we used to develop the logic pointcut language were: Metaspin as the base framework, SOUL and the Rete algorithm implemented in SOUL. We discuss the design of the logic predicates and the rules that had to be defined in order to implement this language.

The implementation allows to write logic pointcuts and temporal logic pointcuts, as well as to make facts at every join point and add them into a Rete network to be evaluated. In the previous chapter, we described each of the components used to build our logic pointcut language separately, but in this section we explain how we combine them, as well as some of the problems encountered during the coupling of these components.

5.2 High-level design of the logic pointcut language

In this section we present Figure 5.1 which gives a schematic overview of how the technologies have been combined to build the logic pointcut language.

In the bottom of the figure we have Metaspin, which serves as the base framework. Then, we have the declarative framework of SOUL, which has to be combined with Metaspin to be able to write logic pointcuts. These two technologies were coupled by

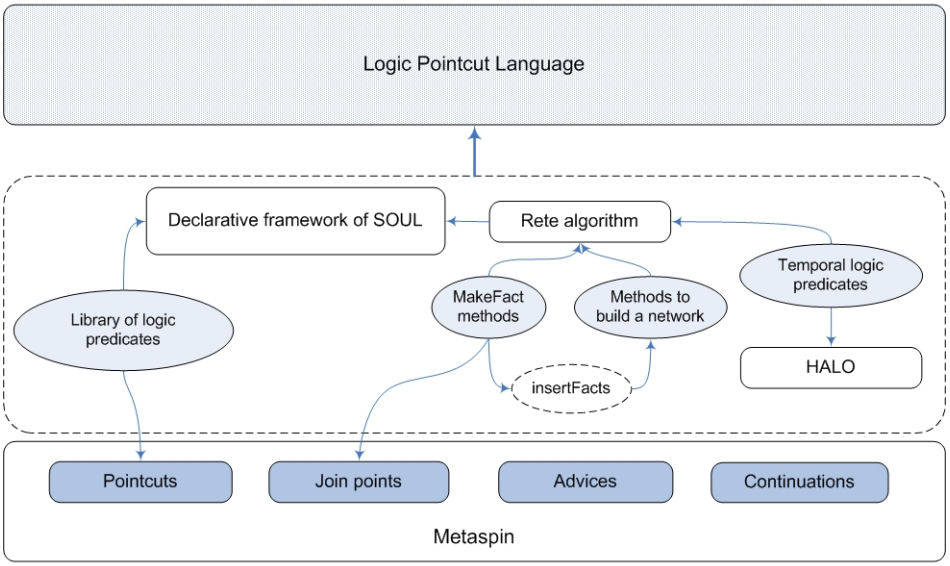

Figure 5.1: Schematic diagram of the logic pointcut language

creating a library of logic predicates, which now supports Metaspin pointcuts to be expressed and evaluated as queries. Figure 5.1 also shows the Rete algorithm implementation in SOUL (implemented in the Programming Lab at the Vrije Universiteit Brussel), and which also has an extension that supports HALO logic pointcut language for temporal logic. We combined this Rete implementation with Metaspin by adding some methods that allows to convert any join point into facts, and then insert them inside a network to be evaluated by the Rete algorithm.

Each of these components compose our logic pointcut language. We now describe in the next section how we can define pointcuts.

5.2.1 Defining pointcuts

The selector/advice binding definition is based on the structure used by Metaspin for this purpose. We can write an aspect in Metaspin as follows:

```
1 SelectorAdviceBinding
2     selector: [:jp | jp1 inAdviceExecution not and:
3                 [jp isMessageSendJoinPoint]]
4     advice: (Advice block: [ jp | jp proceedJoinPoint ])
```

This was the regular form of a pointcut in Metaspin, which uses Smalltalk block expressions for the execution of a pointcut (lines 2 - 3). We have combined it with SOUL to create a logic counterpart of this Metaspin structure. SOUL provided the means to define a library of logic predicates in Metaspin, which allows to write a pointcut as a query instead of as a block expression.

Defining a pointcut in our implementation is made by using a class called *CarmaSelectorAdviceBinding*, which uses the message *selector:advice:* to create an instance of that class. An aspect is then written like:

```
1 CarmaSelectorAdviceBinding
2     selector: 'not(inAdviceExecution(?jp)),
3                 isMessageSendJoinPoint(?jp)'
4     advice: (Advice block: [ jp | jp proceedJoinPoint ])
```

The pointcut (lines 2 - 3) is now expressed in a logic form (as a query). This format for writing pointcuts combines the infrastructure provided by Metaspin and the SOUL logic primitives. A method called *evaluator* was re-implemented to accept queries as arguments instead of Smalltalk expressions. This method only changed the way it takes arguments by delegating it from SOUL.

In this example, the pointcut is defined to match all the possible message send join points encountered in the execution of a program. When it finds a match, it will trigger the advice and will execute it by using the Metaspin advice evaluation process, which was explained in chapter 3. In this case, when the advice is triggered, the *proceedJoinPoint* is a message that allows to proceed with the next join point. The same will happen for each of the other join points matched during the execution of the program.

5.3 Defining predicates

The original Metaspin predicates do not allow to write logic pointcuts. However, based on the pointcut structure from Metaspin and the use of SOUL syntax, we defined the necessary rules to be able to write logic pointcuts as seen in the previous section. After this, the predicates can use queries instead of Smalltalk expressions to represent pointcuts.

The library of logic predicates consists of nine rules. These rules re-implement the methods that execute pointcuts in Metaspin by applying delegation to SOUL predicates on top of Metaspin join point methods, giving our logic pointcut language more expressiveness when defining a pointcut. The rules contain the logic structure and the behavior of a pointcut.

The first rule we explain comes from the method *inAdviceExecution*, which matches a join point that is going to be executed for the advice. This method will return either if there is a continuation for that advice or not. If there is a continuation, then the advice has to be triggered and executed. The predicate for this method is:

```
inAdviceExecution(?jp) if
   [?jp inAdviceExecution]
```

By coupling Metaspin with SOUL, we can define rules like this one. This predicate is of the order *inAdviceExecution/1* because it only takes one argument, which is the join point. Another example of a rule that can take either one or two arguments is the

isAssignmentJoinPoint. We have two predicates to handle this kind of join point: *isAssignmentJoinPoint/1* and *isAssignmentJoinPoint/2*. The first predicate checks only if an assignment is made in a join point, while the second predicate also checks if a value is bound to a variable used in that join point. These predicates are defined like:

```
isAssignmentJoinPoint(?jp) if
   [?jp isAssignmentJoinPoint]

isAssignmentJoinPoint(?jp, ?variableName) if
   isAssignmentJoinPoint(?jp),
   equals(?variableName,[?jp variableName])
```

After defining these rules, our logic pointcut language is able to express and execute pointcuts in the form of a query, like the next example:

```
1 MetaSpin add:
2         (CarmaSelectorAdviceBinding
3                  selector: 'not(inAdviceExecution(?jp)),
4                            isAssignmentJoinPoint(?jp)'
5                  advice: (Advice block: [ :jp | jp:=1 ])).
```

The example shows the use of the logic predicates inside the definition of a pointcut (lines 2 - 3). The different rules of our library of logic predicates are written with the same formula, and a review on the complete set of these rules is given in Appendix A.

5.3.1 Rete and SOUL

The original Rete algorithm was implemented as an extension of SOUL in the Programming Lab of the Vrije Universiteit Brussel (VUB). Rete is implemented as part of HALO logic pointcut language, and adds the possibility to evaluate temporal predicates and rules using the logic predicates provided by SOUL. This implementation of Rete has some extra features compared with the original Rete algorithm. Features like: join nodes that implement temporal operators, filter nodes that implement built-in temporal predicates, more optimizations to the algorithm, etc. HALO extended the Rete algorithm by adding these features.

5.4 Creating Facts

Now, we explain how we combine Metaspin with HALO and with the Rete network algorithm implemented in SOUL. We add this functionality to our implementation to be able to create facts for each of the join points created. This will add the functionality to process join points in a Rete network to have the possibility to apply rules over the execution history of those join points.

The use of Rete has the intention to create a better pattern-matching mechanism and to optimize the execution of pointcuts. It also allows to reason about a time line of the join points executed, this increases the number of logic predicates that can be applied.

To make this design possible, we have to implement some new methods. First of all, to achieve our implementation goal, we make use of polymorphism to be able to combine Rete with the Metaspin framework, as well as define methods for the same purpose.

We need to create some methods that could turn each kind of join point into a fact. To do this, we made use of a method called *new:holdsOn:* from the class ReteFact, which is part of the Rete implementation of SOUL. We defined a method called *makeFact*, which takes a join point as argument and turns it into a fact. The next example shows the method *makeFact* for the assignment join points:

```
(ReteFact new:#jpAssignment holdsOn: (Array
   with: self variableName
   with:(self continuation baseContext top)))
```

This method assigns an atom name *jpAssignment* to the fact, and assigns as arguments the name of the variable that is being assigned at that join point and the value. This *makeFact* method retrieves the information of a join point from the continuations created in Metaspin. Lets assume that the assignment we made was $x := 1$, the fact for this assignment join point is of the form:

$$'jpAssignment(x, 1)'$$

Each of the different join points that can be created in Metaspin are converted into facts in the same way. However, the method *makeFact* is adapted to the requirements of each join point. Now that we can make facts of the join points, we will explain how a Rete network is built in our logic pointcut language.

5.5 Building a Rete network

The Rete network can be built automatically when we execute a program in our logic pointcut language. The facts are created at the same time a join point is evaluated, and these facts are added at that same time into a Rete network. In this way, once we finish the execution of a program, we have a network with all the facts that were generated during the execution of a program.

We create a method called *addFactsToReteNetwork:*, which constructs a network by converting join points into facts. This method has the code:

```
1 addFactsToReteNetwork: aContinuation
2   | time aNet |
3
4   aNet := ReteNetwork restart.
5   (ReteCompiler new) buildNetworkFromCode:
6   self rulesForNetwork.
7   time:=1.
8   [aContinuation atEnd] whileFalse:[
9      aNet insertFact: (aContinuation currentJoinPoint
```

```
10                          makeFact) atTime: time.
11      time:=time+1.
12      aContinuation advanceStep].
13
14   ^ self retrieveInferredFacts: aNet.
```

First, we initialize a Rete network inside a variable called *aNet* (line 4). Then, we add the specific rules that evaluate that network (lines 5 - 6). The rules are added using a method called *rulesForNetwork*, which allow us to define all the rules that we wanted to be applied over a certain network. A *time* variable (line 7) is initialized with the value one, just for the case when we need to use the time of the creation of the facts. After that, we add the facts into the network (line 8 - 12), while at the same time, the conversion of the join points evaluated in Metaspin into facts is made (lines 9 - 10). At last, once all the facts have been added to the network, the method returns the inferred facts (line 14) that were the result of applying the rules to the facts.

The network evaluates the facts against the rules defined for that same network. The *addFactsToReteNetwork:* method is implemented by the combination of Metaspin and Rete to allow our implementation to build a network from the join points evaluated. However, for a better understanding on how this works, we show an example of an evaluation of a network:

```
1 aContinuation:=MetaspinContinuation atContext:
2       [ | x | x := 1. x := 2. x := 3. x := 4 ].
3 MetaSpin add:
4   (SelectorAdviceBinding
5       selector: 'isAssignmentJoinPoint(?jp)'
6       advice: (Advice block: [ :jp | 42 ])).
7
8 self rulesForNetwork: 'adviceap1(?x) if
9                       jpAssignment(x,3),
10                      allpast(jpAssignment(?x,2))'.
11 aNet := self addFactsToReteNetwork: aContinuation.
```

In this example we use a variable called *aContinuation* (lines 1 - 2) to bind some assignments to be executed in Metaspin. Then, we add a pointcut (lines 3 - 6) that will match all the join point assignments encountered during this execution. After that, we define a HALO pointcut (lines 8 - 10). In this case we use the temporal logic predicate *allpast*, which searches all the join points in the execution history that are bounded to a certain variable and value. Finally, to evaluate the network, we execute the method *addFactsToReteNetwork* (line 11). This method generates the facts from the join points evaluated with Metaspin, and obtained a result by evaluating those facts against the rules defined before (HALO pointcut).

This example executes one advice, which is $'jpAssignment(x, 2)'$. As we can see from the rules defined, this is the only result possible as we only have one assignment of $x := 2$ before the assignment $x := 3$.

5.6 Summary

In this chapter, we described how we designed and implemented our logic pointcut language by using Metaspin as the base framework. Defining a library of logic predicates with SOUL on top of Metaspin was the first step to write logic pointcuts. We also explained the syntax to write aspects and how we coupled Metaspin and SOUL.

The use of a Rete network adds extra features to our implementation, like the increase of expressiveness as well as the use of temporal predicates to evaluate join points in the past. By adding Rete, we extended the pattern-matching mechanism and the possibility to evaluate pointcuts with HALO temporal predicates.

In the next chapter we show an evaluation of our implementation of a logic pointcut language.

Chapter 6

Evaluation

In this chapter, we describe an experiment that we have done with our implementation. It is based on an example of a shop application in Smalltalk. We briefly describe the shop application and its purpose. This shop application is used to evaluate memory and time efficiency on temporal pointcuts. We show how the implementation of the logic pointcut language measures the criteria.

6.1 Shop example

This section briefly describes the application that allows us to experiment with our implementation. The shop application is a small application that simulates an e-shop, and can do the following:

1. Register a new article with a name and a price.
2. Login.
3. Buy an article and add it into a basket.
4. Checkout to end a session.

We can start this application by creating a few articles that will be available in the shop. Then, we can simulate that one or more users log in the shop. After that, each logged user can buy different articles, and those articles are going to be added to the basket of each user. Finally, the checkout option of the application simulates the end of a session. Figure 6.1 shows the class diagram of the shop application.

6.2 Memory evaluation using a Rete network

We use the shop application to evaluate our logic pointcut language. First, all the join points generated from the shop application are converted into facts and we apply a set

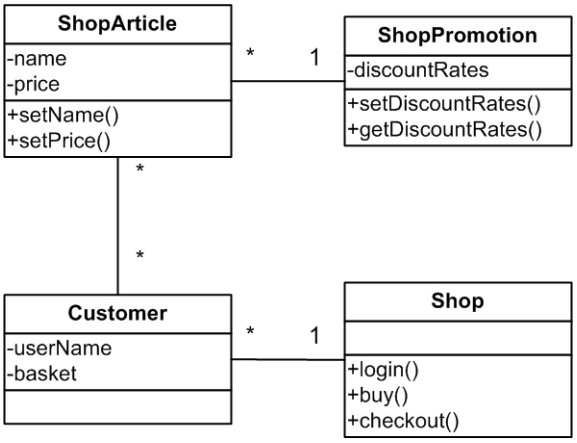

Figure 6.1: Shop application class diagram

of rules (HALO pointcut) over those facts. The facts that match the HALO pointcut are inserted into a Rete network.

After building a network from the shop example, we can perform an experiment that focuses on the memory and time performance of the Rete network with and without garbage collection. In this case, we apply this experiment to obtain the number of entries of the memory tables from the network nodes, so we can evaluate the differences on memory and time performance of a network when the garbage collection is active, and when it is not.

First, we show the examples we used to evaluate the experiment. The next example consists of three articles registered in the shop and two users logged in. We apply some rules to obtain the last article put in shopping cart when a user ends the session. This example is used to introduce the shop application and the setup of the experiment.

```
1 aContinuation:=MetaspinContinuation atContext: [
2     a1:=ShopArticles newWithName: 'book' andPrice: 23.
3     a2:=ShopArticles newWithName: 'cd' andPrice: 24.
4     a3:=ShopArticles newWithName: 'backpack' andPrice: 24.
5     c:=Customer new login: 'bob'. c buy: a1. c buy: a2. c checkout.
6     c2:=Customer new login: 'kris'. c2 buy: a3. c2 checkout.
7     ].
8
9 HaloCarmaTest new rulesForNetwork:
10                         'adviceap1(?user,?article)
11                         if jpMessage(checkout,?user),
12                            mostrecent(jpMessage(buy,?user,?article),
13                            mostrecent(jpMessage(login,?user)))';
14  addFactsToReteNetwork: aContinuation.
```

In the example, we register three articles in the shop: *book*, *cd*, and *backpack*, with their respective price (lines 2 - 4). Then, a customer *bob* logs in the shop, buys a *book* and a *cd*, and makes a checkout to end its session (line 5). After that, a customer *kris* logs in, buys a *backpack*, and makes a checkout to end its session (line 6). This example is evaluated in

Metaspin. After that, we define some rules with temporal logic to obtain the last article bought by each customer (lines 9 - 13). The next step is to generate the Rete network by adding the join points as facts (line 14). Finally, we obtain as a result that the last article bought by the customer *bob* was a *cd*, and the last article bought by the customer *kris* was a *backpack*, which were the result of evaluating the rules against the facts in the Rete network. Figure 6.2 shows the Rete network with the memory tables associated to its nodes.

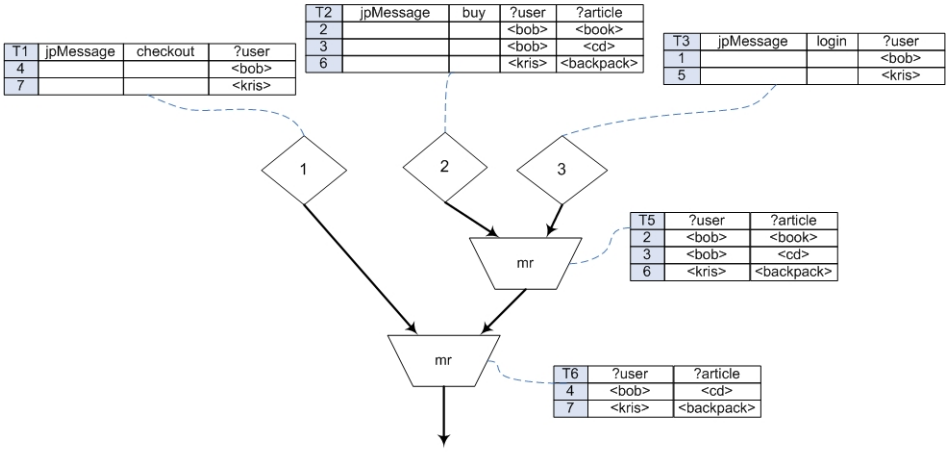

Figure 6.2: Rete network from shop application

Now that we have the Rete network, the experiment evaluates the information contained in the memory tables of a that network. The goal of this experiment, is to obtain the performance of a network when the garbage collection is used, and when it is not used. We want to measure memory. We apply the example of the shop application with the temporal logic predicate *most-recent*. The file with the data from the network also contained the evaluation time (in milliseconds) of each join point.

6.2.1 Evaluation of most-recent predicate

We want to illustrate the results of an example by using a graph. First, some articles and users are defined for the shop application. Then, different users are logged in the shop randomly, and the purchases of different articles, as well as the end of a session are also made randomly. We used a program to generate all this data randomly, for the sake of the simplicity we do not show this program. However, we use it to be able to gather enough information to perform the experiment.

Figure 6.3 shows a graph that were generated from the data of an example that randomly generated login and buy events. This example uses the garbage collection, and it was evaluated against the following rules (HALO pointcut) that uses the temporal predicate *most-recent*:

```
'adviceap1(?user,?article)
if jpMessage(checkout,?user),
   mostrecent(jpMessage(buy,?user,?article),
   mostrecent(jpMessage(login,?user)))'
```

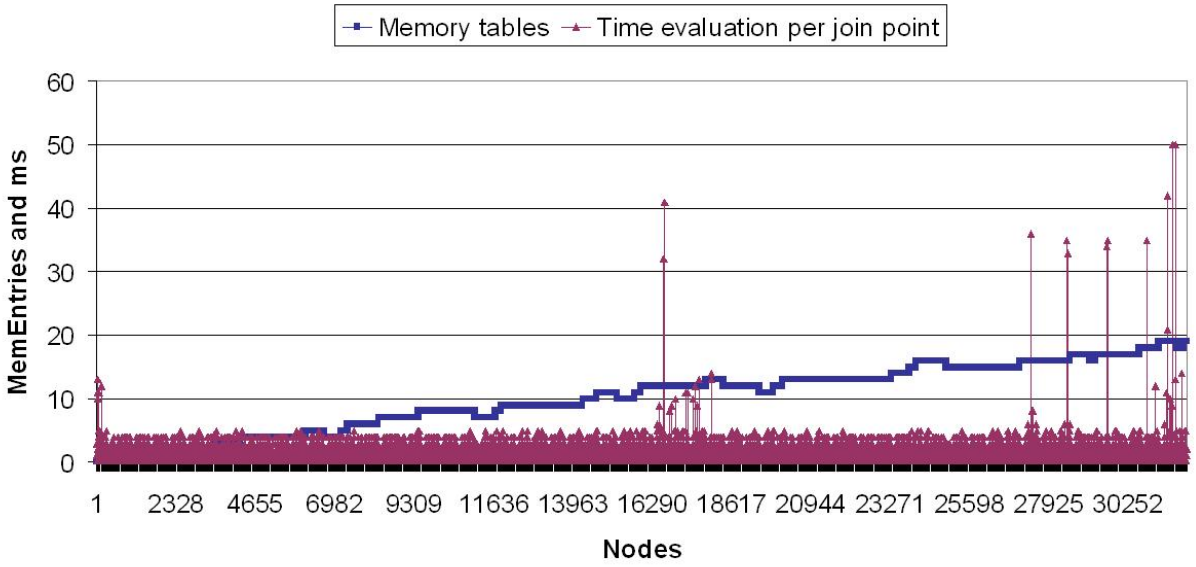

Figure 6.3: Memory tables and timing information from the Rete network with garbage collection and temporal predicate most-recent

The graph shows that the memory entries of a Rete network vary during the evaluation of the shop application. We can notice than in some join points the data from the memory decreases. This happens due to the fact that if a join point is matched, the garbage collector discards the information that did not match before, and that it is not necessary to be kept anymore because the *most-recent* predicate only needs the last matched join point. The graph also shows that the time performance is more or less constant along the experiment.

We did not expect to obtain these results from the graph. The memory entries should not keep increasing after a certain number of facts are evaluated because there are no new users added. This means that after a certain time, the garbage collection should keep the number of memory entries constant. This shows that there is still a problem with the implementation of the garbage collection in the Rete network.

6.2.2 Evaluation of all-past predicate

We applied a second experiment using the shop application. This time we used the temporal predicate *all-past* with garbage collection, and the following rules:

```
'adviceap1(?user,?article)
if jpMessage(checkout,?user),
```

```
3     allpast(jpMessage(buy,?user,?article),
4     allpast(jpMessage(login,?user)))'
```

Figure 6.4 shows the results of evaluating an example of the shop with the *all-past* predicate. We can see from the graph that the memory entries of the Rete network keeps increasing during the evaluation of the join points. This resulted because even when we used garbage collection, all the data has to be kept in the network to be used when needed. This is the behavior of the temporal predicate *all-past*, which needs to have all the history execution data to be able to match all the join points evaluated in the past.

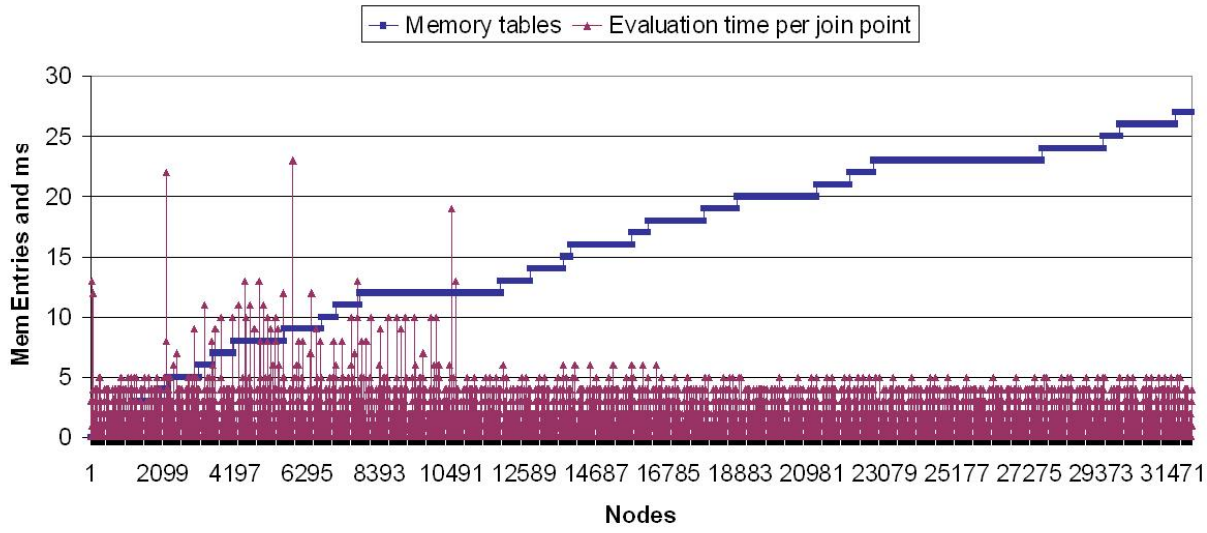

Figure 6.4: Memory tables and timing information from the Rete network with garbage collection and temporal predicate all-past

6.3 Summary

This chapter presented an example of a shop application. We made use of this example to experiment with the implementation of the logic pointcut language in Metaspin. We evaluate the memory efficiency and the timing information of a Rete network with the shop application.

After evaluating some examples on the implementation, we generate some graphs to illustrate the performance with garbage collection and without it. We obtain as a result that there are big differences in the use of garbage collection between the temporal predicate *most-recent* and the temporal predicate *all-past*. This can lead to a future optimization of the temporal predicates. We noticed from the graphs that there is still a problem with the garbage collection, because the entries of the memory tables keep increasing along the evaluation of an example.

Finally, this chapter allowed us to prove that our implementation helps to experiment with aspect languages. The next chapter presents the final conclusion of this dissertation, as well as the future work.

Chapter 7

Conclusion

We implemented a logic pointcut language in the Metaspin framework to experiment with aspect languages and help understanding how they work in the inside. In this chapter we give a summary of how we did this.

First of all, in chapter 2, we introduce the concept of aspect-oriented software development and the reason of its existence. We gave the basic concepts of aspect-oriented programming. We presented a brief introduction of logic meta programming, as well as logic pointcut languages. After that, we described the different approaches of logic pointcut languages that have been developed. Then, we concluded this first part of the dissertation with a brief description of the main features of the existing logic pointcut languages.

Then, we presented a detailed description of Metaspin in chapter 3. We gave a complete explanation on the bytecode interpreter from Smalltalk, as Metaspin makes use of it to evaluate a program. Then, we explained Metaspin, a framework to experiment with programming languages for aspect-oriented programming. After that, it followed a full description on how Metaspin bytecode interpreter works. Finally, we gave some examples that complemented and showed the description of the Metaspin framework.

Right after describing Metaspin, chapter 4 introduced logic meta programming and temporal logic, as well as the approaches we used together with Metaspin to build the implementation of the logic pointcut language. Here, we first explained logic programming and its basic concepts. After that, we described logic meta programming, as it is the underlying technology to build a logic pointcut language. Then, we introduced SOUL, the *Smalltalk Open Unification Language* designed for logic meta programming. SOUL is a symbiosis between a logic language (Prolog) and an object-oriented language (Smalltalk), and is the language we used as the declarative framework to write logic pointcuts in Metaspin. Then, we described temporal logic meta programming, its basic concepts, and we gave as an example the logic pointcut language HALO, which we described in the beginning of the dissertation. Finally, we introduced the Rete algorithm that is used to implement HALO, with extensions to support temporal logic.

In chapter 5, we presented a detailed description of the implementation of a logic pointcut language in Metaspin. In this part of the dissertation, we gave a high-level diagram of the design of the implementation that showed how the different approaches

were connected with each other. Based on this high-level design, we described in detail how SOUL was coupled to Metaspin to enable writing logic pointcuts. For this purpose, a library of logic predicates was created using the declarative framework of SOUL and the framework from Metaspin. We described how to define predicates by means of examples. Then, we explained how the implementation converts into facts the join points evaluated in Metaspin, as this is necessary for the use of HALO and the Rete algorithm. After, we described in detail the creation of a Rete network from the facts generated in Metaspin. This was used to evaluated temporal logic pointcuts with HALO predicates. Each of the explanations included examples that showed how we combined the different approaches (Metaspin, SOUL, HALO, and Rete algorithm) to implement a logic pointcut language.

Finally in chapter 6 we showed an experiment we did with our implementation. The evaluation criteria for this experiment focused on memory and timing performance. We wanted to know the memory performance of a Rete network with garbage collection and without it. Therefore, we presented a shop application as an example. Based on that application, we run some examples to obtain the memory tables from the Rete network and to be able to measure the efficiency of the network in terms of memory and timing. We presented some graphics that visualized better the results of the experiment, and allowed to interpret those results.

This dissertation discussed the implementation of a logic pointcut language in the Metaspin framework, which is a framework for experimenting with aspect languages. It showed the usefulness of an implementation like this to help to understand better aspect languages, and to evaluate performance or efficiency of a program.

7.1 Technical contributions

This dissertation provided two implementations:

- The creation of a library of logic predicates for Metaspin, using SOUL declarative framework.
- The combination of a Rete network with Metaspin framework. It is possible to build a Rete network with the join points evaluated in Metaspin, and execute HALO temporal predicates for pattern matching.

This two implementations were the two most important technical contributions. They provided the means to understand better an aspect language and to experiment with it.

7.2 Future work

In this section, we present the possible future research related to this dissertation. The use of Metaspin as a framework, opens the possibility for more research on logic pointcut languages.

First, our implementation can be improved by increasing the join points structure from Metaspin. This framework can be improved by adding a bigger structure that could match more pointcuts. It is also possibly to add a different approach to increase the functionality of the implementation. So far, we included logic pointcuts and temporal logic pointcuts to reason about past join points. However, we can include some features from the existing logic pointcut languages to design new logic pointcut languages and to experiment with them.

A second possible research related to this dissertation is the optimization of the Rete network. There are still improvements that can be done, like improving the speed of the evaluation of facts inside a network. Therefore, a future research could be optimize the garbage collection by taking in account weak references. It is also possible to improve the garbage collection in the sense of the temporal predicates applied. For example, the *all-past* predicate keeps track of all the facts in the past. Nevertheless, this can be optimize by adding a time stamp, which says that the memory tables from a Rete network should be clean after certain period of time.

Appendix A

Library of logic predicates

This appendix gives a review of all the rules defined to create a library of logic predicates. Our logic pointcut language makes use of these rules to express pointcuts as queries.

The rule *isBlockReturnJoinPoint* was defined to be able to express a block return join point. The rule is:

```
isBlockReturnJoinPoint(?jp) if
     [?jp isBlockReturnJoinPoint]
```

The general formula to define a rule is to show that the join point is bound to the variable ?*jp*, and then ask if that variable, which contains a join point, is the specific join point for that rule. So, in the case of the *isBlockReturnJoinPoint* predicate, the rule is asking if the content of the variable ?*jp* is a block return join point.

The next rule has two possible predicates: *isMessageSendJoinPoint/1* and *isMessageSendJoinPoint/2*, which have one and two arguments respectively. The *isMessageSendJoinPoint/1* rule uses the general formula to define a rule, but the *isMessageSendJoinPoint/2* adds a second argument to the rule. This second argument is the variable ?*message* which is also bound to the join point. This rule was added to check if a message is being sent in this join point. The rule asks if the message send join point is bound to the variable ?*jp* and if the message that is bound to the variable ?*message* is equal to the message in the join point. The two predicates are defined like:

```
isMessageSendJoinPoint(?jp) if
     [?jp isMessageSendJoinPoint]

isMessageSendJoinPoint(?jp, ?message) if
     isMessageSendJoinPoint(?jp),
     equals(?message,[?jp message])
```

Now that we explained these two predicates, we can say that some of the rules are defined with only one predicate, while others need two predicates to be able to handle two arguments. For example, the rule for *isReferenceJoinPoint* only needs one predicate because it does not need more than one argument to be expressed. The definition of the rule is:

```
isReferenceJoinPoint(?jp) if
     [?jp isReferenceJoinPoint]
```

This rule checks if a reference join point is bound to the variable ?*jp*, and with this we can write a logic pointcut for this kind of join point.

The next rule for *isReturnJoinPoint* can have two predicates: *isReturnJoinPoint/1* and *isReturnJoinPoint/2*, just as the same as the *isMessageSendJoinPoint*. It is define like:

```
isReturnJoinPoint(?jp) if
     [?jp isReturnJoinPoint]

isReturnJoinPoint(?jp, ?returnValue) if
     isReturnJoinPoint(?jp),
     equals(?returnValue,[?jp returnValue])
```

This rule allows us to write a logic pointcut for the return join points. The second rule for this kind of join point can receive a return value from a method as a second argument, and it will ask if that value is the same as the value from the join point.

Finally we show the three rules that were introduced in the main text in section 4.5.2. The rules are:

```
inAdviceExecution(?jp) if
     [?jp inAdviceExecution]

isAssignmentJoinPoint(?jp) if
     [?jp isAssignmentJoinPoint]

isReturnJoinPoint(?jp, ?variableNme) if
     isAssignmentJoinPoint(?jp),
     equals(?variableName,[?jp variableName])
```

Obviously, these rules are defined in the same way as the others. The rule for *inAdvice-Execution* matches a join point that is executed for the advice, this means that the rule returns if there is a pointcut continuation or not, and in the case of a continuation, the advice is executed. The rules for *isAssignmentJoinPoint* can receive one or two arguments. The rule with one argument checks if we are making an assignment and a join point of this kind is being evaluated. The rule with two arguments also checks if the name of the variable is equal the variable of the join point.

We note that the use of SOUL with Metaspin allows to define more general rules to write logic pointcuts. This characteristic increases the expressiveness of the language because of the use of logic expressions.

Bibliography

[1] Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Semantics of static pointcuts in aspectj. In Matthias Felleisen, editor, *Principles of Programming Languages (POPL)*. ACM Press, 2007.

[2] J. Brichau, M. Mezini, J. Noye, W. Havinga, L. Bergmans, Vaidas Gasiunas, C. Bockisch, J. Fabry, and T. DHondt. An initial metamodel for aspect-oriented programming languages. In *AOSD-Europe Deliverable D39, AOSD-Europe-VUB-12*, pages 1–26, 2006.

[3] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[4] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[5] Peter Flach. *Simply Logical*. John Wiley, April 1994.

[6] LogicAJ (Logic Aspects for Java). http://roots.iai.uni-bonn.de/research/logicaj.

[7] Charles L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. pages 324–341, 1990.

[8] Antony Galton. Temporal logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2003.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[10] Tobias Rho Gnter Kniesel. Generic aspect languages - needs, options and challenges, jfdlpa 2005. Sep 2005.

[11] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[12] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM.

[13] Charlotte Herzeel, Kris Gybels, and Pascal Costanza. A temporal logic language for context awareness in pointcuts. In *Workshop on Revival of Dynamic Languages*, 2006.

[14] Charlotte Herzeel, Kris Gybels, Pascal Costanza, and Theo D'Hondt. Modularizing crosscuts in an e-commerce application in lisp using halo. In *ILC 2007: Proceedings of the International Lisp Conference 2007*. ACM, 2007.

[15] Gregor K Iczales, John I Rwin, John Lamp Ing, Cr Istina V Ideira Lopes, and Chr Is. Aspect-oriented programming. 1997.

[16] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP*, pages 195–213, 2005.

[17] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *Foundations of Aspect-Oriented Languages workshop (FOAL'05), Chicago, USA*, 2005.

[18] Robert A. Kowalski. *Logic for Problem Solving*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1979.

[19] Pattie Maes. Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155, 1987.

[20] AspectJ Crosscutting objects for better modularity. http://www.eclipse.org/aspectj/.

[21] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Andrew P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*:pages 214–240, Springer, 2005.

[22] Leon Sterling and Ehud Y. Shapiro. *The art of Prolog : advanced programming techniques*. Logic programming. MIT Press, Cambridge, Mass., 2nd edition, 1994.

[23] Gnter Kniesel Tobias Rho. Independent evolution of design patterns and application logic with generic aspects - a case study, technical report iai-tr-2006-4, computer science department iii, university of bonn. In *Technical Report IAI-TR-2006-4, Computer Science Department III, University of Bonn*. April 2006.

[24] David A. Watt. *Programming language concepts and paradigms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[25] AOSD Europe Webiste. http://www.aosd-europe.net/.

[26] CARMA Website. http://prog.vub.ac.be/ kgybels/research/aop.html.

[27] SOUL Website. http://prog.vub.ac.be/soul/index.html.

[28] Roel Wuyts. Declarative reasoning about the structure object-oriented systems. In *Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.

[29] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, 2001.