Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica en Toegepaste Informatica

A Temporal Logic Language for Context-Dependence in Crosscuts

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Informatica

Door: Charlotte Herzeel
Promotor: Prof. Dr. Theo D'Hondt
Adviseurs: Kris Gybels, Dr. Pascal Costanza
Juni 2006

Acknowledgements

This dissertation would simply not have come to be without the help of many. I thank Prof. Dr. Theo D'Hondt for promoting my thesis and for giving me the opportunity to go to AOSD 2006 to present part of this work. Much gratitude goes to my advisers Kris Gybels and Dr. Pascal Costanza, not only did they provide me with a subject for this dissertation, they helped me throughout the whole process. I not only thank them for them for taking the time to proofread this dissertation, but more importantly for the countless hours they invested in discussing my thesis and for being so generous in sharing their thoughts. If anything, I hope they are not disappointed in the results. Special thanks go to Coen De Roover, for proofreading a large part of this dissertation. Furthermore, I thank Brecht Desmet and Elisa Gonzalez Boix for their various useful comments on this work and for looking after me at AOSD 2006. I also want to take the opportunity thank my parents and my brother for motivating and supporting me during the past years, though this was not always easy for them. Last, but not least, I thank my friends Philippe, Yannik, Lieven and Jo for all their shenanigans involving pirates, ninjas and monkeys and making the past years at the VUB a pleasant experience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this dissertation we investigate how a logic meta programming approach based on temporal logic can be used to reason about program execution and past program state. The result of this research is the specification and a (non straightforward) implementation of a declarative pointcut language dedicated to writing pointcuts that relate multiple join points in the (past) execution of a program.

Successful software is subject to reuse: a computer program's lifetime is stretched by the many (re)releases, that hopefully add functionality and eliminate bugs compared to earlier versions. Updating old software is often hard – especially software that was written by other people; All programmers know that uncomfortable feeling they get when they skim through source code – often badly documented – in the hopes of finding that piece of code they need to adapt or use to implement their desired changes in a program. Therefore it is of utmost importance that programs are carefully designed, so that it is easy (for other programmers) to understand the program structure and its relation to the program requirements or concerns, which is the case when each concern is addressed *separately* in the program. This is an important principle in software engineering known as "separation of concerns".

Currently, separation of concerns is pursued through program modularisation: programmers break up a program into different program requirements or concerns that combined reflect the program's desired functionality and then they try to map these concerns onto separate modules, where modules are the basic abstraction mechanisms a programming language offers, such as classes, methods, functions, procedures, packages etc. However, *full* separation of concerns is hard to achieve using a traditional object-oriented, procedural or functional language.

Full separation of concerns through modularisation is difficult because a program can only be modularised in one way at a time, possibly matching very well for particular concerns, but other concerns that do not align with this modularisation end up scattered over different modules: concerns that do not align with a particular modularisation are called crosscutting concerns. Many types of crosscutting concerns exist: logging, synchronization, profiling, error handling to name just a few.

The implementation of a crosscutting concern is scattered over different modules and leads to tangled code, because modules now implement multi-

ple concerns (partly). Scattering and tangling makes programs harder to read, maintain and reuse, because programmers looking to adapt one concern are forced to read, maintain or discard pieces of code that have nothing to do with that particular concern, but just happen to be stuffed in the same module. A new paradigm called *aspect-oriented programming* emerged to cope with this problem through modularisation.

Aspect-oriented programming is all about modularising crosscutting concerns. An aspect language is a language extension for a base language, such as Java or Smalltalk, that introduces new constructs that allow the implementation of crosscutting concerns in distinct modules, called *aspects*. The ability of an aspect language to support the modularisation of crosscutting concerns depends on its *join point model*.

An aspect language's join point model consists of *join points*, *a means of describing join points* and *a means of affecting behavior at a join point*. Two types of join points exist in the space of AOP: dynamic join points, such as method calls, refer to events in the execution of a program and static join points refer to places in the program code, such as the body of a method. The AOP idea is then that one specifies the join points a crosscutting concern's behavior affects and that the crosscutting behavior is assured at these places, rather than implementing the crosscutting behavior by producing scattered or tangled code. The means for describing join points is called a *pointcut language*, where a pointcut can be seen as a predicate over all join points in a program, to pick out join points of interest.

The expressiveness of an aspect language depends of course on the variety of join points that are included in its join point model, but more importantly on the expressiveness of the pointcut language for describing these join points and the composition mechanisms for composing complex pointcuts out of simpler pointcuts. A logic meta programming approach to aspect-oriented programming proposes the use of a full-fledged logic programming language as a pointcut language with a built-in library of predicates that are pointcut predicates for the individual join points in the join point model: this approach is recognized to be a very *declarative* and *open* view on pointcut languages.

Although early research in AOP focused on aspects that are triggered at a single join point, recent research has evolved towards aspects that are triggered based on the occurrence of a series of join points in the execution of a program and past program state: they were dubbed event-based aspects, stateful aspects and context-aware aspects. However, currently, no (sufficiently expressive) pointcut language that allows to write pointcuts, to be used in the implementation of these sorts of aspects, exists.

Program execution generates a (dynamic) join point at each computational step: all the join points that are generated this way, can be placed on a time line, reflecting the order in which they occurred, and we observe that there exists a temporal relation between any two join points on this time line. If we can express this temporal relation, we can describe the sequence of events in a pointcut. A formalism dedicated to reasoning about time and temporal relations, is *temporal logic*.

Temporal logic is a term being used to denote a series of logics of *qualified truth*. In temporal logic formulas are evaluated in terms of an implicit temporal context. A temporal logic adds new types of logic connectives called *temporal operators* that abstract the explicit handling of time in a formula. Depending

on the definition of time applied, these temporal logics have a different meaning. For example, there exists a form of *metric temporal logic* where time is defined as a linear sequence of time points and the temporal operators express temporal relations between formulas in terms of intervals of time points. As defended in this dissertation, this particular form of metric temporal logic, is a solid basis for designing (and implementing) a logic pointcut language that can be used to describe sequences of (constrained) join points.

The remainder of this dissertation is dedicated to the description of a temporal logic pointcut language, named HALO, as a declarative means to write down pointcuts that match part of the execution history of a program. In HALO, a pointcut about multiple (past) join points is evaluated in respect to the the program state at these (past) join points, so that constraints about past join points are checked against the program state at the occurrence of a past join point. This makes it possible to reason about current and past program state in HALO, and furthermore, it is possible to put constraints on a past join point in a pointcut, that involve values from a later join point. The language's applicability is evaluated by implementing some example promotional aspects for an e-commerce application, that exploit HALO's expressiveness.

1.1 Document Overview

This dissertation consists of seven parts. In the next chapter we give an introduction to aspect-oriented programming and present an overview of current AOP systems that allow one to write aspects based on the occurrence of a sequence of join points, but lack dedicated pointcut languages.

The third chapter gives an introduction to logic meta programming and its applications, including aspect-oriented programming; This approach is the basis of our problem solution: we add temporal logic to the formalism as a means for reasoning about program execution.

A next chapter investigates the use of (temporal) logic meta programming as a basis for our pointcut language and as an experiment we use the language to implement aspects for an e-commerce application and a video game application. We present a non straightforward implementation, involving a dedicated forward chainer, in the subsequent chapter.

We continue the dissertation by giving an overview of related work. Finally, a last chapter in this dissertation summarizes our results and outlines future work.

Chapter 2

Aspect-Oriented Software Development

2.1 Overview

In this chapter we give an overview of *aspect-oriented programming* (AOP) as a solution for implementing *crosscutting concerns*, introduced by Kiczales et. al. in the paper titled "Aspect Oriented Programming" [23].

In a first section we explain what crosscutting concerns are and we introduce the concepts behind AOP and aspect languages, being join point models, aspects, join points, pointcuts, advices and weavers. Next we cover two existing aspect languages, namely AspectJ and CARMA to illustrate these concepts.

A subsequent section deals with new types of crosscuts and the types of aspects to develop them, namely stateful aspects and context-aware aspects. We review a couple of existing AOP systems that allow one to implement stateful or context-aware aspects (EAOP, JasCo, Reflex).

To conclude the chapter we present our conclusions and a summary of the chapter.

2.2 Separation of concerns

The way programmers design an application is by problem decomposition: they break up the program into distinct requirements or concerns that overlap in functionality as little as possible. This has an important positive consequence on their design, namely that one concern does not depend on another concern and consequently they do not need to know about one concern to understand another and they do not need to consider other concerns when changing one [1]. Next step in obtaining a program, is implementing it.

2.2.1 Modularisation

The implementation of a program results in mapping the different concerns from the problem decomposition onto separate modules, where modules are the basic

[1] This strategy was dubbed "Separation of concerns" by Dijkstra. [45]

abstraction mechanisms a programming language offers: this mapping is known as program modularisation. Depending on the target programming language's paradigm, different types of modules are offered; An object-oriented language offers classes and methods and inheritance as a composition mechanism, whereas a functional language offers functions to abstract a concern. Inherently, problem decomposition is tightly coupled to the target programming language's paradigm. Full separation of concerns is not easy to achieve through modularisation using a traditional language [2], because

> A program can be modularised in only one way at a time, and the many kinds of concerns that do not align with that modularisation end up scattered across many modules.

This property is known as the"Tyranny of the Dominant Decomposition" [38].

2.2.2 Crosscutting concerns

The concerns that do not align with a particular modularisation are called *crosscutting concerns* [23]. Crosscutting concerns are said to be *scattered* over different modules which means their implementation is spread over different modules, which results in *tangled* code, because now one module implements multiple concerns. Examples of crosscutting concerns are error handling, synchronisation, profiling, logging etc. To make the concept less abstract, we discuss a crosscutting concern for a banking application implemented Java.

Example: logging sensitive operations in a banking application.

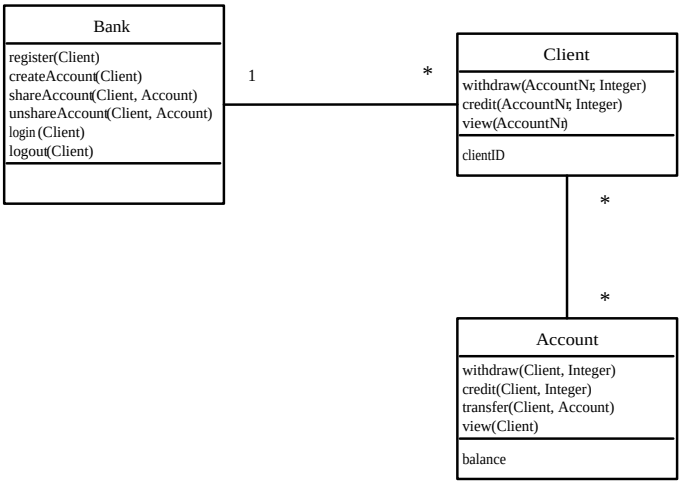

Figure 2.1: UML for a banking application.

Consider the UML diagram of a banking application as shown in Figure 2.1: a bank has many clients who can create and share accounts to manage their money by making deposits and withdrawals: this is the program's main

[2]Object- oriented, functional, procedural.

functionality and this is reflected in the different classes and methods. Another concern in the banking application is security: sensitive operations – e.g. operations that affect an account's balance – need to be logged. The application programmer identifies two sensitive operations, implemented by the `withdraw` and `deposit` methods and needs to extend these methods to include the logging behavior, which the programmer can achieve by inserting an extra statement in the body of the method definitions (Figure 2.2); Worst case scenario, this is repeated in other classes for other sensitive operations (e.g. for the `shareAccount` method).

```
public class Account
{
  private int balance;

  public void withdraw(Client client, int money) {
    /* sensitive operation */
    balance = (balance - money);
    /* create message */
    String msg = "Withdraw called by ";
    msg += client.getClientID()
    log(msg);
  }

  public void credit(Client client, int money) {
    /* sensitive operation */
    balance = (balance + money) ;
    /* create message */
    String msg = "Credit called by ";
    msg += client.getClientID();
    log(msg);
  }

  public void log(String message) {
    Date now = new Date();
    message += now.toString();
    system.out.println(message);
  }

  public void view() {
    system.out.println(balance);
  }

}
```

Figure 2.2: Logging sensitive operations.

The problem with this approach, is that one concern, namely logging of sensitive operations, is implemented across several methods (scattering). This is bad because when the programmer wants for example to reuse one of the methods to implement a method in a subclass, she also inherits the logging concern's behavior, which has nothing to do with the behavior the method is named after or when she needs for example to debug such a method, she needs to read through the logging code that has nothing to do with the regular behavior of a sensitive operation (tangling). In short: crosscutting concerns make code harder to reuse and maintain. This problem can not simply be solved by inheritance.

One can try to solve this problem using inheritance: make a sub class `LoggedAccount` for the class `Account` and in the class `LoggedAccount` implement two methods `withdraw` and `deposit` that re-implement these methods using super sends. The problem is that there can be more than one class with

operations that need to be logged (e.g. method `shareAccount` in class `Bank`.), one would need to make special sub classes for these classes too (e.g. a class `LoggedBank`) which implies the logging concern will still be scattered over different modules. In the next section we introduce AOP as a way of modularising crosscutting concerns such as the logging example.

2.3 Modularising crosscutting concerns via aspects

The goal of AOP is to modularise crosscutting concerns by introducing new types of modules (abstractions) in an existing programming language, called base language, that can be used to implement these so-called crosscutting concerns separately: the resulting language extension is called an aspect language [3]. An AOP-based implementation of an application consists of a base application (a program written in the base language implementing the concerns that can be modularised) and a bunch of aspects written in the aspect language: it is up to a *weaver* program to combine the aspects and the base program. The question when designing an aspect language is of course, how does one go about modularising a crosscutting concern and can we abstract this in new language constructs?

2.3.1 Join Point Models

The ability of an AOP language to support a modular implementation of crosscutting concerns depends on its *join point model* [27] [22]. A join point model consists of three concepts:

1. *Join points* are the points of reference in a base program which a crosscutting concern affects. We distinguish two types of join points, namely lexical join points (referring to a location in the program text (e.g. the body of a method)) and dynamic join points (referring to an event in the execution of a program (e.g. a method call)).

2. *A way to describe (or quantify) multiple join points*, often called a *pointcut*. A pointcut can be seen as a predicate over all join points in a program, to pick out join points of interest: e.g. "The bodies of the methods in class A" or "All the calls to method A."

3. *A means of affecting behavior at a join point* e.g. "Synchronize this method" or "Execute this piece of code".

2.3.2 Weaving

The weaver is responsible for making sure that aspects are added to a base program. There are two ways of doing this:

static weaving : the program in the base language is compiled to a new program (program transformation). Advantage: No overhead at runtime.

[3] We follow [23] where AOP is achieved through language extension; there are however approaches that propose frameworks for implementing crosscutting concerns [35] [5]

Disadvantage: Aspects can not be turned on/off at runtime, base code needs to be recompiled when aspects are added/ removed.

dynamic weaving : the base language interpreter is aware of aspects and divides program execution into join point events: for each event possible pointcuts are evaluated for truth and if true, the related piece of code is wrapped around the join point event in the execution. Advantage: aspects can be turned on/off at runtime, aspects can be added/removed at runtime. Disadvantage: runtime overhead.

In the next section we analyze two different aspect languages, namely AspectJ (for Java) and CARMA (for Smalltalk) to clarify the concepts discussed so far in this section.

2.3.3 Case study: AspectJ

In this section we introduce AspectJ [39], which is a general-purpose aspect language for Java. We discuss the language in terms of its join point model (see section 2.3.1).

Join Points

AspectJ is a general-purpose aspect language for Java, the join points in the AspectJ join point model are therefore related to the different modules available in Java, i.e. methods and classes. Join points in AspectJ are well-defined events in the execution of a Java program, like a method call, method execution, constructor call, constructor execution, field reference and field set [4]. Join points have a *signature*, for example in case of a method call join point, the signature is the method's name and all the other static information the programmer was required to type in when defining the method.

Pointcut language

AspectJ defines a *pointcut language* as a declarative means to describe join points: a pointcut is a predicate over events from the program execution. The AspectJ pointcut language consists of constructs to define primitive pointcuts and logical connectives that can be used to compose pointcuts. The primitive pointcuts in AspectJ are any of the following [39]:

- *Kinded pointcuts* are pointcut designators based on the kind of a join point (`call`, `execution`, `get`, `set`, `initialization` , ...) [5].
- *Context-exposure pointcuts* expose part of the execution context at their join points: these values are bound to a name, that can be used throughout the rest of a pointcut definition. (`this`, `args`, `target`)
- *Control flow-based pointcuts* to capture all join points in the control flow of another join point (`cflow`, `cflowbelow`).

[4] For a detailed reference, check the language reference [39]

[5] The term "kinded poincuts" is adopted in AspectJ 5 to refer to the primitive pointcuts previously known as method-related pointcuts, field-related pointcuts and object creation-related pointcuts (see [40])

- *Expression-based pointcuts* is a predicate on join point context (`if`).

Remember that join points have a signature and the primitive pointcut's argument is a *signature pattern*: it is possible to leave part of the signature pattern variable by using wildcards, allowing pointcuts to range over multiple join points.

Some examples primitive pointcuts are the following:

- `call ( void Account.credit(Client, int) )` matches any join point that represents a call to the method `void Account.credit(Client, int)`.
- `call ( Account.new(..) )` matches any join point that represents a call to any constructor of the class `Account`.
- `get ( int Account.* )` matches any join point that represents reading a slot of type `int` defined in the class `Account`.
- `cflow (call(void Account.deposit(Client, int)))` matches any join point in the control flow of a call to the method `void Account.deposit (Client, int)`. This includes the call itself.
- `if(5 == 5)` matches any join point, because the condition `5 == 5` is always true.

Pointcuts are composed using connectives that define a certain relation between pointcuts. Following logic connectives are available to compose pointcuts:

`&&` logical "and" (binary connective): e.g. `pointcutC = pointcutA && pointcutB` implies that `pointcutC`, applied to a join point, evaluates to true when both `pointcutA` and `pointcutB`, applied to the join point evaluate to true.

`||` logical "or" (binary connective): e.g. `pointcutC = pointcutA || pointcutB` implies that `pointcutC`, applied to a join point, evaluates to true when at least one of the two pointcuts `pointcutA` and `pointcutB` evaluates to true.

`!` logical "not" (unary connective): e.g. `pointcutC = !pointCutA` implies that `pointcutC`, applied to a join point, evaluates to true when `pointcutA`, applied to the join point, evaluates to false.

Take for example the following pointcut definition:

```
pointcut callWithdraw(int price, Article art) : (
    execution(void Account.withdraw(int)) &&
    target(account) &&
    args(amount)
    && if(amount > 100)
}
```

A pointcut definition in AspectJ looks like a method definition: the name `callWithdraw` before the () names the pointcut and the code between the () declares a type for each variable used in the pointcut, defined between the {}.

The pointcut is a composition of four pointcuts, all connected by the "and" connective `&&`: so the entire pointcut is true for a given join point, if each of the four sub-pointcuts is true for that join point. The first sub-pointcut, namely `execution(void withdraw(int))` matches a join point, when that join point represents the execution of a method named `withdraw`, defined in the class `Account`. The second pointcut and the third pointcut are so-called context-exposure pointcuts (which always evaluate to true): they bind the variables `account` and `integer` to the object on which the method `withdraw` is being invoked and the argument of the method `withdraw` respectively. The last pointcut, `if(amount > 100)`, is an expression-based pointcut that evaluates to true for a join point if the value bound (by context-exposure predicates) to `amount` is larger then 100. So this pointcut matches the execution of the piece of code (given `a = new Account(1000)`): `a.withdraw(200)`, but not `a.withdraw(50)` or `a.deposit(200)`.

Introduction

In AspectJ it is possible to add methods and data slots to a class from within an aspect definition; This is done by defining the method/data slot in an Aspect, binding the method to the class by using a scope operator to define the method (c.f. c++) (e.g. introduction of method `log` in class `Account` in Figure 2.4).

Advice language: connecting advice code and pointcuts.

The advice language consists of constructs such as `before`, `after` and `around` and `proceed` to connect advice code with a pointcut. Depending on the construct used to connect the advice code and the pointcut, the advice code is executed before or after an event for which the pointcut (= predicate) is true in the execution of a program in case of a before or after advice respectively. The around connector allows the programmer to replace the event completely by the execution of the advice code, but note that the event can be called from within the advice, using the proceed construct.

For example, a before advice connected to the pointcut `callWithdraw` defined earlier, might look like this[6]:

```
before() : (callWithdraw(int, Article)) {
  String msg = "Balance = ";
Account acc = (Account) thisJoinPoint.getTarget();
  msg += acc.getBalance();
System.out.println(msg);
}
```

If we execute the piece of code `a.withdraw(200)`, given `a = new Account(1000)`, a message "Balance = 1000" is printed to the screen. On the other hand, if we define an after advice like this:

```
after() : (callWithdraw(int, Article)) {
  String msg = "Balance = ";
```

[6]The keyword `thisJoinPoint` can be used within the advice code to refer to the join point for which the pointcut (connected to the advice code) matches: this way, the arguments, target etc. can be retrieved from the join point.

```
Account acc = (Account) thisJoinPoint.getTarget();
  msg += acc.getBalance();
System.out.println(msg);
}
```

The message "Balance = 800" is printed to the screen if `a.withdraw(100)`, given `a = new Account(1000)` is executed. Instead, if we define an around advice like:

```
around() : (callWithdraw(int, Article)) {
  String msg = "Balance = ";
Account acc = (Account) thisJoinPoint.getTarget();
acc.setBalance(acc.getBalance() - (acc.getBalance() / 100));
proceed();
  msg += acc.getBalance();
System.out.println(msg);
}
```

Then, if we execute `a.withdraw(200)`, given `a = new Account(1000)`, the message "Balance = 790" is printed to the screen.

Example: logging sensitive operations using AspectJ

In Figure 2.4 we show how the logging concern for the banking application from section 2.2.2 can be implemented using AspectJ. In Figure 2.4, we see the definition of an aspect `LoggingAspect`: this module contains the definition of two advices and shows the introduction of the method `log` in the class `Account`. The advices are both after advices and they make sure a message is logged when the methods `credit` or `deposit` is called. The class `Account` is now freed from any code that implements the logging concern (Figure 2.3): hence full separation of concerns is achieved as the logging concern is now implemented in one separate module, namely the aspect `LoggingAspect`.

```
public class Account
{
  private int balance;

  public void withdraw(int money) {
    ;; sensitive operation
    balance = (balance - money);
  }

  public void credit(int money) {
    ;; sensitive operation
    balance = (balance + money);
  }

  public void view() {
    system.out.println(balance);
  }
}
```

Figure 2.3: Account class from the banking application in Figure 2.1 freed from the logging code.

```
public aspect LoggingAspect {

    public void Account.log(String message) {
        Date now = new Date();
        message += now.toString();
        System.out.println(message);
    }

    after(Client client) : execution(void Account.credit(Client, int)) && args(
    client, ...) {
        String msg = "Credit called by ";
        msg += client.getClientID();
        log(msg);
    }

    after(Client client) : execution(void Account.withdraw(Client, int)) && args(
    client, ...) {
        String msg = "Credit called by ";
        msg += client.getClientID();
        log(msg);
    }
}
```

Figure 2.4: AspectJ modularising logging concern for banking application in Figure 2.1; Note that the class `Account` is now freed from all logging code (Figure 2.3).

2.3.4 Case study: CARMA

CARMA [17] is an aspect language for Smalltalk, inspired by AspectJ, but based on logic meta programming, meaning that in CARMA, crosscuts are expressed in terms of logic programs. The idea behind CARMA is to use a logic meta language, namely SOUL, extended with a libray of pointcut predicates for the individual join points in CARMA's join point model, as pointcut language.

SOUL [28] is based on Prolog, though with some variations on syntax [7]. For example the rule in Figure 2.5 is a valid rule definition. Note line nr. 1, this is not an ordinary predicate call; The code between the `[]` is regular Smalltalk code: such a block of code is evaluated at the Smalltalk level and should return a boolean value. This mechanism is known as *escaping to the base level* [25]. In addition to full-fledged SOUL, CARMA adds a library of predicates over join points.

```
member(?H, <?H | ?T> ) if [true].
member(?X, <?H | ?T>) if member(?X, ?T).
```

Figure 2.5: SOUL: definition of `class` predicate.

The join points in CARMA are based on the key events in the execution of a Smalltalk program, being message sends/receptions and assignments/references: the join points and primitive pointcuts are described in Table 2.1. The idea is to compose these primitives into more complex pointcuts, that can be abstracted by rule definitions. Hence it is possible to build a library of specialized join

[7] Instead of the symbol `:-`, the keyword `if` is used to separate the head from the body of a rule; The symbol for conjunction is also `,` and lists are surrounded by `<>` instead of `[]`. Queries are simply rules with an empty head.

point predicates that extend the primitive join point predicates: CARMA is therefore called an *open weaver* [19].

`Message reception`	`reception(?jp, ?selector, ?arguments)`
`Message send`	`send(?jp, ?selector, ?arguments)`
`Assignment`	`assignment(?jp, ?varName, ?oldValue, ?newValue)`
`Reference`	`reference(?jp, ?varName, ?value)`

Table 2.1: CARMA primitive pointcuts.

For example, the logging aspect for the banking application in Figure 2.1 looks as in Figure 2.6 in CARMA. The advice definition consists of two parts: the part before the `do` keyword specifies the pointcut for the join point and the part after the `do` keyword is the advice code. Note the `member(?msg, <[#withdraw], [#deposit]>` pattern: the predicate `member` predicate is defined as a plain rule (Figure 2.5).

```
after ?jp matching
  reception(?jp, ?msg, <?client, ?int>),
  member(?msg, <[#withdraw], [#deposit]>)
do
  Transcript show: (?msg: asString), ' called by ', (?client: getClientID)
```

Figure 2.6: CARMA advice: logging sensitive operations for banking application in Figure 2.1.

The advantages of using a logic meta language to express crosscuts are versatile. First of all, the unification mechanism from logic programming is included in the pointcut language as wildcard mechanism; Second, it is possible to reuse crosscut definitions when they are defined by means of a rule; And finally, from meta programming, the languages gets inspection and reflection functionality to reason about a program's (static) structure.

2.4 Expressive Pointcut Languages

Early AOP languages focused on providing support for implementing crosscuts that depend on a single join point and this was reflected in their pointcut languages. With the introduction of control-flow based pointcut descriptors, it was recognized that some crosscuts depend on a sequence of join points in the control flow of another join point (e.g. "log all functions called by function A"). More recently, it was advocated that there exists a great deal of crosscuts that depend on multiple join points, that are not necessarily related by control-flow. We next discuss three AOP approaches that support such aspects, but first we illustrate how to implement these aspects in AspectJ.

2.4.1 Context-aware crosscuts in AspectJ

In this section we discuss an e-commerce application and the difficulties that arise in AspectJ when one tries to implement some example aspects, that depend on (past) program state at (past) join points.

Consider a simple e-commerce application. A shop has customers and sells articles. Customers have an account and have to login to put shop articles in their basket and to checkout their basket. The UML diagram is shown in Figure 2.7.

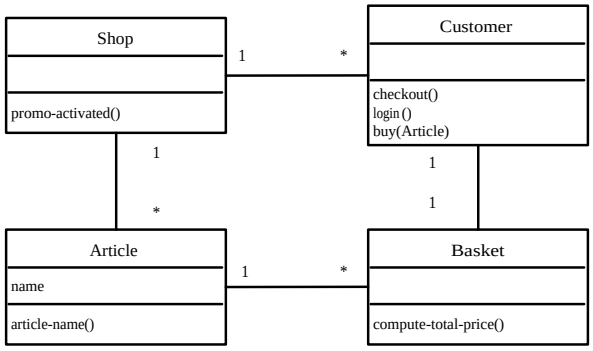

Figure 2.7: E-commerce application

In order to attract customers to the shop, the shop occasionally engages in promotional marketing campaigns. Such a promotional campaign has two effects on the shop, which can be implemented as aspects: the advertising aspect adds banners to the shop's pages to advertise the promotion, and the discounting aspect gives customers a discount on articles when they check out. The shop application can automatically engage in promotions based on certain conditions. There can be several variations of conditions that activate a promotion:

- The current time is in a pre-set interval (e.g. before Christmas, "happy hour", ...).
- There is a stock overflow for a particular item.
- A competing website does a promotion.

The discount aspect affects the computation of the price of the basket when the customer checks out. Whether a discount is given depends on the activation of a promotion, but again, there can be several variations:

- The promotion is active when the customer checks out.
- The promotion is active when the customer logs in.
- The promotion is active when an article is added to the shopping basket.

To illustrate the program's desired behavior, consider a sample program run depicted in Figure 2.8. There is a time line depicted for two users: we see that there is temporarily one promotional context active, namely `seasonal-promo`. On the timeline, we see that user 1 logs in when the context `seasonal-promo` is active: at that time, the website is popping up banners to lure customers to login (e.g. "Login now and get a discount on checkout!"). The idea is that when user 1 checks out at a time the `seasonal-promo` context is no longer active, user 1 still gets her discount related to the `seasonal-promo` context. User 2 however

does not get this discount, as she logged in when the promotional context was not active, but no harm done: the banners promising a discount were no longer being displayed when she logged in anyway.

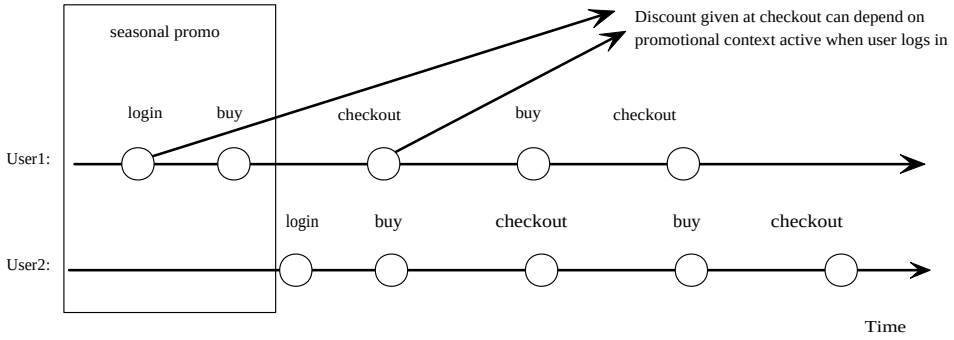

Figure 2.8: E-commerce application program run

In the example, there are two aspects that depend on the same promotional context of the shop. First of all, appropriate banners must be displayed on the website when a promotion is active and second, the promotion context affects the discounts a particular user receives. So when considering an implementation, we need to separate the promotion context definition so that we can use it for implementing both the banners and the discount aspects. We next discuss how these aspects can be implemented in AspectJ.

The first aspect we implement, gives a discount at check out time for a user, if the shop was in a promotional state when the customer logged in. In order to do this in AspectJ, we need to create a new aspect:

```
public aspect DiscountAtCheckout {
    ...
}
```

This module encapsulates the different advices. We note that in order to give a discount at check out, we need to advice the method `computeTotalPrice`, as this method is responsible for calculating the bill – by accumulating the price of all articles in the user's shopping basket – and giving the discount, which results in subtracting an amount from this total (Figure 2.9).

```
int around(): (callCheckout(User)){
    int total = proceed();
    total -= (int) (total * 0.10);
    return total;
}
```

Figure 2.9: AspectJ advice definition.

However, the discount is only given if the user logged in when the discount promotion was active. So, the pointcut looks as in Figure 2.10. The pointcut is a composition of four pointcuts: the one on line nr. 2 matches all join points in the control flow of the execution of the method "checkout"; Line nr. 3 shows

a context-exposure pointcut, binding the variable `usr` to the target object on which a method call, represented by such a join point, is called; Line nr. 4 is an expression-based pointcut to check whether there was a promotion at login time for the object bound to `usr` which uses the method `promotionsActiveAtLoginP` and Line nr. 5 matches a join point that represents the execution of a method named `computeTotalPrice`. This pointcut, named `callCheckout` evaluates to true for a join point if all of the sub-pointcuts evaluate to true (because && is used to connect all of them). Except for the implementation of the method `promotionsActiveAtLoginP` (see Figure 2.11) used in the expression-based pointcut, we have described the complete definition of the pointcut.

```
1. pointcut callCheckout( User usr): (
2.    cflow(execution(void User.checkout())) &&
3.    target(usr) &&
4.    if(promotionsActiveAtLoginP(usr)) &&
5.       execution(int User.computeTotalPrice())
6. );
```

Figure 2.10: AspectJ pointcut definition.

The method `getPromosActiveAtLogin` returns true if the `User` object, on which it is invoked, logged in when a promotion was active. However, this information can not simply be computed when the method `getPromosActiveAtLogin` is called (when the pointcut in Figure 2.10 is applied to a join point) because the shop might not be in a promotional state anymore[8]. So we need to keep a state variable for each user object that stores this information when a user logs in: AspectJ's introduction mechanism allows us to do this (Figure 2.11), by adding a new instance slot to the class `User` (line nr. 1) and method (lines nr. 3-5). Of course, now we need a means to make sure the instance slot `promosActiveAtLogin` is set.

```
1. private boolean User.promosActiveAtLogin = false;
2.
3. public boolean User.getPromosActiveAtLogin(){
4.        return promosActiveAtLogin;
5. }
6. private static boolean promotionsActiveAtLoginP(User usr){
7.        return usr.getPromosActiveAtLogin();
8. }
```

Figure 2.11: AspectJ introduction of an instance slot and an instance method in the class `User`.

We need to implement another advice that computes the value for the instance slot `promosActiveAtLogin` when a user logs in (ergo when the method `login` is invoked on a `User` object). The pointcut is depicted in Figure 2.12 and the advice in Figure 2.13.

The AspectJ implementation of the aspect works fine, but is not very intuïtive. We need to manage a state for `User` objects to track if a user logs in when promotions are active, to check when the check out event happens in order to decide to give a discount or not. In fact, the implementation of the (conceptually one)

[8]Remember that the promotional state of a shop object is represented by a slot `promoActivated`, containing a boolean value, in the class `shop`, which is flipped when the promotions are turned on/off.

```
pointcut callLogin(User usr): (
execution(void User.login()) &&
target(usr) &&
    if(promotionsActiveP(usr))
);
```

Figure 2.12: AspectJ pointcut to match join points representing invocations of the method `login` when promotions are active.

advice "trigger a discount at check out if user logged in when promotions were active" is scattered over several advices (Figure 2.11, 2.13). The problem is that the conceptual advice depends on multiple (past) join points in the execution history of the program – not related by control flow – and this relation cannot be expressed in AspectJ's pointcut language. This example is not a lonely exception.

```
after() : (callLogin(User)) {
User usr = (User) thisJoinPoint.getTarget();
usr.promosActiveAtLogin = true;
}
```

Figure 2.13: AspectJ advice on pointcut `callLogin` in Figure 2.12.

A second aspect we will illustrate, implements the fact that a discount is given on an article, whenever there was a stock-overflow for that type of article, when the user added it to her basket. Again we need to create a new aspect:

```
public aspect DiscountAtCheckout {
...
}
```

Adding an article to a basket is implemented by the method `buy`. So assigning a discount to an article happens when this method is executed (Fgure 2.14). Again, the pointcut is a composition of pointcuts, all connected by &&: the pointcut evaluates to true for a join point if all these sub-pointcuts evaluate to true. Line nr. 2 captures the execution of the method `buy`; Line nr. 3 and nr. 4 bind the target object and arguments of the join point to the variables `usr` and `art` respectively; Line nr. 5 is a context-exposure pointcut that calls the method `stockOverflow`, which expresses that a discount is only assigned if there is a stock-overflow for that particular type of item.

```
1. pointcut callBuy(User usr, Article art) : (
2. execution(void User.buy(Article)) &&
3. target(usr) &&
4. args(art) &&
5. if(stockOverflow(usr, art, 2))
6. );
```

Figure 2.14: AspectJ pointcut.

The method `stockOverflow` is implemented as in Figure 2.15. We need to introduce an instance method `stockOverflowShopP` in the class `shop`: it simply counts the articles of a particular type and compares this number to a given threshold for "stock-overflow".

```
public boolean Shop.stockOverflowShopP(Article art, int nr){
        LinkedList arts = this.getArticles();
        int countedSoFar = 0;
        for(int i = 0; i < arts.size(); i++){
            Article art2 = (Article) arts.get(i);
            if(art2.getName().equals(art.getName())){
                countedSoFar += 1;
            }
        }
        return (countedSoFar > nr);
  }
```

Figure 2.15: AspectJ introduction of a method in class `Shop`.

The advice (Figure 2.16) simply stores the fact that an article is assigned a discount when the pointcut of Figure 2.14 matches for a join point. Note that we need to manage a dictionary to store the discounts assigned to the articles (line nr. 10), this dictionary can then be queried to compute the discounted price of an article when it is billed to the user on check out (Figure 2.17). Basically, we get "complex" code for just the same reasons as noted for the implementation of the previous aspect.

```
1.  before() : (callBuy(User, Article)) {
2.        Object args [] = thisJoinPoint.getArgs();
3.        Article art = (Article)args[0];
4.        User usr = (User) thisJoinPoint.getTarget();
5.        // store discount
6.        Double d = new Double(0.10);
7.        collectDiscounts.put(art, d);
8.  }
9.
10. private Dictionary collectDiscounts = new Hashtable();
```

Figure 2.16: AspectJ before advice.

Implementing crosscutting concerns that depend on multiple join points in the execution of a program in AspectJ, is not straightforward, because AspectJ's pointcut language does not provide a sufficiently expressive mechanism to express the ordering relation between two join points [9]: if we try to implement such aspects, this leads to complicated code, involving the manipulation of state variables.

```
  int around() : (callgetPrice()){
        Article art = (Article) thisJoinPoint.getTarget();
        double newPrice = proceed();
        Double percentage = (Double) collectDiscounts.get(art);
        if (percentage != null){
            double prcentage = percentage.doubleValue();
            newPrice -= (newPrice * prcentage);
            collectDiscounts.remove(art);
        }
        return (int) newPrice;
  }
```

Figure 2.17: AspectJ around advice.

[9] The `cflow` pointcut allows one to match all join points in the control flow of another join point, but we need more!

2.4.2 Event-based Aspects

Douence et. al. [12] [33] formally define event-based aspect oriented programming (EAOP) as a general framework for AOP in which they define aspects in terms of sequences of events emitted during program execution. The EAOP model allows one to express pointcuts that become true if a sequence of events (unrelated by control flow) is matched in the execution history of a program [10].

EAOP-tool

Currently, Douence et. al. [35] have implemented an EAOP tool which is basically an object-oriented framework for writing EAOP applications in Java: with this approach they do not offer a pointcut language, but rely on the programmer to manually generate and match events using built-in methods, which is not a very declarative means for specifying pointcuts.

We illustrate how this framework can be used to implement a discount aspect for an e-commerce application depicted in Figure 2.7. The purpose of the discount aspect is to give a discount to a user if it is her first check out ever.

To define a new aspect, we need to create a subclass of the class `Aspect`. The class `Aspect` implements the methods `nextEvent` and `definition`. The method `nextEvent` returns the next event in the execution of a program. The idea is that we use this method within other method definitions to manually define pointcuts. Figure 2.18 shows the method `nextCheckout` which defines a pointcut that matches calls of the method "checkout".

```
Event nextCheckout(Customer c) {
  Event e = nextCheckout();
  while (((MethodCall)e).receiver != c) {
    e = nextCheckout();
  }
  this.isCrosscutting = true;
  return e;
}

Event nextCheckout() {
  boolean ok = false;
  Event e = null;
  while (!ok) {
    e = nextEvent();
    ok = (e instanceof MethodCall) &&
         ((MethodCall)e).method.getName().equals("checkout");
  }
  return e;
}
```

Figure 2.18: EAOP pointcut definition.

The purpose of the method `definition` is to define an advice for the aspect. To implement our discount aspect, we need to override this method, as in Figure 2.19. Note that we need to manually match the pointcut by means of the method `nextCheckout` and that we need to manually attach a discount aspect per user instance (cf. `insert`).

From this relatively simple example it becomes clear that without the support of a pointcut language writing event-based aspects is rather cumbersome:

[10] We mean the whole execution history of a program here, not just methods that were called previously in the current control flow.

```
public void definition() {
  Customer c = newCustomer();
  insert(new Discount());
  while (true) {
    Event e = nextCheckout(c);
    if firstCheckout(e){
      System.out.print("discount: "); /* advice */
    }
  }
}
```

Figure 2.19: EAOP advice.

the programmer needs to manipulate the events generated during program execution manually. Currently, there is one other AOP system related to the EAOP model which has a more declarative means to define aspects which allows writing aspects that depend on multiple events in the execution history, namely JasCo.

2.4.3 JAsCo Stateful Aspects

JAsCo [42] is an AOP system for Java with some language support to write down stateful aspects. The idea is that aspects are triggered by sequences of join points and they evolve according to the join points they match. For example, an aspect responsible for logging sensitive operations in a banking application, must wait for the log file to be opened before it can start writing logging information to it: these aspects are called *stateful aspects* because a state is needed to represent their evolution (logging on/off if file opened/closed). Note that a change of state is triggered by an event in the execution of a program. The problem with current AOP languages is that they do not offer direct support to write down stateful aspects, but rather depend on the programmer to manage this "aspect state" by means of variables manually. JAsCo however, does offer some linguistic support for writing down stateful aspects.

In JAsCo, stateful aspects [7] are implemented by means of a state machine to track the state an aspect is in: JAsCo offers constructs to describe the states and transitions of this state machine declaratively so that the programmer does not have to generate events at join points and feed these to a state machine manually (as is the case in section 2.4.2 with method `nextEvent`). We illustrate the JAsCo language by means of an example.

Consider for example a banking application where sensitive operations need to be logged (as in section 2.2.2), but the logging should only be active when a user is logged in. In JAsCo this boils down to the aspect depicted in Figure 2.20. The idea is that the program execution is mapped onto the transitions in a state machine: the method `ProtocolLogger` implements this by enumerating the different possible transitions and by specifying which possible transitions can be taken next, when a certain join point [11] is encountered, following the pattern:

```
"transition name" : "JAsCo pointcut" > "following transitions";
```

[11] JAsCo primitive pointcuts include call(method), execution(method), cflow(method), withincode(method), target(MyClass) and connectives ||, !, && [41].

For every transition a before-, after- or around-advice can be defined, allowing one to insert behavior at a state transition, following the pattern:

```
before|after|around "transition name" {....}
```

```
class ProtocolLogger extends Logger {

 hook StatefulProtocolLogger {

 StatefulProtocolLogger(methodLogin(..args),methodDeposit(..args),methodWithdraw(..args), methodLogout(...args)) {
   CalledLoginTrans :
     execution(methodLogin) > CalledWithdrawTrans || CalledDepositTrans || CalledLogoutTrans ;
   CalledWithdrawTrans :
     execution(methodWithdraw) > CalledWithdrawTrans || CalledDepositTrans || CalledLogoutTrans ;
   CalledDepositTrans :
     execution(methodDeposit) > CalledWithdrawTrans || CalledDepositTrans || CalledLogoutTrans ;
   CalledLogoutTrans :
     execution(methodLogout) > CalledLoginTrans;
 }

 after CalledLoginTrans() {
   System.out.println("Started logging for client ");
   System.out.println(thisJoinPoint.getArgumentsArray()[0].getClientID());
 }

 before CalledWithdrawTrans() {
   System.out.println("Logging withdraw for ");
   System.out.println(thisJoinPoint.getCalledObject().getClientID());
   System.out.println("thisJoinPoint.getSignature()");

 }

 before CalledDepositTrans() {
   System.out.println("Logging deposit for ");
   System.out.println(thisJoinPoint.getCalledObject().getClientID());
   System.out.println("thisJoinPoint.getSignature()");

 }

 before CalledLogoutTrans() {
   System.out.println("Stop logging for ");
   System.out.println(thisJoinPoint.getArgumentsArray()[0].getClientID());
 }
 }
 }
```

Figure 2.20: JAsCo AspectBean

It is also possible to put conditions on the transitions (e.g. Figure 2.21).

```
isApplicable CalledDepositTrans() {
  return thisJoinPoint.loggingEnabled();
}
```

Figure 2.21: JAsCo condition on transition

In JAsCo, aspects need to be deployed using a connector (Figure 2.22): this makes sure a state machine is created for an aspect. Note that the programmer must decide at what time a state machine must be created (keyword perobject, perthread, etc).

The main benefit of the JAsCo approach is that one needs not concern oneself with events generated at runtime nor with feeding them to a state machine or managing state variables to reflect the state the aspect is in as was the case with

```
static connector TimingConnector {
  perthread ProtocolLogger.StatefulProtocolLogger logger =
    new ProtocolLogger.StatefulProtocolLogger(
    void Bank.login(Client),
    void Client.withdraw(AccountNr, integer),
    void Client.deposit(AccountNr, integer),
    void Bank.logout(Client)
  );
}
```

Figure 2.22: JAsCo connector

EAOP tool (section 2.4.2). There are however some problems with the JAsCo approach:

1. It is difficult to write down a stateful aspect in terms of transitions without a good clear picture of the state machine in ones mind (Figure 2.23) [12]
2. There is no declarative way to pass context information from one join point to another to check conditions on the second join point: one needs to create a variable and assign it when a transition is entered.
3. One needs to decide manually when to create create an instance of an aspect.

Therefore we believe there is still room for improvement in JAsCo.

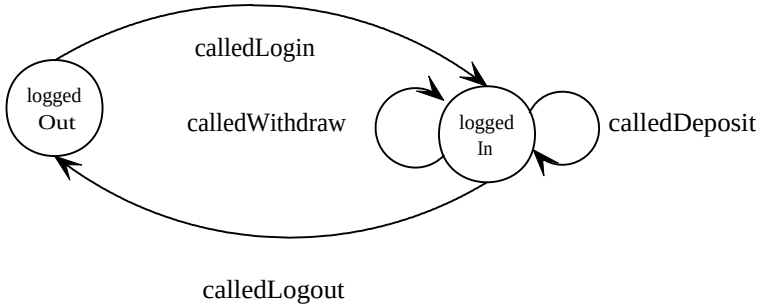

Figure 2.23: State machine for JAsCo aspect in Figure 2.20.

2.4.4 Context-aware aspects

The notion of context-aware aspects has been introduced by Tanter et. al. [36]: Related to the idea of context-aware applications, context-aware aspects are aspects whose behavior is context-dependent. In that paper, the authors address the problem that current AOP languages incorporate a too limited notion of context when considering context-aware applications (e.g. context is only information on the join point and not the state of a program) and that therefore the

[12] We found it particularly difficult to write the aspect without first drawing the state machine.

definition of context needs to be extended, for example to include the ability to refer to *past* contexts. The use of a specialized pointcut language is proposed, but only back-end technology based on Reflex is discussed.

Reflex [37] is an open reflexive extension to Java that makes it possible to do both structural and behavioral modifications of programs. The main concepts in Reflex for doing such modifications are the *behavioral link* and the *structural link*: only the behavioral link is discussed in [36] as a means to implement context-aware aspects. A behavioral link is responsible for invoking messages on a *metaobject* at occurrences of operations specified by a *hookset*. A hookset, similar to a pointcut in AspectJ, is a means to describe where an aspect will apply (= a set of operations such as method calls, method executions, etc.): the class `Hookset` can be used to create new hooksets; The constructor of the class `Hookset` takes three *selection conditions* (a sort of predefined predicates under the form of classes) as arguments which describe the type of operations, the class in which the operations occur and constraints on the operations themselves: note that only lexical crosscuts can be expressed by hooksets. For example, Figure 2.24 defines a hookset in Reflex: it describes all method (call)s (line nr. 2) where the method is defined in the class `Bank` (line nr. 3).

```
// Reflex hookset
1. Hookset logHookset = new Hookset(
2.                          MsgReceive.class,
3.                          new NameCS(Bank, false),
4.                          new AnyOS()));

// Equivalent AspectJ pointcut
execution(* Bank.*(..))
```

Figure 2.24: Reflex hookset definition: captures all method calls, where the method is defined in the class `Bank`

Creating a behavioral link is done by means of the constructor `Links.createBlink` which takes a hookset and a metaobject as arguments and by setting the different properties of the link (by methods like `setCall` defined in `BLink`). For example, in Figure 2.25 we define a behavioral link, named `log` which is responsible for sending a message `log`, before a method of the class `Bank` (described by hookset `hookset` in Figure 2.24) is called, to a metaobject that is the instance of the Java class `Logger`. Notice line nr. 5, this shows an important feature of Reflex: the method `addActivation` can be used to add an *activation condition* which restricts the set of intercepted operations to those that satisfy a *condition* at runtime. A condition is implemented by implementing the interface `Active`, which exposes one method namely, `evaluate` that takes as argument the object on which an operation (intercepted by a hookset) is called. In [36] Reflex is extended with support for defining contexts and context-specific link activation conditions.

Defining a context in Reflex is done by subclassing the class `Context` and overriding the method `getState`. This method returns `null` if a context is not currently active or returns an instance of the class `ContextState`, indicating the program is in a context. Calling the method `getState` is refered to as *snapshotting*. An object `ContextState` contains context information (e.g. the discount rate in a promotional context). The Reflex implementation makes sure that the `ContextState` object's context information, is the same as when the snapshot

```
1. BLink log = Links.createBLink(logHookset, new Logger());
// class Logger contains advice code under the form of methods
2. log.setCall(Logger.class, log);
3. log.setControl(Control.BEFORE); // before advice
4. log.setScope(Scope.GLOBAL); // singleton metaobject
5. log.addActivation(new Condition()); // adding a dynamic condition
```

Figure 2.25: Reflex behavioral link for hookset in Figure 2.24

was requested (i.e. stateful). For example in Figure 2.26 defines a `ChargeCtx` for our banking application, defining the rate for performing a transaction. A object `ChargeState` contains the rate at the moment the snapshot was created, no matter when the snapshot is accessed.

```
class ChargeCtx extends Context {
    CFlow cf = CFlowFactory.get(new Hookset(MsgReceive.class,
        new NameCS(Bank, false), new AnyOS()));
    double rate; // with setter
    ContextState getState() {
    if(!cf.in()) return null;
    return new ChargeState(rate);
}
class ChargeState
    extends ContextState {
    double rate;
    double getRate() { return rate; }
}}
```

Figure 2.26: Reflex context definition

The idea is now that contexts are used in link activation conditions. In [36] Reflex is extended with the classes `CtxActive` and `SnapshotCtxActive` which implement the interface `Active` and these classes need to be subclassed in order to obtain a context-dependent link activation condition. The class `CtxActive` class contains an instance variable, `itsContext`, referring to the activation condition's context, the method `evaluate` returns true if the associated context is/was active. To determine if a context is/was active, the method `getCtxState` needs to be overridden. For example the context `CurrentlyInCtx` defines an activation condition that can be used in a link to capture operators only if a context is currently active:

```
class CurrentlyInCtx extends CtxActive {
ContextState getCtxState(Object o){ return itsContext.getState(); }
}
```

The abstract class `SnapshotCtxActive` extends the class `CtxActive` and provides the necessary support for defining activation conditions that depend on past context snapshots by implementing the methods:

- `void snapshot(Hookset hs, Parameter p)`: stores a snapshot of the context in `p` when an operation matches the given hookset `hs`
- `Snapshot getSnapshot(Object o)`: returns the snapshot taken for an object `o`

For example, we can implement a `Charge` context for a banking application – which represents the current rate that is charged for handling a transaction – as in Figure 2.27[13]. The idea is that the rate that is charged for a transaction

[13]Example `CreatedInCtx` adapted from [36].

changes over time (e.g. when it's Christmas, the rate is lower than usually). The class `CreatedInCtx` is responsible for making a context snapshot for each created object. The advice code (implemented by method `amount` in a class `Charge`) is only executed when the chargeCtx is active when an operator matches the specified hookset (because `inCharge` is added as activation condition). The (built-in) method `getCtxParam` is used to fetch the rate that was set when the object, on which an operator is used, was created.

```
class CreatedInCtx extends SnapshotCtxActive {
    CreatedInCtx(Context c, BLink l){
    super(c);
    ClassSelector cs = l.getClassSelector();
    annotate(cs);
    snapshot(new Hookset(Creation.class, cs, new AnyOS()), Parameter.THIS);
} }

BLink charge = Links.createBLink(chargeHS, new Charge());
Context chargeCtx = new ChargeCtx();
CtxActive inCharge = new CreatedInCtx(chargeCtx, charge);
charge.addActivation(inCharge);
charge.setCall(Charge.class, amount, inCharge.getCtxParam(rate));
```

Figure 2.27: Reflex create context definition

The nice thing about Reflex is that it actually does offer support for implementing context-aware aspects. However, this support is offered under the form of a framework, rather than a declarative pointcut language. We believe that a dedicated pointcut language can greatly improve the expressiveness of such a system.

Problem statement

Crosscuts in terms of sequences of join points are a recognized problem. Current AOP systems that allow one to express them in terms of aspects, such as EAOP tool, JasCo and Reflex do not offer pointcut languages for defining them, but rather rely on the programmer to manipulate the events generated during program execution themselves.

We believe however that a dedicated pointcut language can add to the expressiveness of these AOP systems. Furthermore, we believe that it can in fact be done without extending the model of AOP languages we defined in section 2.3.1. We believe it suffices to introduce connectives into an existing pointcut language which define a temporal relation between the pointcuts they connect. In the next chapter we discuss temporal logic as a basis for developing such connectives.

2.5 Summary and Conclusion

In this chapter we introduced aspect-oriented programming (AOP). AOP is a new programming paradigm in which a strict separation of concerns is pursued, which is not possible to achieve using a traditional object-oriented, functional or procedural language. AOP is currently achieved through the extension of a base language with language constructs to define modules for implementing crosscutting concerns: this language extension is called an aspect language.

Aspect languages are based on a join point model which consists of join points, a means of describing join points and a means of affecting behavior at a join point, which in most aspect languages is done by means of a pointcut language and an advice language. Pointcut languages allow one to describe sets of join points declaratively. Pointcuts are connected to pieces of code that implement part of a crosscutting concern using a before, after or around + proceed connector.

An example aspect language we introduced is AspectJ, which is an aspect language for Java. AspectJ allows one to write pointcuts that match single events in the execution of a program and pointcuts that match events in the control flow of another event. There is however a great deal of crosscuts that depend on sequences of events in the execution history of a program. This lead to the introduction of EAOP as a new model for AOP where crosscuts depend on sequences of events and context- aware aspects in which context-exposure of past events is possible. Current implementations of these AOP models do not offer pointcut languages to deal with temporal relations. We believe however that a dedicated pointcut language can greatly improve the expressiveness of these systems and in the next section we investigate temporal logic as a basis for designing such a pointcut language.

Chapter 3

Logic Meta Programming

In this chapter we discuss the concepts behind *logic programming* by means of the prototypical logic programming language called Prolog. We next define *logic meta programming* and its relation to logic programming and we present the current (research) applications of logic meta programming. We continue the chapter with the definition of *temporal logic meta programming* and discuss its relation to *temporal logic*. To conclude the chapter we give a summary and we introduce the experiment to evaluate the use of temporal logic meta programming.

3.1 Logic Programming

Before we define Logic Meta Programming, we take a look at logic programming. Logic Programming is a programming paradigm: the idea is that a program describes a logic theory and that a procedure call (or *query*) is nothing but a theorem which needs to be verified for truth using this program [13]. Logic programming is often called *declarative* programming as a program describes *what* a problem is, rather than *how* to solve it. As an example logic programming language, we discuss Prolog, as this is the basis of most logic meta programming applications (including CARMA) and hence of particular interest for this dissertation.

3.1.1 Definite clause logic

Before we delve into Prolog, we introduce definite clause logic, which is the logic after which Prolog was designed. Understanding definite clause logic helps understanding Prolog better. Another reason why we introduce definite clause logic is that this allows us to introduce some important concepts from logic theory that are used throughout the rest of this chapter, by example – for a formal overview of logic theory, we refer to [13] or [32]. Discussing definite clause logic –or any logic language for that matter– is done by covering three topics: syntax, semantics and proof theory. Syntax defines the sort of logic formulas or sentences one can write down in a logic language. Semantics gives meaning to logic formulas, meaning that some formal theory can be used to determine whether a formula is true or false. Finally, proof theory defines how

constant	::	a single word starting with lower case letter
functor	::	a single word starting with lower case letter
predicate	::	a single word starting with lower case letter
variable	::	a single word starting with upper case letter
term	::	<constant> | <variable> | <functor>(term{,term}*)
atom	::	<predicate>(<term>{,<term>}*)
clause	::	<head>: −<body>.
head	::	<atom>
body	::	<atom>{,<atom>}*

Table 3.1: Grammar for definite clause logic.

we can acquire new sentences from assumed sentences by means of pure symbol manipulation [13].

Syntax

The syntax of definite clause logic is defined as follows. A *variable* is a single word, starting with an uppercase letter, whereas a *constant*, a *predicate* and a *functor* are represented by a single word, starting with a lower case letter. A *term* is either a constant, a variable or a functor followed by a number of terms, enclosed between brackets and separated by comma's (e.g. `pirate(lotte, Ship)`). A *ground term* is a term without variables. An *atom* is a predicate, followed by a number of terms (called *arguments*), enclosed between brackets and separated by comma's (e.g. `parent(marie, lotte)`). A *clause* consists of a *head*, followed by the symbol `:-` and a *body*. A head is simply an atom and a body is a sequence of atoms, separated by a comma, ending with a dot (e.g. `parent(Person, lotte) :- mother(Person, lotte).`. A *program* is a set of clauses. The grammar is summarized in Backus-Naur form in Table 3.1.

An example program is depicted in Figure 3.1 [1]. In order to give meaning to this program, we need to interpret the program. Informally, the purpose of the program is to represent the `grandparent` relation; It states which people are parents of which people and consequently which people are grandparents of which people. Formally, the semantics of a program written in definite clause logic, is defined by its *model theory*.

```
1. parent(anna, marie).
2. parent(marie, lotte).
3. grandparent(GrandPerson, Child) :-
4.     parent(GrandParent, Parent),
5.     parent(Parent, Child).
```

Figure 3.1: Prolog: grandparent relation.

Model theory

Remember that the idea behind logic programming is that a program describes a logic theory and that a query is nothing but a theorem (= a formula) that

[1] The notation `parent(anna,marie).` is short for `parent(anna,marie):- true`

needs to be proved given this program. Now that we know the syntax of definite clause logic we can write syntactically correct programs, but we still don't know the semantics of a formula: i.e. is either false or true? In order to determine the semantics of a formula, given a program, we need to cover quite a few concepts.

The *Herbrand universe* of a program is defined as the set of ground terms that can be constructed from the constants and functors present in the program. For example the Herbrand Universe for the program shown in Figure 3.1 is defined as $\{anna, lotte, marie\}$. Note that in general it possible that the Herbrand universe is infinite: e.g. the program `plus(0,0,s(X))` has as Herbrand universe the set $\{0, s(0), s(s(0)), ...\}$.

Next, the *Herbrand base* of a program is the set of ground atoms that can be constructed using the predicates present in the program and the ground terms from the program's Herbrand universe. For example the Herbrand base for the program depicted in Figure 3.1 is the set $\{parent(anna, marie)$, $parent(marie, anna)$, $parent(marie, lotte)$, $parent(lotte, marie)$, $parent(anna, lotte)$, $parent(lotte, anna)$, $grandparent(anna, marie)$, $grandparent(marie, anna)$, $grandparent(marie, lotte)$, $grandparent(lotte, marie)$, $grandparent(anna, lotte)$, $grandparent(lotte, anna)$, $parent(anna, anna)$, $grandparent(anna, anna)$, $parent(marie, marie)$, $grandparent(marie, marie)$, $parent(lotte, lotte)$, $grandparent(lotte, lotte)\}$.

The *Herbrand interpretation* for a program is a subset of the Herbrand base of the program, so that each element is assigned the truth value *true*. A *substitution* is a mapping from variables to terms (e.g. {Child→lotte} and {X→Y} are substitutions). Applying a substitution to a clause means that the occurrence of each variable on the left-hand of a substitution is replaced by the term on the right-hand of the same substitution. For example, if we apply the substitution {Parent→marie, Child→lotte, GrandPerson→anna} to the clause featured in Figure 3.1, we get the clause:

```
grandparent(anna, lotte) :-
    parent(anna, marie),
    parent(marie, lotte).
```

A *ground clause* is a clause without variables and a *ground instance* of a clause is the clause obtained by applying a substitution that replaces all variables in the clause. A ground clause is true given an interpretation if at least one atom in the body of the clause is false in the interpretation *or* the head of the clause is true in the interpretation. An interpretation is a *model* for a clause, if the interpretation is a model for every ground instance of the clause. For the program depicted in Figure 3.1, we can construct the model {parent(marie,lotte),parent(anna,marie), grandparent(anna, lotte)}. A clause C is a *logical consequence* of a program P ($P \models C$), if every model of P is also a model of C.

So, in theory, in order to resolve a query Q given a program P, we can check $P \models Q$: however, this is a very inefficient method, as the number of models for a program can be quite large, even infinite! Proof theory allows to determine much faster if a query is true or false.

Proof theory

Proof theory is a technique that relies on the idea that if a program P can be rewritten, by some clever symbol manipulation following well-defined rules called *inference rules*, to a query Q, then that query is said to be *proved* ($P \vdash Q$)

and preferably true ($P \models Q$). Proof theory is *sound* if $P \vdash Q \rightarrow P \models Q$. The proof theory for definite clause logic consists of one single rule called *resolution*.

Resolution makes use of a matching process called *unification*. A substitution is called a *unifier* for two atoms, when the substitution assigns values to the variables in the atoms such that both atoms are equal after applying the substitution. For example, the substitution {Parent→marie, Child→lotte } is a unifier for the atoms $parent(Parent, lotte)$ and $parent(marie, Child)$: both patterns are said to *unify*). Note that before one tries to unify two patterns, one must check if a variable does not occur in a term one is trying to substitute the variable for: this is called the *occur check* . E.g. unifying `sails(X, boat_owned_by(X)` and sails(Y, Y) should fail, but without the occur check, the unification loops. Unification without the occur check makes resolution *unsound* [13].

The resolution rule is now as follows. If there is an atom in the body of a clause that unifies with the head of another clause, resulting in a unifier, the two clauses *resolve* to the clause gotten after applying the unifier to a clause where the head of that clause is the same as the head of the first clause and the body of that clause consists of union of the atoms in the bodies of both clauses, except the for the head of the second clause. E.g. applying resolutions to the clauses nr. 1 and 2 resolves in the clause nr. 3 shown in Figure 3.2.

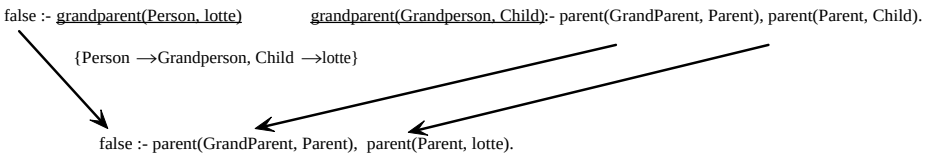

Figure 3.2: Resolution example.

Instead of randomly applying resolution to the clauses in a program in order to prove a query, a proof technique called *proof by refutation* is used. The idea is that we try to prove the opposite of the query and if we stumble upon a contradiction, the query is proved. In order to find a contradiction, we repeatedly apply resolution to the query $false :- query$ and if we come to a point where we get the clause $false :- true$, we have a contradiction, as this clause can never be true. For example, if we try to prove the query $grandparent(Person, lotte)$, given the program depicted in Figure 3.1, we try to prove $false :- grandparent(Person, lotte)$, which leads to a contradiction. The different resolution steps are depicted in the proof tree depicted in Figure 3.3. Note that sometimes, multiple clauses can be chosen to perform a resolution step (e.g. in Figure 3.3 at resolution step 2, we can chose the clause $parent(anna, marie)$ aswell), depending on the clause chosen, the proof theory can even fail, meaning one resolves to a clause –different from $false :- true$– that can not be resolved anymore. Nevertheless, all possibilities must be tried to make sure the clause $false :- true$ cannot be obtained; If this is the case for all possibilities, the query simply fails.

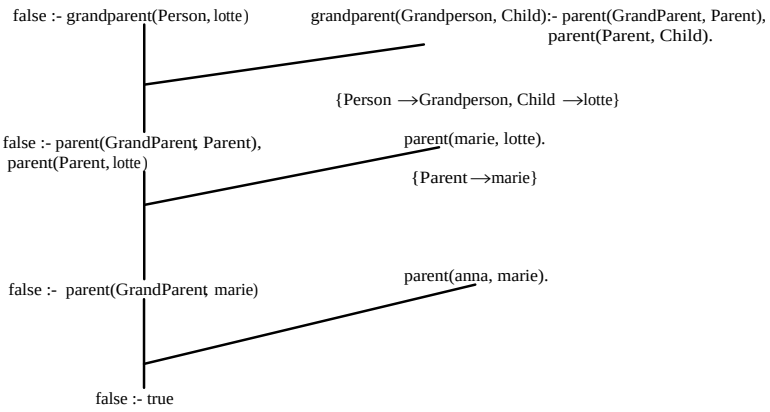

Figure 3.3: Proof tree for the query $grandparent(Person, lotte)$, given the program in Figure 3.1.

3.1.2 Prolog

In principle, Prolog is *almost* the same as definite clause logic. The main difference between Prolog and definite clause logic is Prolog's *resolution strategy* and built-in features (such as keywords `cut`, `is`, etc.). In this section we will focus on explaining the resolution strategy, as it is important to realize –because we are going to implement our own logic language– what the differences between logic and logic programming are.

The syntax and semantics are practically the same as the syntax and semantics for definite clause logic –though clauses are sometimes referred to as *rules* and clauses with an empty head as *facts*. The main difference is that there are a set of keywords –depending on the implementation– built-in such as `cut`, `is` etc. . For a decent overview we refer to the documentation of a particular Prolog distribution (e.g. [46]).

Prolog's proof procedure is based on the proof theory defined for definite clause logic, namely proof by refutation. In order to make that proof theory executable, we need to chose the order in which an atom in a body of a clause is picked out to apply a resolution step and we need to define how the second clause in a resolution step is picked out from a program. Together this is called a *resolution strategy*. Prolog implements the *SLD-resolution strategy*. SLD stands for selection rule, linear resolution and definite clauses. The idea is that the clauses in a program are tried top-down, one by one and the atoms in the body of a clause are resolved from left to right. E.g. resolving the query `?grandparent(Person,lotte)`, given the program in Figure 3.1, results in a search, depicted as a path in Figure 3.4 in a tree of possible resolution steps.

3.2.

The SLD-resolution strategy makes Prolog executable, but also less declarative. The order in which the clauses are added to the program and the order of the atoms in the body of a rule influences whether a query is answered or not. There are even SLD trees with infinite branches [13]. Consequently, we might never reach a success branch in the SLD tree, because we keep going down an

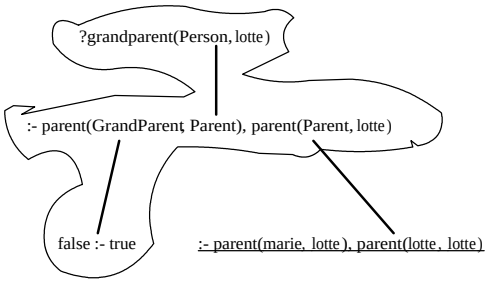

Figure 3.4: Path followed in SLD tree for answering query `?grandparent(Person,lotte)` given the program depicted in Figure 3.1

infinite subtree. This makes Prolog *incomplete*, as there are logical consequences of a program that cannot be found. Secondly, Prolog is *unsound*, as the *occur check* (see Section 3.1.1) is omitted from Prolog's unification algorithm. Hence, Prolog is different from definite clause logic, which is sound and complete.

3.2 Logic Meta Programming

The definition of logic meta programming is as follows:

> Logic Meta Programming is the use of a Logic Programming language at the Meta level to reason about and manipulate programs built in some underlying base language.

The idea behind logic meta programming is to use a logic programming language, called the *meta language*, to write meta programs that state knowledge about programs written in some other language, called *base language*. This knowledge base can then be queried to reason about a program written in the base language.

The goal of logic meta programming research is to examine how logic meta programming can be used to support the software development process in one way or another [6]. Current applications of logic meta programming being investigated can be placed in one of the following categories:

- verification of source code to some higher-level description (e.g. checking if source code matches certain design patterns, coding conventions etc. [28])
- extraction of information from source code (e.g. visualization, browsing, measurement, etc.) [29]
- generation of source code
- aspect-oriented programming [44][20]

For example, in the previous chapter, we discussed CARMA which is a logic pointcut language based on the logic meta language SOUL for Smalltalk. As briefly discussed in that chapter, SOUL is basically Prolog. However, SOUL

is different from Prolog in that SOUL has built-in support (under the form of predicate libraries) to reason about Smalltalk code, access the Smalltalk base level and even generate Smalltalk code. CARMA introduces a predicate library for describing join points on top of SOUL as a pointcut language for Smalltalk. Other tools related to logic meta programming and ideas actively being researched are plenty – which proves that logic meta programming is a valuable technique: SOUL [47], TyRuBa [11], JQuery [43], CARMA [20], etc.

All these logic meta programming systems have one thing in common, namely the logic meta languages they propose are all modeled after Prolog. There is however a wealth of other logics out there and these are perhaps more expressive for certain logic meta programming applications. In the next section we investigate a new form of logic meta programming based on temporal logic we dubbed temporal logic meta programming to reason about the execution history of a program.

3.3 Temporal Logic Meta Programming

We define *temporal logic meta programming* as a form of logic meta programming where the logic programming language is based on a temporal logic programming language. The focus of this dissertation is to use temporal logic meta programming as a means for developing an AOP language to express aspects that depend on multiple join points in the execution of a program, such as stateful aspects and context-aware aspects (see Chapter 2). In order to describe a sequence of join points, we note that there is a temporal relation between these join points: if we can describe this relation, we can describe the sequence. For example in Figure 3.5 the execution of a simple program is depicted: the program is fed to an interpreter which executes the commands sequentially, resulting in the events or join points shown on the time line. Describing the execution history – or time line –, is done by saying things like "event A happens before event B" and "event C happens after event B". So our logic meta language needs to be able to express these temporal relations.

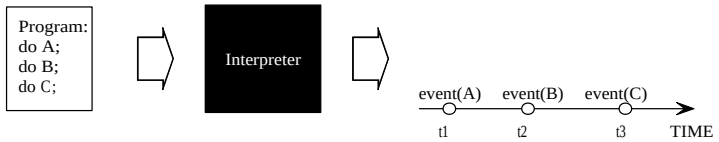

Figure 3.5: The execution history of a program depicted on a time line.

The easiest way to include the notion of time in Prolog, which is used as logic meta language in current logic meta programming applications, is to add an extra variable to every pattern: this temporal variable is then bounded to a time stamp and we express temporal relations by putting constraints on these temporal variables. For example, to describe the execution history in Figure 3.5, we can write the following query in Prolog: `event(X1, T1), event(X2, T2), event(X3, T3), T1 < T2, T2 < T3`. Manual manipulation of time stamps in general is tedious and a natural need to introduce some form of abstraction arises: temporal logic introduces temporal operators and assumes an implicit

temporal context under which each formula is evaluated, as an abstraction mechanism for time manipulation.

The term *temporal logic* [15] refers to all logics that allow the (abstract) representation of temporal relations. In temporal logic, a query is evaluated in relation to time and the truth value of a formula can change over time. Take for example the statement: " Lotte sleeps.": depending on the time this statement is evaluated to (e.g. from 1h30 - 9h30) this is true or false. A temporal logic introduces temporal operators – or temporal connectives – into a logic [2], which abstract the explicit handling of time. The origin of temporal logic lies with modal logic.

3.3.1 Modal Logic

A *modal logic* extends propositional or predicate logic with a new type of operators, called *modalities*. The goal of modal logic is to represent possibility: e.g. "is it necessary that A is true" ($\Box$A) or "is it possible that A is true" ($\diamond$A).

More formally, we define the syntax of classic modal logic as follows. The alphabet of the language consists of:

- a set of propositional variables $P = \{ p, p, ... \}$
- logic connectives $\leftarrow$, $\neg$
- modalities $\Box$, ($\diamond = \neg\Box\neg$)

Then, if ψ and φ are well-formed formula, so are: $p \in P$, $\Box\psi$, $\diamond\psi$, $\varphi\leftarrow\psi$.

The semantics of modal formulas is defined by means of *Kripke structures*. A Kripke structure K (or model) is a 3-tuple consisting of a set W of countable *worlds*, an *accessibility relation* T which determines if it is possible to go from one world to another and an *interpretation function* L which determines which propositions are true in a world (cf. state machine). Figure 3.6 is a graphical representation of a Kripke structure.

A modal formula is evaluated in terms of one of the worlds. For example, in classical modal logic, the formula $\diamond\psi$ resolves to true in a world ω_1 if there is at least one world accessible from ω_1 for which ψ true is and the formula $\Box\psi$ resolves to true in a world ω_2 if ψ is true for all worlds accessible from ω_1[3]. More formally, given a Kripke structure $K = (W, T, L)$ and a world w, we say that a formula is true in a world for a model K if:

1. $K,w \models p$ if $(w, p) \in L$
2. $K,w \models \neg\, \varphi$ if $K,w \not\models \varphi$
3. $K,w \models \varphi \leftarrow \psi$ if $K,w \not\models \psi$ or $K,w \models \varphi$
4. $K,w \models \Box\, \varphi$ if $\forall\, v : (w, v) \in T : K,v \models \varphi$

and a formula φ is true for a model K if: $K \models \varphi$ if $\forall\, w \in W : K,w \models \varphi$

For example, for the Kripke structure in Figure 3.6, formulas $\Box q$ and $\diamond p$ are true in world w_1 ($K,w_1 \models \Box q$, $K,w_1 \models \diamond p$).

[2] Being propositional logic or predicate logic.

[3] Note that this is very different from traditional logic, where all operators are truth-functional, meaning the truth value of a formula depends on the truth value of its components, whereas in temporal logic, the truth value of a formula is relative to the temporal context to which it is evaluated.

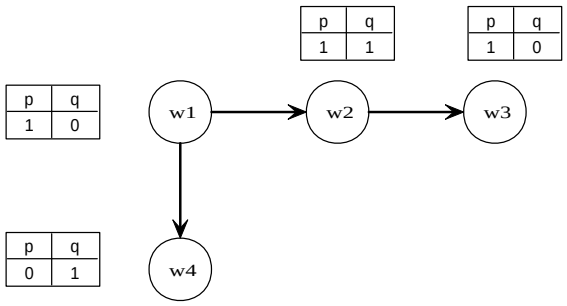

Figure 3.6: Graph for Kripke $K = (W, T, L)$ with
$W = \{ w_1, w_2, w_3, w_4 \}$
$T = \{ (w_1, w_2), (w_1, w_4), (w_2, w_3) \}$
$L = \{ (w_1, p), (w_2, q), (w_2, p), (w_3, p), (w_4, q) \}$

Temporal logic was first introduced by Arthur Prior under the form of Tense logic [10] as a form of modal logic to reason about the past and the future truth value of a formula, which lead to the introduction of the past modalities P, H and the future modalities F and G into propositional logic. More formally, we define the syntax of Tense logic as follows. The alphabet of the language consists of:

- a set of propositional variables $P = \{ p, q, ... \}$
- logic connectives $\leftarrow$, $\neg$
- modalities P, H, F, G

Then, if ψ and φ are well-formed formula, so are: $p \in P$, $P\psi$, $G\psi$, $H\psi$, $F\psi$, $\varphi \leftarrow \psi$.

The semantics of the temporal operators is as follows. Pp expresses that proposition p was true sometime in the past, Fp expresses that proposition p will be true in the future, Hp expresses that proposition p was always true in the past and Gp that proposition p will always be true in the future. For example $Hp \wedge p \wedge Fp$ expresses that there never will be a moment where p is false. More formally, the semantics are defined in terms of *temporal frames*. A temporal frame consists of a set W of time points (time), an ordering relation for these time points, an accessibility relation T which determines if it is possible to go from one time point to another and an interpretation function that defines at each time point the truth value of a proposition. A temporal logic formula is evaluated to a time point. We say that a formula is true at a time point t for a temporal frame $K = (W, <, T, L)$ if:

1. $K,t \models p$ if $(p, t) \in L$
2. $K,t \models \neg \varphi$ if $K,t \not\models \varphi$
3. $K,t \models \varphi \leftarrow \psi$ if $K,t \not\models \psi$ or $K,t \models \varphi$
4. $K,t_1 \models P \psi$ if $\exists\ t_2 : t_2 < t_1 : T,t_2 \models \psi$

5. $K,t_1 \models F\ \psi$ if $\exists\ t_2 : t_1 < t_2 : K,t_2 \models \psi$
6. $K,t_1 \models H\ \psi$ if $\forall\ t_2 : t_2 < t_1 : K,t_2 \models \psi$
7. $K,t_1 \models G\ \psi$ if $\forall\ t_2 : t_1 < t_2 : K,t_2 \models \psi$

Figure 3.7 defines an example temporal frame for Tense logic: from it, we can conclude for example at 12h00 that "lotte has been sleeping" ($K,t_3 \models Plottesleeps$). [4] Depending on the definition of time points, different temporal logics are defined as a variation on tense logic.

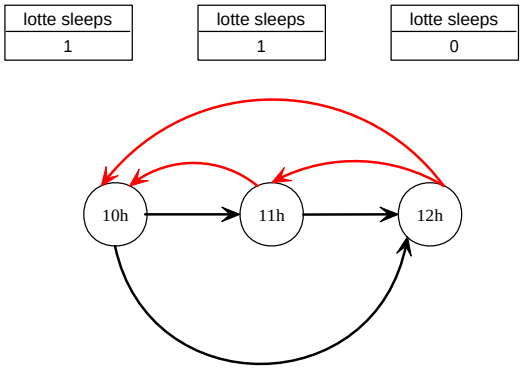

Figure 3.7: Graph for temporal frame $K = (W, , T, L)$ with
$W = \{\ 10h, 11h, 12h\ \}$
$T = \{\ (w_1, w_2) : w_1 < w_2\ \}$
$L = \{\ (10h,$ lotte sleeps$), (11h,$ lotte sleeps$), (12h,$ lotte sleeps$)\ \}$

3.3.2 Classification of temporal logics

If we define time as a set of time points and a time stamp refers to a certain "moment" in time and is used to associate a pattern with time, we can classify temporal logics by the following possible properties of time [16]:

Time points or time intervals : a time stamp refers to a single time point or an interval of time points.

Bounded or unbounded time : if every time stamp is preceded by another time stamp, time is said to be *unbounded in the past*; If every time stamp is succeeded by another time stamp, time is said to be *unbounded in the future*. If on the other hand there is a time stamp for which there is no time stamp preceding it, time is called *bounded in the past*; If there is a time stamp for which there is no time stamp succeeding it, time is said to be *bounded in the future*

[4] The graph is a representation of the Kripke structure, notice two accessibility relations: red arrows to see if one can access a past time point and black arrows to see if one can access a future time point.

Discrete, dense or continuous time : depending on the domain chosen to represent time points, time is said to be discrete, dense or continuous: this naming is used when referring to the set of integers, the set of rational numbers or the set of real numbers to represent time-points respectively.

Linear or branching time : time is called *linear* when the set of time-points is *totally ordered* and when the set of time-points is *partially ordered* time is called *branching*.

In addition to these options, we include the option for a temporal logic to be based on propositional logic or first-order logic.

3.3.3 Temporal logic meta programming experiment: required properties

Remember that one of our goals, is to find a language that allows one to easily express aspects that depend on multiple join points in the execution of a program. For this purpose, we need a pointcut language in which it is easy to describe (ordered) sequences of events generated by program execution and we need to be able to place constraints on the arguments of these events. If we define program execution as a linear sequence of computational steps that starts at a well-defined point in time, we observe that in order to describe such a sequence using temporal logic, the properties we are looking for in a temporal logic are:

1. time stamps refer to time points,
2. time is unbounded in the future (in general, we never know if program execution ever stops), execution starts at a well-defined point in time, so time is bounded in the past.
3. time-points are mapped to integers, because we want to reason about an execution trace in terms of execution time. We could for example map time points to real time, making the temporal formulas be evaluated in relation to real time. This could for example be interesting for expressing time-related business rules using temporal logic, however this is not the focus of this dissertation.
4. linear time: time points are totally ordered. In this dissertation, we chose not consider reasoning about multi-threaded programs.
5. based on FOL (we want to be able to put constraints on the arguments of the events).

In the next section we take a look at metric temporal logic which has these properties.

3.3.4 Metric Temporal Logic

Metric Temporal Logic [9] [8] – or MTL – allows reasoning about linear – possibly infinite – time intervals: MTL introduces temporal operators that allow the logician to say things as "B becomes true sometime within t time points A was true" or "A becomes true, so at the next time-point B is true". The base

logic for MTL is FOL; the properties (section 3.3.2) of MTL are summarized as follows

- time stamps label time points
- discrete time points
- linear time
- time unbounded in the future, time bounded in the past
- base logic: FOL

Syntax and Semantics

The alphabet of MTL consists of:

- a set of constants $C = \{ c_1, c_2, ... \}$
- a set of predicate symbols $P = \{ p_1, p_2, ... \}$
- a set of functor symbols $F = \{ f_1, f_2, ... \}$
- a set of variable symbols $X = \{ X_1, X_2, ... \}$
- logic connectives $\leftarrow$, $\neg$ and derived logic connectives $\vee$, $\wedge$
- temporal operators $\bullet$, $\circ$, $\Box_t$, $\diamond_t$ $(t \in Z \cup \{-\infty, +\infty\})$
- symbols (and)

Then we say that t is a *term*, if t is a variable $(t \in X)$ or t is a constant $(t \in C)$ or if $t = f(t_1, ..., t_2)$ where $t_1 ... t_2$ are terms and f is a functor $(f \in F)$.

Then, if x is a variable $(x \in X)$ and if p is a predicate $(p \in P)$ and t_1, ..., t_2 are terms and if ψ and φ are well-formed formulas, then so are $P(t_1, ..., t_2)$ (an atom), $\neg\psi$, $\phi \leftarrow \psi$, $\bullet\psi$, $\circ\psi$, $\Box_t\psi$, $\diamond_t\psi$, $\diamond_{-\infty}\psi$, $\diamond_{\infty}\psi$, $\Box_{-\infty}\psi$, $\Box_{\infty}\psi$, $\Box_{[t,k]}\psi$ and $\diamond_{[t,k]}\psi$ well-formed formulas.

Informally, the meaning of the temporal operators is as follows (mathematical notation between braces):

- ψ `next` φ (or $\psi \wedge \circ \varphi$) is true if currently ψ is true and at the next time point from now φ is true.
- ψ `previous` φ (or $\psi \wedge \bullet \varphi$) is true if currently ψ is true and at the previous time point from now φ is true.
- ψ `always-within-t-timepoints` φ (or $\psi \wedge \Box_t$ is true if ψ) is currently true and ψ is true for all time points, within t time points from now.
- ψ `sometime-within-t-timepoints` φ (or $\psi \wedge \diamond_t \varphi$) is true if ψ is currently true and for at least one time point, within t time points from now, φ is true.
- ψ `sometime-next` φ (or $\psi \wedge \diamond_{+\infty} \varphi$) is true if ψ currently true and and φ is true some later time point (unrestricted) from now.

- ψ `sometime-previous` φ (or $\psi \wedge \diamond_{-\infty} \varphi$) is true if ψ is currently true and φ is true some earlier time point (unrestricted) from now.
- ψ `always-within-interval-t-k` φ (or $\psi \wedge \Box_{[t,k]} \varphi$) is true if currently ψ is true and for all time points, within t, k time points from now, φ is true.
- ψ `sometime-within-interval-t-k` φ (or $\psi \wedge \diamond_{[t,k]} \varphi$) is true if currently ψ is true and for at least one time point, within t, k time points from now, φ is true.

  meaning that ψ is true at some time-points after φ happens, if one time stamps φ at t and the final time that ψ is true at t.

Formally, the semantics of MTL formulas are defined in terms of *MTL structures*. An MTL structure is a 4-tuple consisting of a set of time points T, the ordering relation $<$, an interpretation function I which assigns at each time point, a truth value for each possible ground atom, and a initial temporal context t_0. A variable assignment α is a mapping α: $X \rightarrow C$.

Given an MTL structure $M = (T, <, t_0, I)$ a time point t and a variable assignment α, we say that a formula is true in a time point t for a model M under a variable assignment α if:

1. M,α,$t \models p(t_1, ..., t_n)$ if $(p(\alpha(\mathrm{t}_1), ...,\alpha(\mathrm{t}_n)), t) \in I$
2. M,α,$t \models \neg\ \phi$ if M,α,$t \not\models \phi$
3. M,α,$t \models \phi \leftarrow \psi$ if M,α,$t \not\models \psi$ or M,α,$t \models \phi$
4. M,α,$t \models \bullet\psi$ if M,α,$t-1 \models \psi$
5. M,α,$t \models \circ\psi$ if M,α,$t+1 \models \psi$
6. M,α,$t_1 \models \diamond_c\psi$ if $\exists\ t_2$:
   $t_1 \leq t_2 \leq t_1 + c$: T,α,$t_2 \models \psi$ if $c \geq 0$ or
   $t_1 + c \leq t_2 \leq t_1$: T,α,$t_2 \models \psi$ if $c \leq 0$.
7. M,α,$t_1 \models \Box_c\psi$ if $\forall\ t_2$:
   $t_1 \leq t_2 \leq t_1 + c$: T,α,$t_2 \models \psi$ if $c \geq 0$ or
   $t_1 + c \leq t_2 \leq t_1$: T,α,$t_2 \models \psi$ if $c \leq 0$.

and a formula ψ is true for a MTL structure M, if M,α,$t_0 \models \psi$:
E.g. $M = (\{1, 2\}$,¡,$1, \{ (1, p(a)\ , 0), (2, p(a), 1))\}) \models \circ\ p(a)$.

Metric temporal logic programming is based on a subset of MTL, meaning that metric temporal logic programming programs are built out of a subset of metric temporal logic. The reason why metric temporal logic programming is based on a subset of MTL, is the same reason that Prolog is based on definite clause logic, a subset of first-order logic: this allows metric temporal logic to become executable [9]. A metric temporal logic programming program consists of a set of MTL clauses and we can query a program by means of an MTL query Q, defined by:
$Q ::= \epsilon \mid A$[5] $\mid \circ Q \mid \bullet Q \mid \diamond_t Q \mid Q \wedge Q$

[5] Atom.

and an metric temporal logic programming clause C is defined by:

$C ::= \mathrm{A} \mid \circ C \mid \bullet C \mid \Box_t C \mid C \leftarrow Q$

So, one can only use the temporal operator $\diamond_t$ in the body of a clause or a query and the operator $\Box_t$ is not allowed in the head of a clause.

An example program, is depicted in Figure 3.8[6]. In this example, we assume that the time points refer to years. The program is a so-called historical database: it stores the career information of a person named "john" over the past 20 years. The first clause stores for example the information that john was a salesman between 16 and 20 years ago and the second clause states that he managed the sales department between 15 and 11 years ago. The last clause states that, at any time, if a person manages a department, that person is a manager.

1. $\Box_{[-20,-16]}\ salesman(john)$
2. $\Box_{[-15,-11]}\ manages(john, sales)$
3. $\Box_{[-10,-6]}\ manages(john, development)$
4. $\Box_{[-10,-6]}\ manages(john, board)$
5. $\Box\ (manager(Person) \leftarrow manages(Person, Department))$

Figure 3.8: Metric temporal logic programming program.

Example queries are depicted in Figure 3.9. The first query means "was john a manager in the past 20 years" and the second one asks something like "who was a manager last year". In order to find the answer to these queries, MTL must be made executable – the same way as definite clause logic is made executable for Prolog–: Therefore, the program and queries are translated into constraint logic programs [9], so that finding out the answer to a query can be done using constraint logic programming resolution [9]. Constraint logic programming resolution is similar to Prolog resolution, but in addition, the constraints that are encountered during resolution, must be satisfiable. When a query is being resolved, constraints are collected in a constraint repository and at each unification step, these constraints are checked for satisfiability; This is different from Prolog, where a constraint must simply be true when it is being resolved. E.g. given the program:

```
B(X,1):-X<0.
B(X,Y):-X=1, Y>0.
A(X,Y):-X>0, B(X,Y).
```

the query `A(X, 1)` results in resolving `X>0, B(X,1)` resulting in putting the constraint `X>0` in a constraint repository and resolving `B(X, 1)`, which in turn results in the answer X⇀1.

1. $\diamond_{-20}\ manager(john)$
2. $\bullet\ manager(Person)$

Figure 3.9: MTL query.

[6]Example taken from [9])

1.	$\Pi(\psi)$	$= \pi(\psi, 0, \{\})$ where
2.	$\pi(\bullet, \psi, t, S)$	$= \pi(\ \psi, (t - 1), S)$
3.	$\pi(\circ, \psi, t, S)$	$= \pi(\ \psi, (t + 1), S)$
4.	$\pi(\diamond_t\psi, t, S)$	$= \pi(\psi, (t + x), S \cup \{0 \leq x \leq c\ \}$ if $c > 0)$
5.	$\pi(\diamond_t\psi, t, S)$	$= \pi(\psi, (t + x), S \cup \{c \leq x \leq 0\}$ if $c < 0)$ (with x a new variable)
6.	$\pi(\Box_t\psi, t, S)$	$= \pi(\psi, (t + x), S \cup \{0 \leq x \leq c\ \}$ if $c > 0)$
7.	$\pi(\Box_t\psi, t, S)$	$= \pi(\psi, (t + x), S \cup \{c \leq x \leq 0\}$ if $c < 0)$ (with x a new variable)
8.	$\pi(\psi \leftarrow \varphi$, t, S)	$= \pi(\psi$, t, S) $\leftarrow \pi(\varphi$, t, S)
9.	$\pi(\psi \wedge \varphi$, t, S)	$= \pi(\psi$, t, S) $\wedge\ \pi(\varphi$, t, S)
10.	$\pi(\mathrm{p}(\mathrm{t}_1, ..., t_n), t, S)$	$= \mathrm{p}(\mathrm{t}, \mathrm{t}_1, ..., t_n) \wedge$ S

Table 3.2: Rules for translating MTL formulas to FOL formulas.

In order to translate metric temporal logic programming programs to constraint logic programs, it suffices to add an additional argument to each predicate and to express the temporal relations defined by the temporal operators by putting constraints on these temporal variables (see [9]). The translation function Π is depicted in Table 3.2.

For example, the program from Figure 3.8 can be translated into an equivalent Prolog program (Figure 3.10) and so can the queries (Figure 3.11) revealing us that john was a manager in the last 20 years and that john was a manager last year.

1. $salesman(X, john) \leftarrow X \leq$ -16; $X \geq$ -20
2. $manages(X, john, sales) \leftarrow X \leq$ -11; $X \geq$ -15
3. $manages(X, john, development) \leftarrow X \leq$ -11; $X \geq$ -15
4. $manages(X, john, board) \leftarrow X \leq$ -11; $X \geq$ -15
5. $manager(X, Person) \leftarrow manages(X, Person, Department)$

Figure 3.10: Metric temporal logic programming program from Figure 3.8 translated to Prolog.

1. $manager(X, john)$, $-20 \leq X \leq 0$
2. $manager(X, Person)$, $X = -1$

Figure 3.11: Metric temporal logic programming queries from Figure 3.9 translated to Prolog queries.

3.3.5 HALO: a subset of metric temporal logic

In this section we define a subset of metric temporal logic, which is the basis of our pointcut language HALO, defined in the next chapter. The temporal operators we restrict ourselves to in HALO are the past operators from MTL, namely the operators $\bullet$, $\diamond_{-\infty}$ and $\diamond_{[a,b]}$. We then define a HALO clause as in Table 3.3. A clause is always defined under the operator $\Box_{[-\infty,\infty]}$ and consists of a head and a body, where the head is an atom and the body is a conjunction of formulas. The formulas are restricted to be an atom or a conjunction of formulas as argument of one of the temporal operators $\bullet$, $\diamond_{-\infty}$ or $\diamond_{[a,b]}$, where the first formula in that conjunction is an event pattern (an atom with predicate = *call* or *create*) . E.g. $\Box_{[-\infty,\infty]}(advice(Discount, User) \leftarrow call(checkout, User) \wedge \diamond_{-\infty} call(login, User))$.

$clause$	::	$\Box_{[-\infty,\infty]}$ ($< atom > \leftarrow < conjunction >$)
$conjunction$	::	$< formula > \{\wedge < formula > *\}$
$formula$	::	$< atom > \mid < temporal > (< event > \wedge < conjunction >)$
$temporal$	::	$\bullet \mid \diamond_{-\infty} \mid \diamond_{[l,r]}$
$event$	::	$< call > \mid < create >$
$call :: call(F, A)$		
$create :: create(C, I)$		

Table 3.3: HALO as a subset of MTL.

Under these restrictions, we can transform the rewrite rules depicted in Table 3.2 into an algorithm for compiling MTL to Prolog.

For, example, given the rules in Table 3.2, we observe the following equivalences, for the operator $\diamond_{-\infty}$ for rule nr. 5 in Table 3.2:
$p(a) \wedge \diamond_{-\infty} q(b)$
$\Leftrightarrow p(X, a) \wedge q(Y + X, b) \wedge -\infty \leq Y \leq 0 \wedge$Z = Y + X
$\Leftrightarrow p(X, a) \wedge q(Z, b) \wedge -\infty \leq Z - X \leq 0$
$\Leftrightarrow p(X, a) \wedge q(Z, b) \wedge Z \leq X$

Furthermore, we restrict the semantics of the temporal operator $\diamond_{-\infty}$ such that the formula $\diamond_{-\infty}(call(X, Y) \wedge conjunction)$ matches only the most recent, past $call(X, Y)$ for which the entire formula is true. E.g. $\diamond_{-\infty}(call(display, X) \wedge X > 2)$, $3 \models call(display, 3)$ and $\not\models call(display, 4)$ if we know that at time point 1 $call(display, 4)$ is true and at time point 2 $call(display, 3)$ is true.

Similarly, for the operator $\diamond_{[a,b]}$, we observe the equivalences for rules nr. 4,5 in Table 3.2:
$p(a) \wedge \diamond_{[c,d]} q(b)$
$\Leftrightarrow p(X, a) \wedge q(Y + X, b) \wedge c \leq Y \leq d \Leftrightarrow Z = Y + X$
$\Leftrightarrow p(X, a) \wedge q(Z, b) \wedge c \leq Z - X \leq d$
$\Leftrightarrow p(X, a) \wedge q(Z, b) \wedge c + X \leq Z \leq d + X$

For the operator $\bullet$, we observe the equivalences for rule nr. 2 in Table 3.2:
$p(a) \wedge \bullet q(b)$
$\Leftrightarrow p(X, a) \wedge q(X - 1, b)$
$\Leftrightarrow p(X, a) \wedge q(Z, b) \wedge Z = X - 1$

The algorithm now works as follows. In order to compile a clause, we must first compile away the $\Box$: for this purpose, we generate a new temporal variable

```
1. result = nil.
2.
3. def compile-rule(rule):-
4.   temporal-var = gensym()
5.   compile-to-prolog(temporal-var, head(rule))
6.   head = result
7.   result = nil
8.   compile-to-prolog(temporal-var, body(rule))
9.   body = result
10.  result = nil
11.  return rule(head, body)
```

Figure 3.12: Pseudo code for compiling restricted MTL clause to Prolog.

(nr. 4 in Figure 5.2), which we use to compile the head of the clause (nr. 5 in Figure 5.2) and the body of the clause (nr. 8 in Figure 5.2) respectively (derived from rule nr. 6 in Table 3.2).

Compiling the head or the body of a clause is done using the function `compile-to-prolog` depicted in Figure 5.3. This function takes as argument a temporal variable and a formula to compile. The temporal variable is added as an argument to the formula, when the formula is an atom (nr. 19 in 5.3 and rule nr. 10 in table 3.2), otherwise, in case the formula being compiled is one with a temporal operator, we generate a new temporal variable and compile the arguments of the temporal operator, using this new temporal operator and we add a constraint between the new temporal variable and the old one to the compiled formula, to express the temporal relation intended by the temporal operator (e.g. line nr. 4-8 in Figure 5.3 and the redefinition of rule nr. 5). Note that in order to reflect the semantics for the `sometime-past` operator and the `sometime-interval` operator, we add some extra constraints, see line nr. 6 and line nr. 16 and line nr. 27 - 32.

E.g. compiling the clause $\Box_{[-\infty,\infty]}(a(b) \leftarrow b(b) \wedge \diamond_{-\infty} c(d))$ results in the Prolog clause:

```
1. a(T, b) :-
2.   b(T, b),
3.   findall(L, c(L, d), Opl),
4.   largest-number(K, Opl),
5.   c(K, d),
6.   K < T.
```

The lines nr. 3 - 4 compute all solutions for the query $\diamond_{-\infty} c(d)$, and lines nr. 5 - 6 make sure that the solution for $c(K, d)$ with the largest time stamp K is selected.

We define HALO as the subset of MTL defined in this section for two reasons. First of all, we only allow the past MTL operators $\bullet$, $\diamond_{-\infty}$ and $\diamond_{[a,b]}$ because we observe, that in the examples of context-aware aspects and event-based aspects described in Chapter 2, all the aspects are all triggered at a join point, depending on whether there was some condition true at a *past* join point: e.g. give a discount on checkout if promotions were active at login, give a discount when an item is added to the shopping basket if there was a promotion active when the item was added to the shopping basket, etc. Furthermore, when we evaluate a pointcut at a join point, we only have access to the execution *history* so far, so what semantics would we give to pointcuts referring to join points in

```
1.  def compile-to-prolog(temporal-var, conjunction):
2.    pattern = first(conjunction)
3.    if sometime-past-p(pattern):
4.         new-temporal-var = gensym()
5.         new-temporal-var2 = gensym()
6.         add-findall(new-temporal-var, temporal-var, new-temporal-var, arguments(pattern))
7.         compile-to-prolog(new-temporal-var, arguments(pattern))
8.         result and= new-temporal-var < temporal-var
9.    elif previous-p(pattern):
10.        new-temporal-var = gensym()
11.        compile-to-prolog(new-temporal-var, arguments(pattern))
12.        result and= new-temporal-var is (temporal-var - 1))
13.   elif sometime-interval-p(pattern):
14.        new-temporal-var = gensym()
15.        new-temporal-var2 = gensym()
16.        add-findall2(new-temporal-var, temporal-var, new-temporal-var, arguments(pattern))
17.        left = left-interval(pattern)
18.        right = right-interval(pattern)
19.        compile-to-prolog(new-temporal-var, arguments(pattern))
20.        result and= new-temporal-var > (left + temporal-var)
21.        result and= new-temporal-var > (right + temporal-var)
22.   else:
23.        pattern = add-argument(pattern, temporal-var)
24.        result += pattern
25.   compile-to-prolog(temporal-var, rest(conjunction))
26.
27. def add-findall(temporal-var, new-temporal-var, new-temporal-var2, conjunction):
28.        result and= findall(new-temporal-var2,
29.        compile-to-prolog(new-temporal-var2, conjunction)
30.        result ,= new-temporal-var2 < temporal-var
31.        result ,= Opl)
32.        result and= largest-number(new-temporal-var, Opl)
```

Figure 3.13: Pseudo code for compiling restricted MTL conjunction to Prolog.

the future? Therefore, it seemed a natural choice to focus on the past operators in this dissertation and see what we can express using these. The reason for restricting the semantics of the operators somewhat (e.g. for the operator $\diamond_{-\infty}$), is because this allows a *reasonably* efficient implementation (in relation to garbage collection of facts, etc. all discussed in chapter 5).

3.4 Summary

In this chapter we presented the basic concepts behind logic programming and logic theory. We introduced Prolog as an example logic programming language, because Prolog is the basis for many logic meta programming applications. Next we introduced the notions of logic meta programming and listed the example applications of logic meta programming investigated so far. We noticed that current logic meta programming research is mostly based on Prolog and we tried to use Prolog as a basis for solving our problem: namely designing a pointcut language that allows to write down pointcuts that relate multiple join points in the execution of a program. We noticed that between any two such join points, a temporal relation exists and that if we can express this relation, we can describe the execution history of a program. The solution based on Prolog involved extending all predicates with temporal variables, keeping a spot to store a time stamp, and expressing temporal relations by putting constraints on these temporal variables. Manipulating time stamps is however tedious and we looked at temporal logic, which is a formalism dedicated to reasoning about

temporal relations.

Temporal logic is all about reasoning about temporal relations: a temporal logic extends a logic with temporal operators, that abstract the explicit handling of time and queries are evaluated to an implicit temporal context. Next we introduced metric temporal logic, which allows one to reason about time intervals, as an example temporal logic language and we chose this language, based on its properties, we decided to base our poincut language on a subset of metric temporal logic.

Chapter 4

A temporal logic pointcut language

In this chapter we introduce a temporal logic pointcut language, based on metric temporal logic, which has built-in predicates to express temporal relations between join points as a declarative means to write down pointcuts that describe a sequence of join points in the execution of a program needed to implement stateful and context-aware aspects. Throughout this chapter, we refer to this pointcut language as "HALO" – short for History-based Aspects using Logic – which is the name given to the prototype implementation. Pointcuts in HALO range over multiple join points in the execution history of a program. When a pointcut about multiple (past) join points is evaluated in HALO, this is done in respect to the the program state at these (past) join points, so that constraints about past join points are checked against the program state at the occurrence of a past join point. This makes it possible to reason about current and past program state in HALO, furthermore it is possible to put constraints on a past join point in a pointcut, that involve values from a later join point.

In the rest of this chapter, we define the syntax and semantics of HALO. However, before going into details, we give a short overview of HALO by comparing HALO to the logic pointcut language after which HALO was designed, namely CARMA. Next we cover the syntax and semantics of pointcuts, advices and aspects more extensively. We then continue the chapter by explaining the use of HALO by means of some example applications. A first example illustrates how HALO can be used to implement some history-based aspects in a sample e-commerce application. A next example illustrates the use of HALO in a video game application. These experiments illustrate the applicability and use of HALO.

4.1 HALO versus CARMA: an overview

HALO is an instance of (temporal) logic meta programming (see Chapter 3): this means HALO is a logic meta programming approach to aspect-oriented programming, similar to CARMA (see Chapter 2), as will be shown. Remember that in a LMP system, one uses a logic language as a meta language to reason about programs written in some other language, called base language. The

target base language for the prototype implementation is Common Lisp and the meta language is based on metric temporal logic (see Chapter 3). The general idea is that we use a temporal logic language to reason about the (past) execution history of a Lisp program, allowing one to write down pointcuts that match part of the execution history, rather than a single join point. As HALO is based on CARMA, it has many of CARMA's advantages, but also adds some novel ideas.

HALO allows one to write down pointcuts *declaratively* that relate different events in the execution of a program, so that we can match part of the execution history in a pointcut. As in CARMA, join points represent events in the execution of a program: CARMA's primitive pointcuts and their matching counterparts in HALO are depicted in Table 4.1. CARMA is implemented for Smalltalk, which is based on the message-passing model for methods, whereas CLOS uses the generic function model, which justifies why we use "call" in HALO as predicate name for the pointcut that describes a method invocation. Furthermore we explicitly added a primitive pointcut "create" to describe a class instantiation join point. This is not necessary in CARMA because in Smalltalk an instance creation event is a message send "new". The reason there is no primitive pointcut in HALO for capturing slot assignments or references, is that it is a convention in CLOS to access and modify slots through *accessors*, which are simply methods that can be captured using the primitive pointcut for method calls[1].

	CARMA	HALO
Message reception/Method call	`reception(?jp, ?selector, ?arguments)`	`(call ?methodName ?arguments)`
Message send/Method call	`send(?jp, ?selector, ?arguments)`	N/A
Assignment/Access	`assignment(?jp, ?varName, ?oldValue, ?newValue)`	N/A
Reference/Access	`reference(?jp, ?varName, ?value)`	N/A
Instance creation	N/A	`(create ?className ?instance)`

Table 4.1: CARMA's primitive pointcuts compared to HALO's primitive pointcuts.

Pointcuts can be composed to define complex pointcuts: in CARMA the logic connectives are available for this purpose, but HALO also introduces temporal operators[2] based on the temporal operators in MTL. An overview is given in Table 4.2. The purpose of the temporal operators is to use them to express a temporal relation that relates two pointcuts. The execution of a program, results in generating a sequence of join points: e.g. the execution of the program Figure 3.5 can be depicted as a sequence of successive events on a time line (same Figure). The temporal operators can express things like "event A is before event B on the time line". Note that the availability of these connectives in HALO is *the main* difference between HALO and CARMA.

[1]However, it is possible to implement primitive pointcuts to capture slot access and assignment using the CLOS MOP, and this can be added as an extension to HALO (see next chapter).

[2]Technically, temporal operators are not connectives, but rather higher-order predicates (see MTL in chapter 3). However, since the purpose of the operators in HALO is to describe the order in which join points occur, it makes more sense to see them as connectives for these join points. More details in section 4.2

CARMA		HALO
logic and	,	a space
logic not	not(*pointcut*)	(not *pointcut*)
the previous join point	N/A	(*pointcut* (prev *pointcut*))
a past join point	N/A	(*pointcut* (sometime-past *pointcut*))
a previous join point, within a time interval	N/A	(*pointcut* (sometime-past-interval ?l ?r *pointcut*))

Table 4.2: CARMA's composition predicates compared to HALO's.

Similarly to accessing Smalltalk in a CARMA pointcut (by using [] syntax), one can access Lisp in a pointcut definition to evaluate a Lisp form, allowing one to compute a value at the base level needed to evaluate a pointcut (by means the of `escape` predicate). This implies there is some form of *linguistic symbiosis* [18] between HALO and Lisp, as it is possible to use the values of HALO in Lisp and vice versa.

HALO uses, just as CARMA, logic variables and unification, instead of a wild card mechanism, to quantify the arguments of a pointcut, which is very handy to write advices that can refer to these variables or pointcuts that constrain the same join point value multiple times.

Furthermore HALO allows the definition of *contexts*, which are predicates used to constrain pointcuts by placing conditions on the arguments of the events they describe: this is also possible in CARMA. An important property of HALO is that the time on which an event takes place is taken into account, allowing one to reason about program state (context) at a (past) join point, captured by the use of a temporal operator in a pointcut.

Examples exploiting the properties we just discussed can be found in a subsequent section, but first we cover the HALO syntax and semantics in full detail.

4.2 Syntax and semantics

An overview of the grammar (Backus-Nauer form) can be found in Table 4.3.

4.2.1 Patterns

The basic building blocks in HALO are patterns. A pattern is written down as an s-expression (a Lisp form): it is a list whose first argument is a predicate name and the remainder of the arguments are Lisp values or logic variables (a Lisp symbol starting with a "?"): e.g. `(A ?x 1)`.

4.2.2 Pointcuts

The semantics of a pointcut is that it is a predicate, associated with advice code, that is applied to each event (join point) generated during program execution and if the pointcut evaluates to true for an event, the advice code is executed. Syntactically, a pointcut is defined as a list of primitive pointcuts that are composed using logic connectives or temporal operators.

There are three kinds of primitive pointcuts:

event captures an event in the execution of a Lisp program such as a method call or an instance creation (described by a pattern with predicate name

advice-definition	::	(defrule <advice-code> <pointcut>)
advice-code	::	(advice string symbol (<term>*))
event	::	<call> | <create>
call	::	(call symbol (<term>*))
create	::	(create symbol <variable>)
escape	::	(escape <variable> <lisp-form>)
context	::	(<predicate> <term>{ <term>}*)
primitive-pointcut	::	<event> | <escape> | <context>
pointcut	::	(<primitive-pointcut> {<pointcut>)}*
pointcut	::	(<connective> {<pointcut>)}*
pointcut	::	(sometime-interval number number {<pointcut>}*)
connective	::	sometime-past | prev | not
predicate	::	a symbol
lisp-form	::	a lisp-form, can contain <variable>
variable	::	a symbol starting with a question mark
constant	::	a symbol
term	::	<constant> | <variable>

Table 4.3: HALO grammar.

"call" or "create", the patterns are referred to as "call pattern" and "create pattern" respectively)

escape can be used to evaluate a Lisp form containing logic variables and bind the value to a logic variable (described by a pattern with predicate name "escape", the pattern is referred to as an "escape pattern").

context is a conditional pointcut (a new predicate defined by a rule: a pattern using this new predicate is referred to as a "context pattern").

The arguments of the predicates representing the different primitive pointcuts, are the following.

call

A call pattern is a pattern of three arguments, being and representing in order:

1. `call` symbol as predicate name.
2. a string naming the method being called during program execution.
3. a list of logic variables and values that represent the arguments of the method being called.

For example, the pointcut `(call "checkout" (?User))` matches all method call join points where the name of the method being called is "checkout" and the method has only one argument, named by the logic variable `?User`.

create

A create pattern is a pattern of three arguments, being and representing in order:

1. `create` symbol as predicate name.
2. a class symbol, referring to the class being instantiated during program execution.
3. a logic variable or value, referring to the create instance.

For example, the pointcut `(create User (?User))` matches all join points that are instantiations of the class `User`.

escape

An `escape` pattern is a pattern of three arguments:

1. `escape` symbol as predicate name.
2. a logic variable.
3. a Lisp-form.

For example, evaluating the pointcut `(escape ?Nr (nr-of-gifts ?Shop))` on a join point, triggers a call to the method `nr-of-gifts` on a logic variable `?Shop` which needs to be bound to an object for which the method applies and the result of this method call (a Lisp value) is bound to the logic variable `?Nr`.

context

```
(defrule (promo-active ?User)
    ((escape ?Shop (shop ?User))
     (escape ?Active promo-on ?Shop)))
```

Figure 4.1: HALO context definition.

A context predicate is a new predicate, defined by a separate rule. For example, in Figure 4.1 a valid context definition is depicted. The purpose of a context predicate is to place some conditions on the arguments of an event pattern. For example the previously defined context can be used as in this pointcut:

```
((call "login" ?User) (promo-active ?User))
```

Now that we have seen the different predicates to define primitive pointcuts, we cover the predicates (or connectives) to combine them with other pointcuts into complex pointcuts. There are two kinds of connectives:

logic connectives : such as `and` and `not`.

temporal operators : such as `prev`, `sometime-past`, `sometime-past-interval`.

and

Pointcuts are connected by an "and" if one places the pointcuts in the same list, there is no need to explicitly use the predicate `and`: e.g. the pointcut `((call "login" ?User) (promo-active ?User))`. For a poincut, which is a composition of two poincuts, by means of an "and", to evaluate to true for a join point, both of the connected pointcuts need to evaluate to true for the join point.

not

The `not` predicate takes a pointcut as its only argument: e.g. `(not (promo-active ?User))`. For such a pointcut to evaluate to true, the pointcut – the argument of the `not` – cannot be proven to be true (negation by failure).

previous

The `previous` predicate takes a pointcut as its only argument. However, the idea is to use a `previous` pattern in combination with another pointcut. E.g. `((call "authenticate" ?User) (previous (call "login" ?User)))`. Such a pointcut evaluates to true for a join point at a time point t, if the pointcut which is the argument of the `previous` predicate, evaluates to true for the join point exactly one time point before t. So the example given earlier describes an execution history that looks like as in Figure 4.2.

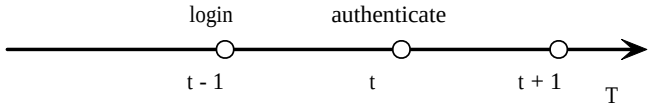

Figure 4.2: Temporal relation `previous` depicted on a time line.

sometime-past

The `sometime-past` predicate takes a pointcut as its only argument. However, the idea is to use a `sometime-past` pattern in combination with another pointcut. E.g. `(call "checkout" ?User) (sometime-past (call "login" ?User))`. Such a pointcut evaluates to true for a join point at a time point t, if the pointcut which is the argument of the `sometime-past` predicate, evaluates to true for any join point at a time point before t. Note that only the most recent join point is matched. So the example given earlier describes an execution history that looks like as in Figure 4.3.

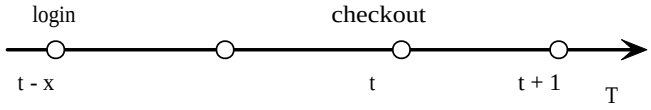

Figure 4.3: Temporal relation `sometime-past` depicted on a time line.

sometime-interval

The `sometime-interval` predicate takes three arguments: two integers a and b and a pointcut. However, the idea is to use a `sometime-interval` pattern in combination with another pointcut. E.g. `((call "checkout" (?User)) (sometime-interval -1 -10 (call "buy" (?User ?Shop))`. Such a pointcut evaluates to true for a join point at a time point t, if the pointcut which

is the argument of the `sometime-interval` predicate, evaluates to true for any join point at a time point within a, b time points from t. So the example given earlier describes an execution history that might look like as in Figure 4.4.

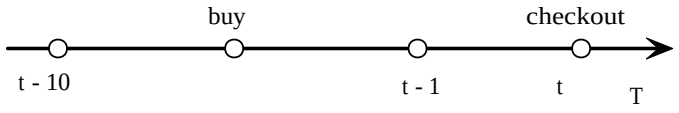

Figure 4.4: Temporal relation `sometime-interval` depicted on a time line.

4.2.3 Defining aspects

In HALO, aspects are not defined in a special class or some other key-worded module, but they are defined as a set of rules, extending the `advice` predicate definition. The `advice` predicate takes three arguments:

1. a string, referring to a name for the advice [3].
2. a function symbol, referring to the method or function implemented at the base level (Lisp) to implement the advice (advice code).
3. a list of variables, denoting the arguments for the function that implements the advice: this list can contain logic variables and ordinary Lisp values.

E.g. `(advice "gift-on-checkout" gift (?User coupon))`. Please note that current HALO implementation only supports a before advice, and this is the default semantics of the `advice` pattern. In this dissertation we focus on the temporal pointcut language, rather than the advice language.

```
(defrule
    (advice "authenticate" authenticate (?User))
    ((call "checkout" (?User))
     (not (sometime-past (call "login" (?User))))))
```

Figure 4.5: HALO security aspect.

A rule is defined using the `defrule` macro as depicted in Figure 4.5. The first argument of the macro is the head of the rule and the second argument is the body of the rule. The rule means something like "execute the advice authenticate if a method called checkout is called now and there was no call to a method called login before that checkout".

4.3 HALO by example

In this section we give two example applications to illustrate the use of HALO more concretely. First, we discuss an e-commerce application and then we discuss a video game application.

[3] Only there for debugging purposes

4.3.1 The e-commerce application in HALO

In this section we discuss the implementation of the e-commerce application described in section 2.4.1, because we want to illustrate that is possible to implement the context-aware aspects using HALO more declaratively than the implementations based on AspectJ, Reflex etc. discussed in that section.

Let's see how we can implement some example aspects based on the possible scenarios for assigning and processing a promotion for the e-commerce application illustrated in section 2.4.1.

A first example is Figure 4.6. It states that a user gets a 10% discount when she checks out, and at that time there is seasonal promotion active.

```
;; Give 10% discount on checkout if there
;; is a seasonal promotion active at that time

;; CONTEXT definition
(defrule (seasonal-promo)
  (escape ?D (christmas-p (today)))

;; ADVICE definition
(defrule (advice "discount" discount '(?User 10))   ; head
  (call "checkout" (?User))                         ; body
  (seasonal-promo))                                 ; body
```

Figure 4.6: HALO discount aspect. Give a user a 10 % discount at checkout if there is a seasonal promotion active.

The implementation of the advice, the discount method, is a plain method (Figure 4.7) and can of course be reused in another aspect definition.

```
(defmethod discount ((u user) percentage)
    "Give the user a percentage discount on each item"
    (let ((user-basket (basket u)))
        (dolist (art (articles user-basket))
            (setf (price art)
                  (- (article-price art)
                  (/ (* (article-price art) percentage) 100))))))
```

Figure 4.7: HALO advice code. Give a user a 10 % discount at checkout if there is a seasonal promotion active. The discount method merely iterates over all the articles in the user's basket and applies the respective discount to those articles.

Note that the promotion context is defined as a separate rule (`seasonal-promo`) and can also be used to implement a second aspect depending on that promotion context, such as an aspect responsible for displaying banners at the website (Figure 4.8).

A second aspect is illustrated in Figure 4.9. This example illustrates that not only past events are captured, but also past values. The activation state of the shop is the state it had when the user logged in. The example is made more interesting by giving customers a gift, but only when there are still gifts left. This refers to the *current* state of the shop. The combination of reasoning about past and current values does not pose any problems.

We present another example in Figure 4.10. This aspect is more complex than the previous examples because inside the `sometime-past` operator we refer

```
;; Pop up banners if there is a seasonal promotion.

;; ASPECT definition
(defrule (advice "pop-up-banner" pop-up-banner '(?User))
  (create User (?User))
  (seasonal-promo))
```

Figure 4.8: HALO banner aspect. When a user connects to the e-commerce website, a banner is popped up if there is a seasonal promotion active.

```
;; CONTEXT definitions
(defrule (gifts-depleted ?User)
  (escape ?Shop (shop ?User))
  (escape ?Nr (nr-of-gifts ?Shop))
  (equal ?Nr 0))

;; ADVICE definitions
(defrule (advice "gift-on-checkout" gift '(?User))
  (call "checkout" (?User))
  (sometime-past
    (call "login" (?User))
    (seasonal-promo))
  (not (gifts-depleted ?User)))
```

Figure 4.9: HALO gift aspect. A user gets a gift on checkout if there are still gifts in stock and she logged in when a seasonal promotion was active.

to a variable `?Article` that is bound in the call form for the buy event. The buy event takes place after the login event but the pointcut for the login event puts a condition on a variable bound by the buy event. This particular example shows off the expressiveness of HALO but imposes some difficulties on the weaver (see next chapter).

```
;; CONTEXT definitions
(defrule (stock-promo-active (?User ?Article))
  (escape ?Shop (shop ?User))
  (stock-overflow (?Shop ?Article)))

;; ADVICE definitions
(defrule (advice "discount" discount (?User ?Article 0.10))
  (call "buy" (?User ?Article))
  (sometime-past
    (call "login" (?User))
    (stock-promo-active (?User ?Article))))
```

Figure 4.10: HALO discount aspect. Give a 10 % discount on the current item bought, as long as the promotions for that type of item were active when the user logged in (e.g. the shop does a promotion for articles with a stock overflow).

4.3.2 Video game application

As a second example, we consider a video game application. The video game we discuss here, is a so-called "shooter-game"; A game consists of several scenarios called levels that each represent a virtual play field. The goal of the game is to move one's character to the exit of the game field, shooting monsters that

pop up to try and kill the player, and to advance to the next level. Note that multiple players can cooperate to complete a scenario. For example, Figure 4.11 is a graphical representation of a game-field: it is comparable to a chess board.

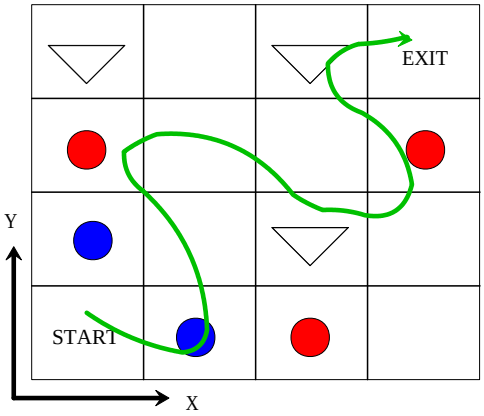

Figure 4.11: Graphical representation of a game field. The blue circles represent player-controlled characters. The red circles are computer-controlled monsters. The triangles represent obstacles, which implies the square they occupy can't be crossed by a character. The green arrow represents a possible path the players can take to the exit.

From this description, we distill the UML diagram of the game in Figure 4.16 (note that get and set methods for the slots are omitted for brevity). The class `GameController` is responsible for "hosting" a game, which means it creates a game (slot `playField`), can pause a game (method `pause`) and set the game's difficulty (method `adjustDifficulty`): it is a sort of server that accepts player connections and is responsible for creating/deleting a player character for people that wish to join/leave the game field it hosts (method `login/logout`).

The class `PlayField` represents a game field (e.g. Figure 4.11). The field is a chess board like plate (sizes of X-axis and Y-axis stored in slots `X-size` and `Y-size` respectively), one special square is reserved for the *exit* of the field (slot `exit-coord`) and another square is reserved for the `start` (slot `start`) where players start the game. The field also has a difficulty set (slot `difficulty`). The field's square's can be occupied by `PlayFieldObject`, such as monsters, obstacles or players. The idea is that every t minutes, the play field adds a monster on a random square, the monster is parametrized by the play field's difficultly (methods `spawnMonsters`, `spawnMonster`).

The last classes we need to discuss are the different `PlayFieldObject`. All of them share the fact that they have hit points (slot `hitPoints`), which can be affected (method `addHitPoints`), and they keep a reference to the square they occupy (slot `occupying`); Such an object can be "killed" (method `kill`), this means the object is removed from the square it removes when its hit points drop below 0. The class `PlayFieldObject` is specialized for three kinds of objects. The `Player` class represents a human-controlled character; The player chooses a name for his character (slot `name`) an when it is created, it is parametrized by

a type (argument `type` of constructor), which scales the number of hit points and the attack power the player has: e.g. a "sniper" has less hit points than a "soldier", but more attack power than a "soldier". The second special object is the class `Monster`: when a monster is created, it gets a number of hit points and attack power scaled by the difficulty the game is set when the monster is created. Monster and player objects can move to a square (method `advance`) and attack a square (method `attack`), which results in subtracting the hit points of the object occupying the square being hit. The last special object is `Obstacle`: an obstacle is just an immobile object that blocks a square so that no monster or player can move on it.

There are however a great deal of crosscutting concerns that have to be added, in order to have a complete game program, that benefit from HALO's expressiveness to be implemented. For example, high-scores, computed by the number of monsters a player kills, must be kept and the scores of the different players are displayed on the computer screen of the different people playing (e.g. Player Sarge: 1000pts). However, to avoid confusion, each player should have a unique name. In order to do assure this, we need to check that when a new player is created, her chosen name is not the same as the name of a previously created player. Similarly, in order to keep the game balanced, only one of each type of player can be in a game at a time (e.g. only one "sniper", only one "soldier"). In HALO, this can be implemented quite straightforward (Figure 4.12). There are two context predicates defined to compare the names and types of players (`name-same` and `type-same` respectively). The pointcut of the advice, tries to match a player created in the past with the same name or type, using the abstract context predicate `name`: if the predicate `name` applies for a past created join point, the function `already-taken` is executing, resulting in a message to choose a new name or type by the player that is joining the game. Note that HALO frees us from iterating over all players in a game and checking the names or types manually.

```
;; CONTEXT definitions
(defrule (name-same (?Player1 ?Player2))
  (escape ?Name1 (getName ?Player1))
  (escape ?Name2 (getName ?Player2))
  (escape ?Equal (equal ?Name1 ?Name2)))

(defrule (type-same (?Player1 ?Player2))
  (escape ?Type1 (getType ?Player1))
  (escape ?Type2 (getType ?Player2))
  (escape ?Equal (equal ?Type1 ?Type2)))

(defrule (same (?Player1 ?Player2 "type"))
  (type-same ?Player1 ?Player2))

(defrule (same (?Player1 ?Player2 "name"))
  (name-same ?Player1 ?Player2))

;; ADVICE definitions
(defrule (advice "already-taken" already-taken (?Player ?Kind))
  (create Player ?Player))
  (sometime-past
    (create Player ?AnotherPlayer)
    (same (?Player ?AnotherPlayer ?Kind))))
```

Figure 4.12: HALO unique name aspect. Each player must have a unique name.

Another feature we want to implement is *auto-dynamic difficulty* [2]. The purpose of auto-dynamic difficulty is to scale difficulty by a factor of how "good" a player plays. A player plays well if she kills lots of monsters and does not get killed herself a lot. When monsters are created, their hit points and attack power are scaled by a factor equivalent to the difficulty the game is in. When a monster is killed, the difficulty level, represented by a counter, goes up by a number, which is in respect to the difficulty level the game was in when the monster was created; When the difficulty level, a counter, is raised by one whole unit, the next spawned monsters (the method `spawnMonsters` uses the method `computeDifficultyLevel` to calculate the difficulty level for creating new monsters, which returns a "floor" of the value of the variable `difficultyLevel`) are created with this new level. When a player gets killed, the difficulty levels scales back one unit. This way, the difficulty changes gradually (e.g. Figure 4.13).

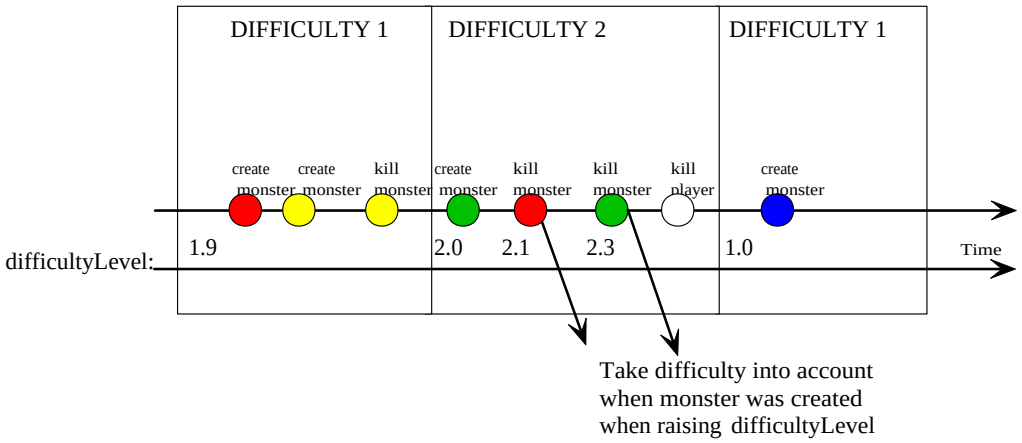

Figure 4.13: The difficulty level of the game scales by killing monsters or players dieing. The first time line depicts the "lifetime" of player and monster objects. The second time line depicts the value of the `difficultyLevel` (see class `GameField`). The circles filled with the same color represent the same objects. The difficulty level scales with 10 % of the difficulty level the monster was created in, when it gets killed. The difficulty level drops a level when a player gets killed.

Auto-dynamic difficulty can be implemented in HALO as in Figure 4.14. Please note that the HALO engine makes sure the `?Nr` value computed by the predicate `difficulty-scale`, is the value that would be gotten when evaluating the predicate when the monster is created, because when the monster gets killed, the difficulty level might have changed (context-aware!).

As a last example, we can extend the game play as follows. Say for example that we add a new kind of `FieldObject`, namely check points. The idea is that a player can only exit the game field when she has visited each checkpoint. If we add a new class `CheckPoint`, we can implement this feature in HALO (as depicted in Figure 4.15). To check if all checkpoints are taken when the player moves to the exit square, we check if there was a check point created that currently is not checked by the player (pointcut in Figure 4.15).

```
;; CONTEXT definitions
(defrule (difficulty-scale (?Monster ?Game ?Nr))
  (escape ?Game (getGameField ?Monster))
  (escape ?Level (computeDifficulty ?Game))
  (escape ?Nr (10% ?Level)))

;; ADVICE definitions
(defrule (advice "addDifficulty" add-difficulty (?Game ?Nr))
  (call "kill" (?Monster))
  (sometime-past
    (create Monster ?Monster)
    (difficulty-scale (?Monster ?Game ?Nr))))

(defrule (advice "addDifficulty" add-difficulty (?Game ?Nr))
  (call "kill" (?Person))
  (escape ?isAPerson (person-p ?Person))
  (escape ?Game (getGameField ?Person))
  (escape ?Level (getDifficultyLevel ?Game))
  (escape ?Nr (/  ?Level 100)))
```

Figure 4.14: HALO aspect for auto-dynamic difficulty.

```
;; CONTEXT definitions
(defrule (check-point-on-square (?Player ?X ?Y ?CheckPoint))
  (escape ?Game (getGameField ?Player))
  (escape ?Square (getSquare ?Game ?X ?Y))
  (escape ?CheckPoint (getOccupies ?Square))
  (escape ?CheckPoint (checkpoint-p ?CheckPoint)))

;; ADVICE definitions

(defrule (advice "markCheckPoint" mark-check-point (?Checkpoint))
  (call advance (?Player ?X ?Y))
  (checkpoint-on-square ?X ?Y ?CheckPoint))

(defrule (advice "CheckPointsNotAllTaken" check-points-not-all-taken (?Player))
  (call advance (?Player ?X ?Y))
  (exit-on-square ?X ?Y)
  (sometime-past
    (create CheckPoint ?CheckPoint))
  (escape ?Visited (visited-p ?Checkpoint)))
```

Figure 4.15: HALO aspect for marking checkpoints.

4.4 Summary

In this chapter we introduced the HALO aspect language for Common Lisp. HALO is a logic pointcut language in the spirit of CARMA, but based on temporal logic where aspects are defined in terms of logic rules. We defined a set of primitive predicates to represent join points in the execution of a program and a set of higher-order predicates, based on the temporal operators in metric temporal logic, to define a temporal relation between pointcuts. Evaluating a pointcut is done in respect to (past) program state. In fact, HALO makes it possible to reason about past and current state of a program. Furthermore, you can put constraints on variables in pointcuts about a past join point, that will be bound only later, at the occurrence of a new join point.

We concluded the chapter by illustrating the use of HALO as a means for implementing promotional aspects for an e-commerce application and some game play aspects in a video game application. We argued that the HALO language is more expressive than existing aspect languages for implementing aspects which

are based on the execution history of a program, rather than a single execution join point, and that current and past context-exposure is very declarative in HALO. In the next chapter we sketch the implementation details of HALO.

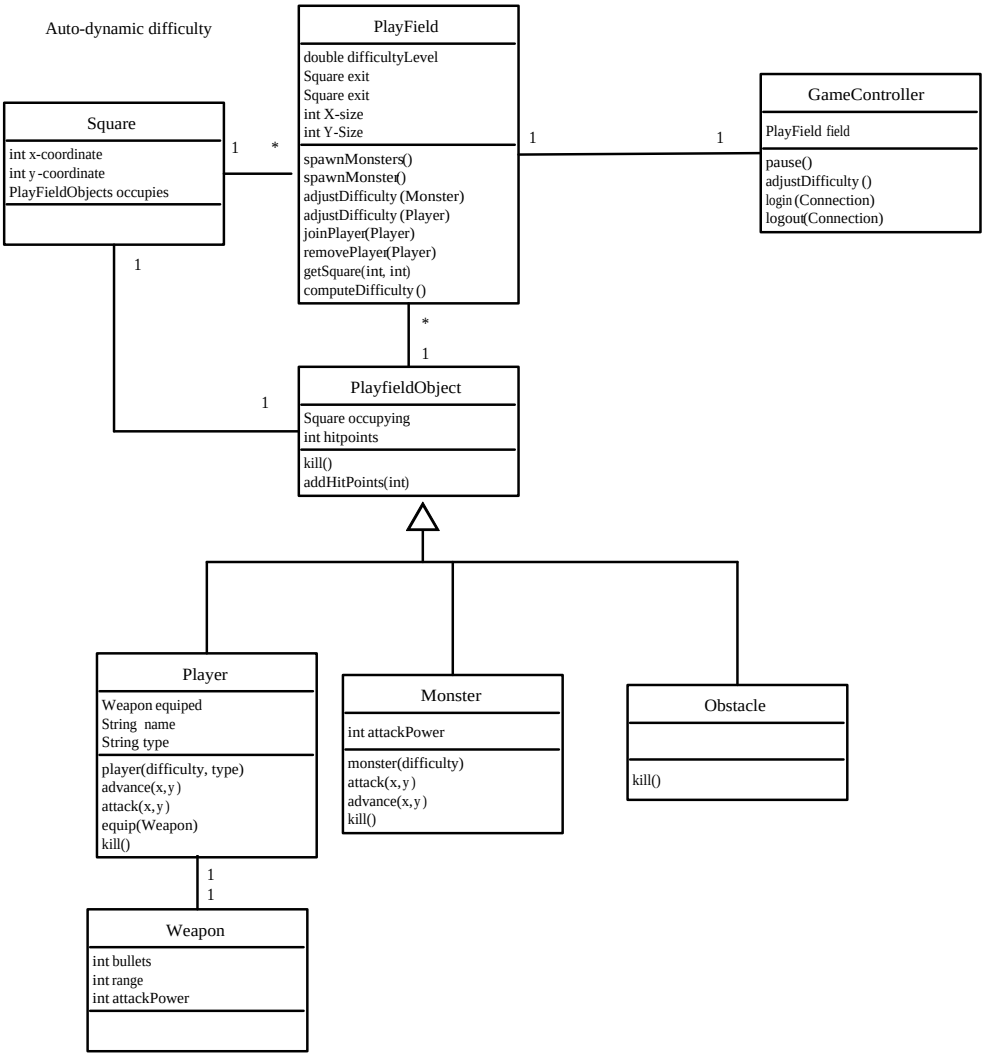

Figure 4.16: UML diagram for a shooter game.

Chapter 5

HALO implementation

In this chapter we cover the implementation of the temporal logic pointcut language, HALO, discussed in the previous chapter. In the spirit of CARMA, HALO is a logic meta programming approach to aspect-oriented programming, focusing on expressing context-aware crosscuts that depend on multiple join points (see previous chapter). The purpose of this chapter is twofold: it serves as a proof that our prototype implementation – which is a language extension for Common Lisp, written in Common Lisp – reflects the semantics of HALO described in Chapter 4 and hence that HALO *can* be implemented and secondly, this chapter serves as a blueprint for implementing a weaver for the HALO language. The structure of this chapter is as follows.

A first section sketches the general approach followed to implement HALO. Note that this is rather a summary of problems that need to be tackled, rather than a detailed description of the HALO weaver: this section is intended as high level description of the different problems that need to be tackled in order to implement the HALO weaver and as an introduction to the follow-up sections that describe the weaver in much more detail. Nevertheless, this section forms the basis for some important design issues: most importantly, we prepare the discussion on what the best resolution strategy for HALO is. As defended, we chose to use a forward chainer and discuss an implementation based on the Rete algorithm.

The subsequent section is a detailed description of the HALO weaver in terms of a compiler, responsible for translating HALO rules to a Rete network, which implements the HALO logic repository, and an interpreter responsible for handling the querying for applicable advices. Please note that this is not a straightforward compilation and interpretation process. The HALO language is more complex than the subset of Prolog for which Rete was originally designed. In fact we need to extend the original Rete algorithm by introducing new types of nodes and we propose a more fine-grained compiler, in order to be able to reflect the HALO semantics in the interpreter. Nevertheless, the pseudo code in this section makes it possible to straightforwardly recreate our solution.

Finally, in a last section, we present a summary of this chapter and we outline future work.

5.1 General idea

We need to make sure that a base program and a HALO program are combined: we need to build a weaver for HALO. The general idea is as follows. We have a logic repository to store facts, rules and aspects (where an aspect is a set of rules that extend the definition of the predicate `advice`). During program execution, the execution history (= a sequence of join points) is reified to a stream of logic facts and at the occurrence of a new event (= join point) at runtime, this trace is scanned and matched against the bodies of the aspect rules in the rule repository – or in other words, the pointcuts of the advices of the aspects–; Successful matches return the head of these rules. These instantiated patterns, representing advices, are reified to execution, meaning the piece of code that implements the advice is executed and hence is inserted in the execution history. Figure 5.1 is a graphical representation of this idea. From this high level description and the HALO semantics described in the previous chapter, we anticipate that there are quite a few problems we need to tackle.

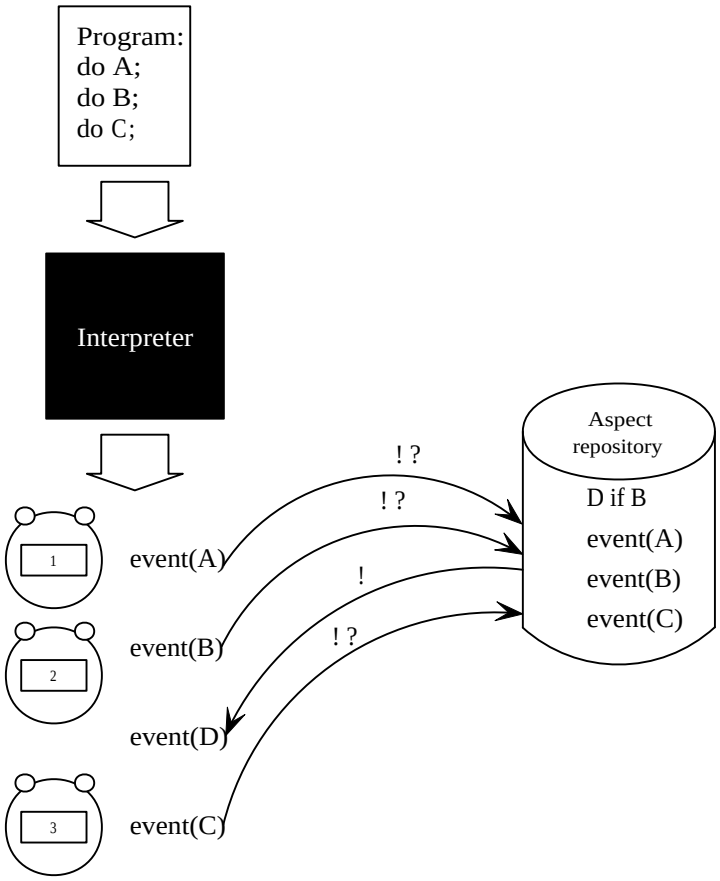

Figure 5.1: HALO implementation: general idea.

In order to implement a working HALO weaver, we need to address the

following problems. A first problem we need to tackle, is how do we generate/advise join points or in other words how do we make sure that each join point is saved as a fact and applicable advices are executed? Second, we need to implement a query interpreter for the pointcut language, namely we want to be able to evaluate pointcuts on a join point reflecting the HALO semantics defined in the previous chapter; As HALO is based on Prolog (for which a great deal of interpreters exist), the main problem we face is implementing the semantics of the temporal operators. Another issue we need to address is the fact that an object's state changes during program execution, but if you have a rule that poses some requirement on an object in the past (ergo when it is constrained under the influence of a temporal operator), we need to make sure, that whenever this rule is evaluated, we evaluate it in relation to the past state of the object. Finally, we need some form of garbage collection for facts, as representing *all* join points as facts and saving them *all* in a logic repository forever, is just begging for memory overflow.

In the subsequent sections, we describe how these problems affect the implementation and we outline the possible solutions.

5.1.1 Generating and advising join points

On each method call or instance creation, a fact has to be recorded in the logic repository so that pointcuts can reason about it in the future. This is achieved by use of the CLOS MOP [24], which allows extending the default CLOS semantics. Facts about such events are stored in the form `(call T ...)` or `(create T ...)` where T stands for a time stamp which is generated by a global event counter. The default CLOS semantics are also extended to include queries of the logic repository for applicable advices and their possible execution.

5.1.2 HALO interpreter

The logic repository consists of a set of HALO rules and facts. At each join point, the logic repository is queried for applicable advices. We chose to use a Prolog interpreter to implement the HALO interpreter, as HALO = Prolog + temporal operators. Before a rule is added to the logic repository, the rule is compiled to Prolog analogous to the translation from metric temporal logic to first order logic defined in [9] (see Chapter 3).

The basic idea is as follows. Each predicate gets an extra argument, namely a temporal variable, that keeps a spot to store the time stamp to which a formula is evaluated. The temporal operators are compiled away by putting constraints on this argument. As an example, consider the following formulas compiled to Prolog.

```
MTL

1. a(A) :- sometime-past(b(B))
2. a(A) :- previous(b(B))
3. a(A) :- sometime-interval(x, y, b(B))

Prolog

1. a(T1 A) :- findall(T, and(b(T2, B), greater(T1, T)), Opl),
              largest(Time, Opl),
              b(Time, B).
2. a(T1 A) :- b(T2, B), T2 is T1 - 1
```

```
3. a(T1 A) :- findall(T, and(b(T2, B),
                             greater(T1, T),
                             greater(T, T1 + x),
                             greater(T1 + y, T)),
                      Opl),
              largest(Time, Opl),
              b(Time, B).
```

Now, if we want to evaluate a query `a(A)`, this query itself is translated to a pattern that includes a time stamp, namely `a(now, A)` (the value of `now` is the value of the global event counter (= nr of events that occurred so far, see previous section) and evaluated in relation of the logic repository containing the compiled HALO rules.

Note the semantics of the `sometime-past` operator. Namely, we want a rule containing a `sometime-past` to evaluate to true, only for the most recent event that makes it true. For example, if we have a logic repository containing the following facts and rule:

```
(call 1 "login" bob)
(call 15 "login" bob)
(advice "already authenticated" print
  (?User "you have already authenticated yourself") if
  (call "authenticate" ?User)
  (sometime-past (call "login" ?User))
```

and we query `(advice "already logged in" log (?Name))`, we want this query to return `(advice "already logged in" log (bob))` once (for the fact `(call 15 "login" bob)`). In the examples we showed before, we compiled away the `sometime-past` using a `forall` (line nr. 1) in order to get this semantics. However, we can optimize this, if we use a Prolog "cut". E.g. as in :

```
((advice "addDifficulty" add-difficulty (?Game ?Nr))
  (call "kill" (?Monster))
  (sometime-past
  (create Monster ?Monster)
    (difficulty-scale (?Monster ?Game ?Nr)))))
```

becomes

```
advice(?T, "addDifficulty", add-difficulty, [?Game, ?Nr]) :-
  call(?T, kill, [?Monster]),
  create(?U, Monster , ?Monster),
  ?U < ?T,
  difficulty-scale(?Monster, ?Game, ?Nr),
  !.
```

We define an algorithm for compiling HALO to Prolog as follows. Compiling a HALO rule, results in compiling the head of the rule and the body of the rule, and creating a Prolog rule from the compiled head and the compiled body (see pseudo code in Figure 5.2). Compiling the body of a clause (see pseudo code in Figure 5.3), results in compiling each pattern in the conjunction and joining each such compiled pattern, using a $\wedge$ (in pseudo code, stored in variable `result`). If the pattern being compiled is an atom, the temporal variable is put as an

extra argument in the atom. A new temporal variable is generated each time a pattern with one of the temporal operators as predicate name is encountered, and an appropriate constraint is added to the result.

```
result = nil.
def compile-rule(rule):-
  temporal-var = gensym()
  compile-to-prolog(temporal-var, head(rule))
  head = result
  result = nil
  compile-to-prolog(temporal-var, body(rule))
  body = result
  result = nil
  return rule(head, body)
```

Figure 5.2: Pseudo code for compiling restricted MTL clause to Prolog.

```
def compile-to-prolog(temporal-var, conjunction):
  pattern = first(conjunction)
  if sometime-past-p(pattern):
       new-temporal-var = gensym()
       compile-to-prolog(new-temporal-var, arguments(pattern))
       result and= (< new-temporal-var temporal-var)
       result and= cut
  elif previous-p(pattern):
       new-temporal-var = gensym()
       compile-to-prolog(new-temporal-var, arguments(pattern))
       result and= (= new-temporal-var (- temporal-var 1))
  elif sometime-interval-p(pattern):
       new-temporal-var = gensym()
       left = left-interval(pattern)
       right = right-interval(pattern)
       compile-to-prolog(new-temporal-var, arguments(pattern))
       result and= (> new-temporal-var (+ left temporal-var))
       result and= (< new-temporal-var (+ right temporal-var))
       result and= cut
  else:
       pattern = add-argument(pattern, temporal-var)
       result and= pattern
  compile-to-prolog(temporal-var, rest(conjunction))
```

Figure 5.3: Pseudo code for compiling restricted MTL conjunction to Prolog.

5.1.3 Saving object State

The HALO rules reason about program execution and the objects involved in the execution. However, an object's state changes during program execution, as fields get set. When one writes a rule about an object in the past, the HALO semantics is such, that whenever the rule is evaluated, the rule is evaluated in relation to the `past` state of the object. Take for example the rule:

```
(defrule (advice "discount" discount (?User 0.10))
  (call "checkout" (?User))
  (sometime-past
    (call "login" (?User))
    (promo-active (?User))))
```

When we evaluate a query `(advice "discount" (lotte t-shirt 0.10))`, we need to evaluate the body of the rule for truth, so:

```
1.  (call "checkout" (lotte))
2.  (sometime-past
3.    (call "login" (lotte))
4.    (promo-active (t-shirt))))
```

When we evaluate the conditions under the `sometime-past` operator (line nr. 2-4), this needs to be done in respect to the state the object `lotte` had at the time the `(call "login" (lotte))` event was recorded. Of course, this is the responsibility of the HALO interpreter.

If we choose a backward chainer as a basis for the HALO interpreter, we need to make sure we make a (deep) copy of the arguments of a recorded event when the event happens. When a query is evaluated at a later point, this deep copy can be used to evaluate constraints. So in the same example, when the fact `(call "login" lotte)` is recorded, make a copy of the object `lotte` and when line nr. 2-4 is evaluated, do so using this copy.

However, if we choose a forward chainer as a basis for the HALO interpreter, we can evaluate the constraints as events are generated. E.g. in the same example, when the event `(call "login" lotte happens)`, evaluate the condition on line nr. 4. So there is no need to take deep copies of the objects involved in the event. Alas, not all conditions can be pre-evaluated.

There are rules that put constraints on variables used inside a temporal operator that are not always bound when the event happens. For example, take aspect 3 (Figure 4.10) of the previous chapter. It states that a user gets a discount on an article if there was a stock overflow for that particular article at the time the user logged in. The problem is that when the login event happens, we cannot possibly know at that time what article the user will buy. So we cannot compute the result of the `promo-activated` condition. This variable is bound when the buy event takes place. However at that time we must evaluate the promo-activated condition in respect to the state of the shop when the user logged in.

Therefore, whenever a method call is recorded with such advices, as can be derived from the rule definitions, we must take a deep-copy from the arguments and this copy is used to evaluate the promo-activated condition later on. This is similar to the notion of taking "snapshots" of the system state in the Reflex-based extension described in Section 2.4.4.

5.1.4 Garbage Collection

It is not very economic to store each method call and instance creation (as a fact) forever because it is very likely that the systems runs out of memory as many methods are called during program execution. However, we observe that there is some optimization possible due the semantics of the temporal operators. Some facts become obsolete as new facts are inserted, because they are "replaced" by these facts in order to resolve a query. This knowledge can be used to build a garbage collector.

The general idea behind the garbage collector is as follows. The garbage collector iterates over the facts present in the logic repository and considers a fact garbage if deleting the fact has absolutely no effect on the result of any query. Of course, we need some guidelines to decide whether a fact is garbage.

We can derive from the rules whether a certain fact can be needed to resolve

a query. If we take one rule, and the fact can be unified with one of the patterns in that rule, the fact is possibly needed to resolve a query. E.g. take the rule `(A ?x) :- (call B ?x), (> ?x 1)`, then the fact `(call B 5)` can possibly be needed to resolve a query `(A ?x)`, because it unifies with the pattern `(call B ?x)`. Now, depending on whether the pattern is the argument of a (temporal) operator and the presence of many unifying facts, we can decide if a fact becomes garbage when there exists another fact that can be used to resolve a query.

no temporal operator

Given a rule R and a fact F that unifies with a pattern P present in the body of a rule, which isn't the argument of a temporal operator, we consider the fact F garbage for the rule R, at a time point t, if the time stamp of the F is smaller than t. E.g. consider a fact `(call B 5)` and the rule R:

```
(advice ?a) :- (call B ?b), (> ?b 2)
```

There is no past reference to a fact of form `(call B ?b)` needed to resolve a query `(advice ?a)`.

sometime-past

Given a rule R and a fact F that unifies with a pattern P present in the body of a rule, as an argument to the `sometime-past` operator, and that `sometime-past` form is not an argument of another temporal operator (0), we consider the fact F garbage for the rule R, if:

1. the query Q resulting from unifying F with the argument list of the `sometime-past` operator cannot be resolved (1)

2. there exists a fact F', with $timestamp(F') > timestamp(F)$ and $target(F) = target(F')$ so that resolving the query Q yields the same result as resolving the query Q' resulting from unifying F' with the argument list of the `sometime-past` operator (2)

where we define $timestamp$: $Fact \rightarrow Number$, a function that returns the time stamp of a fact and $target$: $Fact \rightarrow Object$, a function that returns the first argument in the argument list of F. E.g. if $F =$ `(call 1 "login" (*lotte*))`, then $timestamp(F) = 1$ and $target(F) =$ `*lotte*`.

If (1) is true, it is obvious that the fact is garbage for the rule, because it cannot be used to resolve the rule. If (2) is true, it means that only F' will ever be used to resolve the rule as the semantics of the `sometime-past` operator is such that there must exist only one fact such that a query `(sp ...)` is true and we defined this to be the most recent of all those facts.

E.g. if we have a rule and execution history like:

```
(defrule
  (advice "buy-one-get-one-free" book-for-free (?User ?Article))
  (call "checkout" (?User))
  (sometime-past (call "buy" (?User ?Article))
                 (t-shirt-p ?Article)))

1. (buy *lotte* *book*)
2. (buy *lotte* *t-shirt*) ;; assume that *t-shirt* is a T-shirt
3. (buy *lotte* *cd*)
4. (buy *lotte* *t-shirt*)
```

Then, at time 4, we consider fact nr. 1 and 3. garbage because the query $Q =$ `((call "buy" *lotte* *book*)(t-shirt-p *book*))`, given fact nr. 1 fails (1) – and a similar query for fact nr. 3 fails. Because of (2), we also consider fact nr. 2 garbage, because given fact nr. 4, which has a larger time stamp, the query $Q' =$ `((call "buy" *lotte* *t-shirt*)(t-shirt-p *t-shirt*))` succeeds.

However, in order to resolve a query Q, there can be no variable unbound in Q, that is bound outside Q in the rule R, because this means, we cannot compute Q, because the value for such a variable is not fixed. Take for example the rule:

```
(defrule
  (advice "smth" smth (?a))
  ((call c ?b) (sometime-past (call B ?b) (> ?b ?a)))
```

We cannot determine which `(call B ?b)` events fulfill the `(> ?b ?a)` constraint because the value of `?a` is not fixed. Therefore, we cannot decide when a fact that unifies with `(call B ?b)` is obsolete. In this case, no memory management is possible.

Some explanation is needed why we require (0) to be true. If (0) isn't true, this means that the pattern P is an argument of a `sometime-past` operator that is itself an argument of a temporal operator. E.g. as in the rule:

```
(defrule
  (advice "buy-one-get-one-free" book-for-free (?User ?Article))
  (call "checkout" (?User))
  (sometime-past (call "buy" (?User ?Article))
                 (t-shirt-p ?Article)
                 (sometime-past (call "login" (?User))
                                (t-shirt-promo (?User)))))

0. (login *lotte*)
1. (buy *lotte* *book*)
2. (buy *lotte* *t-shirt*) ;; assume that *t-shirt* is a T-shirt
3. (buy *lotte* *cd*)
4. (buy *lotte* *t-shirt*)
```

For this rule, the garbage collection will bailout for the fact `(call "login" *lotte*)`. Though this doesn't really matter for this example, the problem is that, in general, when (0) is not true, we cannot throw away F if (1) or (2) is true, because Q is relative to the occurrence of an event, that might still happen. This is different from when (0) is true, because then Q is relative to "now". E.g. given some morbid rule:

```
(defrule
  (...)
  ((call (a ?Obj))
   (sometime-past ((call b (?Obj ?Nr))
                   (> ?Nr 5)
                   (sometime-past (call c (?Obj))))))

1. (call c o) ;; still needed though (2) is true
2. (call b o 7)
3. (call a o)
4. (call c o)
5. (call b 3 o)
6. (call a o)
```

We can't throw away fact nr. 1 (as (2) says because of fact nr. 4), because Q is relative to the occurrence of a fact that can unify with `(call b (?Obj ?Nr))`,

so at time 7 fact nr. 1 is the most recent fact that can fulfill the conditions of the rule, however if at time 7 for example a fact (call b o 9) happens, then fact nr. 4 will be the most recent fact that can possibly resolve the query at a time 8. In general we can't decide if a fact that unifies with `(call c (?Obj))` will *ever* be used to resolve the entire rule, though we can say this about a fact that unifies with `(call b (?Obj ?Nr))`, given (1) or (2). [1]

One might consider to be a rather harsh restriction, but we observe that most of the examples used to illustrate HALO in the previous chapter, do not break (0), so in practice, the garbage collection rule described in this section, still seems to be *effective*.

previous

Given a rule R and a fact F that unifies with a pattern P present in the body of a rule, as an argument to the `previous` operator, we consider the fact F garbage for the rule R, if there exists no fact F' such that $timestamp(F) = timestamp(F') - 1$ where F' unifies with a pattern P' that represents the event which F precedes, unless $timestamp(F)$ equals the current time. E.g. if one considers the rule R and execution history as given below, then fact nr. 3 is garbage.

```
(defrule
  (...)
  ((call (a ?Obj))
   (previous (call b (?Obj)))))

1. (call b o)
2. (call a o)
3. (call b o)
4. (call c 0)
```

We can now define the garbage collector in terms of these rules. A fact is garbage for a rule, if any of the previously described rules applies for the rule and the fact; A fact is garbage if it is garbage for all the rules in the logic repository. Figure 5.4 depicts the pseudo code for a garbage collector. Note that the predicates `no-operator-p`, `sometime-past-p`, `sometime-interval-p` and `previous-p` are defined in terms of the descriptions per temporal operator given earlier in this section.

In the rest of this chapter we discuss an implementation of HALO that takes the problems observed in this section into account.

5.2 Resolution strategy for HALO

The HALO implementation needs a logic interpreter to handle the querying for applicable aspect rules. An important decision to make when designing this interpreter is whether to opt for a *forward chainer* or a *backward chainer* as resolution strategy. We remind the reader of the definitions.

The definition for forward chaining is as follows (from [34])[2]:

[1] There is room for some improvement here. We can for example change (2) to say that if (0) is true, that F becomes garbage if the query resulting from unifying F with the argument list of the "highest" temporal operator, is true for an F'. The reason we don't do that for now, is because then we can't combine the garbage collection with the insertion of a new fact, as is currently opted in the Rete implementation.

[2] Beware: AI terminology

```
garbage-collect():-
  FOR fact IN *logic-repository*
  DO
    IF garbage-p (fact, *rules*)
    THEN remove (fact, *logic-repository*).

garbage-p (fact, rules):-
  garbage = true
  index = size(rules)
  WHILE (garbage AND index > 0)
  DO
    rule = rules[index]
    garbage = (no-operator-p(fact, rule) AND
               sometime-past-p(fact, rule) AND
               sometime-interval-p(fact, rule) AND
               previous-p(fact, rule)
  RETURN garbage
```

Figure 5.4: HALO garbage collection.

> Forward chaining is an example of the general concept of *data driven* reasoning – that is, reasoning in which the focus of attention starts with the known data. It can be used within an agent to derive conclusions from incoming percepts, often without a specific query in mind.

Conversely, the definition for backward chaining is as follows (from [34]):

> Backward chaining is a form of *goal-directed reasoning*. It starts with a list of goals and works backward to see if there are data available that will support any of these goals. It is useful for answering specific questions.

In general, forward chaining is preferred over backward chaining when it is not known beforehand what goals need to be proved, because a forward chainer checks each rule for applicability when a fact is added to the logic repository. If on the other hand, it is known beforehand what goals need to be proved, backward chaining is preferred.

In section 5.1, we said that a fact is added to the logic repository for each join point in the execution of a program and at each join point, the logic repository is queried for any possible advice – we don't know which sort of advice of which aspect – to execute, given this new join point fact. This description matches closely to the definition of a forward chainer, where "an agent derives conclusions from incoming percepts". A second reason for opting for a forward chainer involves section 5.1.3. Using a forward chaining approach, requires us to make less deep copies of objects (if any at all), as conditions on (past) events can be immediately evaluated when an event is stored in the logic repository.

5.3 Rete

A naive forward chainer might check each rule against the known facts, when a new fact is added to the logic repository, deriving a conclusion if possible, then moving on to the next rule and looping back to the first rule when finished, until no more conclusions can be derived. Even for a logic repository with a moderate size, this naive approach performs far too slowly. As HALO is likely

to run with many facts and rules, we want a more efficient forward chainer for HALO. We next take a look at the Rete algorithm. The Rete algorithm [14] is a *fast* forward chaining algorithm for *production systems*.

A production system is a formalism for representing knowledge under the form of *condition-action rules* or *productions* where the body of the rule represents the *conditions* and the head of a rule represents the *action*. When the conditions of a certain rule are met, by the presence of certain facts in the logic repository, the rule *fires* which means the rules' action is executed.

The general idea behind the Rete algorithm is to represent the knowledge base, consisting of productions and facts by means of a network of nodes. The productions in the knowledge base are compiled to a network of *filter*, *join* and *production* nodes and each of these nodes has a *memory table*, which are used to store the different facts.

The compilation of a single rule will result in the following network. The patterns in the body of the production are all compiled to (filter) nodes and these nodes are one by one connected to one and other by means of a join node. The join node takes two nodes as input nodes and one node as output node. Finally, the head of the rule is compiled to a production node which is then placed as the output node of the last created join node. Each filter node has a label, which is the predicate name of the pattern it represents; In addition, each filter node has a memory table associated with it, which is a mapping for the logic variables and constants in the pattern they represent. Each join node contains a memory table which is a mapping for the union of the logic variables of the memory tables of its two input nodes. The filter nodes are connected as outputs to a special node, called the root node. For example if we compile the production:

```
IF flies(X) AND feathered(X) AND lays_eggs(X) THEN (add bird X)
```

This rule states something like: if something flies, has feathers and lays eggs, then that something is a bird. Compiling this rule, results in the network depicted in Figure 5.5. The ellipsis represents the root node, while the circles represent filter nodes, the squares represent join nodes and the triangle represents a production node. The tables next to the nodes are the memory tables associated with the nodes.

Storing facts in now done by means of the memory tables in the nodes. To store a fact, the fact is *inserted* in the network, which results in passing the fact from one node to another, according to the following rules, depending on the type of node the fact is inserted in:

root node : insert the pattern into all the filter nodes, which are the output nodes of the root node.

filter node : when the predicate name of the fact matches the label of the filter node, try to map –by means of unification (see Chapter 2)– the arguments of the fact to the variables and constants of memory table of the filter node and insert the fact in the output node of the filter node.

join node : note the node from which the fact is inserted, combine the fact with all the entries in the memory table of the other input node of the join node and insert the obtained tuples into the memory table of the join node, if there are no variable clashes (when the input nodes of the

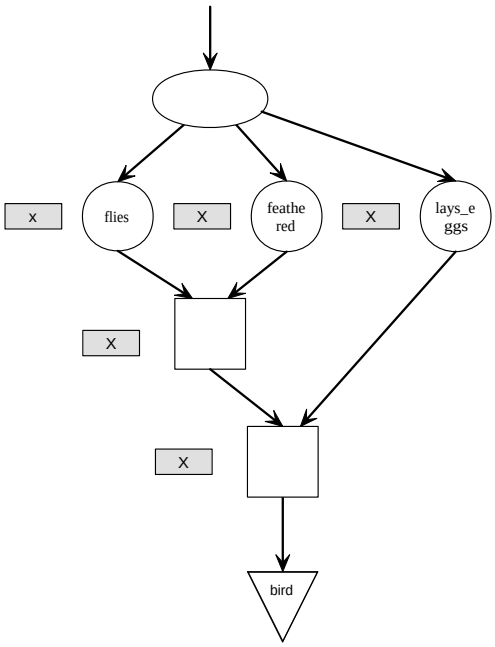

Figure 5.5: Rete network for production `IF flies(X) AND feathered(X) AND lays_eggs(X) THEN (add bird X)`.

join node represent patterns with variables with the same name, and the combination tries to bind that same variable to different values).

production node : fire the rule, execute the actions associated with the production node.

For example if we insert a fact `(flies daffy)`, in the network depicted in Figure 5.5, this affects the memory tables as shown in Figure 5.6. The fact's predicate matches the label of the filter node labeled `flies` and consequently an entry `daffy` is made for the variable `X` in the node's memory table; The fact is passed along the join node connected to that filter node, but as no entries are made in the memory table of the node labeled `feathered`), nothing happens. Now, if we next insert the fact `(feathered daffy)`, this results in an entry for the variable `X` in the memory table of the filter node labeled `feathered`, and again, the fact is passed along to the join node connected to that node. However, now there is an entry in the memory table of the join node's other input node (the node labeled `flies`) and the join node tries to combine the entries, which succeeds as the common variables in the memory table's headers are assigned the same values, namely the variable `X` has the value `daffy` for both entries we are trying to combine, and hence an entry is saved in the join node's memory table and past along the output node of the join node (if we had inserted a fact `(feathered donald)`, this would simply have resulted in an entry in the memory table of the filter node, labeled `flies`). If we next insert the fact `(lays_eggs daffy)`, the rule fires and in this case, this results in inserting the fact `(bird daffy)` in the network.

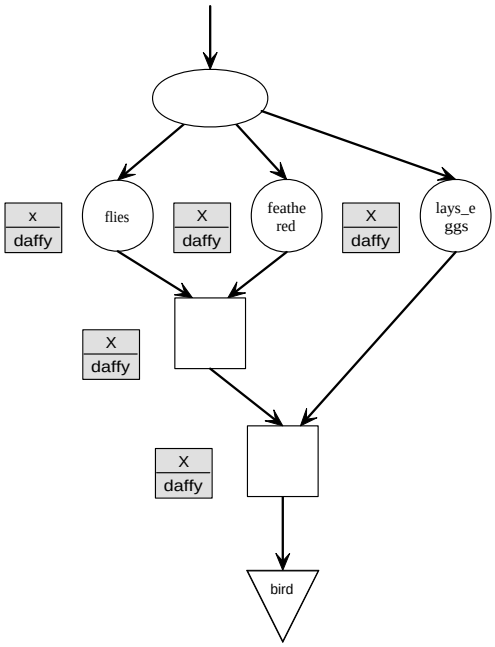

Figure 5.6: Rete from Figure 5.5 after inserting the facts `(flies daffy)`, `(feathered daffy)` and `(lays_eggs daffy)`.

In the next section we show how the Rete algorithm can be extended – by introducing new types of nodes with their own insertion rules, reflecting HALO's semantics – so that the HALO rules can be compiled to Rete networks.

5.4 Extending Rete

The Rete algorithm was not originally designed with higher-order predicates in mind, see for example [14] where the `not` predicate is treated as a special case, implemented by a join node with special behavior for inserting a fact. We take the same approach for the different temporal operators. We introduce new types of join nodes to implement the temporal connectives, meaning that the behavior for inserting a pattern in these nodes will be different than for inserting a pattern in a regular join node. Next to new join nodes, we introduce new filter nodes to implement special built-in predicates such as the `escape` predicate.

One might wonder – as we can compile the rules to Prolog, which removes the higher-order predicates (see Section 5.4.2)– why we don't just use the regular Rete algorithm and why we need to introduce new types of nodes. Though the translation transforms the rules into Prolog, this doesn't assure the HALO semantics completely (see state issue described in Section 5.1.3) and consequently, as is explained in the subsequent sections, we need to introduce some new nodes so that the HALO-specific semantics are implemented. Another reason for implementing our own nodes, is that this leaves room for some optimizations.

In the next subsection we discuss the compilation of HALO rules to Rete

networks in full detail. Basically, compiling a rule goes as follows. The body of the rule is compiled into the Rete network by compiling each pattern to a filter node. Depending on the type of pattern – determined by its predicate name, e.g. `escape`, `call`, ... – a different type of node is used. In section 5.4.1 we present an overview of the different types of nodes used to compile a pattern to a node. Once a pattern is compiled, the resulting node is connected by means of a join node, to another node. Again, the type of the nodes that are being connected matters as this determines the type of join node being used. In section 5.4.1 we discuss the different types of join nodes.

In the subsequent subsection, we discuss the interpreter, which is defined in terms of the behavior of inserting facts in a Rete network. We remember that inserting a fact in a Rete network results in passing around the fact to the different nodes in the network. Depending on the type of node we insert a fact in, different actions are performed. A detailed overview of behavior per node is given in 5.4.2.

A third section explains how we can combine the garbage collector discussed in Section 5.1.4 by combining the behavior with the insertion mechanism behind the interpreter.

The last subsection discusses some specific implementation problems concerning rule definitions and garbage collection we deliberately avoid in section 5.4.1 and 5.4.2 to obtain lesser complicated subsections.

5.4.1 Compiler

The general idea is as follows. Say we have some global variable `*rules*` that stores a list of all the HALO rules added via the macro `defrule` (see chapter 4). Then in order to get a Rete that represents this set of rules, we compile them all. In pseudo code, the compiler looks something like Figure 5.7. The compiler just loops through the different rules and compiles them one by one via the function `compile`.

```
def compile-rules():
  for rule in *rules*
    compile(rule).
```

Figure 5.7: Pseudo code for compiling a set of HALO rules.

In order to compile a single rule, we first compile the body of the rule, which is a conjunction of patterns. This compilation returns the last (join) node that was created during this compilation and to this node we connect a production node, representing the head of the rule. The pseudo code for compiling a single rule is shown if Figure 5.8.

```
def compile-rule(rule):
  body = body(rule)
  head = head(rule)
  node = compile-conjunction(0, body)
  production-node = production-node(head)
  output(node) = production-node
  input(production-node) = node
```

Figure 5.8: Pseudo code for compiling a single HALO rule.

Compiling a conjunction goes as follows. We compile the first two patterns in the conjunction, which we then join using a join node; Then we grab hold of the next pattern in the conjunction and compile it and we join the resulting node to the previously created join node. We repeat this until no more patterns are left in the conjunction to compile. Pseudo code can be found in Figure 5.9. Note that the function `compile-conjunction` takes a mystery parameter `level` – more on this later.

```
def compile-conjunction(level, conjunction):
  rest-to-compile = conjunction
  first = first(conjunction)
  second = second(conjunction)
  fnode = compile-pattern(level, first)
  snode = compile-pattern(level, second)

  fnode = join(level, fnode, snode, pattern-type(second)

  while (length(rest-to-compile) > 0)
  do
    snode = compile-pattern(level, first(conjunction))
    fnode = join(level, fnode, snode, pattern-type(first(conjunction)))
    rest-to-compile = rest(rest-to-compile)
  return
    /* return the last created join node */
    fnode
```

Figure 5.9: Pseudo code for compiling a conjunction of HALO patterns.

Compiling a pattern always returns a node; Depending on the "type" of pattern being compiled, a different kind of node is returned. There are 7 different patterns we distinguish, depending if the predicate name of the pattern is `sometime-past`, `previous`, `sometime-interval` (one of the temporal operators), `escape`, `call`, `create` or some random name. As can be seen in the pseudo code (Figure 5.13), depending on the type of the pattern, a different compilation strategy is followed:

1. Event patterns

Remember that event patterns are patterns that reflect the events in program execution, such as method calls and instance creations (see chapter 4). A *call pattern* is of the form `(call ?fname (?arg1 ... ?argn))` and a *create pattern* is of the form `(create ?class ?instance)`. The compilation of such a pattern results in a filter node. The label the node gets is either `call` or `create`; The memory table's header consists of the arguments of the patterns + a newly generated variable representing a temporal variable (this keeps a spot to store a time stamp (see section 5.4.2)). The filter node is connected to the root node (e.g. Figure 5.10).

2. Escape patterns

Remember that an *escape pattern* is of the form `(escape ?x lisp-form)` and that the idea is that a lisp-form is evaluated at the Lisp level and that the logic variable is bound to the result. Hence an escape pattern is not the sort of pattern for which facts will directly be inserted in the logic repository, therefore an escape pattern is not simply compiled to a regular filter node. In fact, an

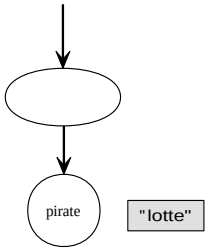

Figure 5.10: Rete for the pattern `(call "login" (?User))` The table header contains a temporal variable `T`, a constant `"login"` and a variable `User`.

escape pattern is compiled to a special filter node. It contains the lisp-form and a memory table with as header the logic variable. This filter node is *not* connected to the root node (e.g. Figure 5.11) – it is connected as a one input join node to another node (see below).

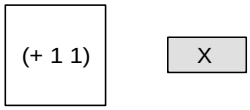

Figure 5.11: Rete for the pattern `(escape ?X (+ 1 1))`.

3. Patterns where the predicate name is a temporal operator (or `not`)

The argument of a pattern where the predicate name is a temporal operator (`sometime-past`, `sometime-interval`, `previous`) or `not`, is a conjunction of patterns, and consequently compiling such a pattern requires to compile this conjunction (Figure 5.9) ((e.g. Figure 5.10 can be the result of compiling the pattern `(sometime-past ((call "login" (?User))))`).

4. Pattern with random predicate name

A pattern with a predicate name that is not one of the event predicates, escape predicate or temporal operators, is compiled to a plain filter node. The label is the pattern's predicate name and the the memory table's header is the argument list of the pattern. The filter node is connected to the root node (e.g. Figure 5.12).

Now that we know how to compile patterns to nodes, we need to join them (see use of function `join` in Figure 5.9 and see Figure 5.17). Joining two nodes means they are connected to one another using a join node. Depending on the "type" of the second node [3], a different kind of join node is used (Figure 5.17):

[3]The "type" of the second node = the predicate name of the pattern it represents.

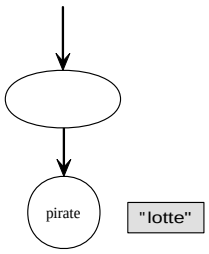

Figure 5.12: Rete for the pattern `(pirate "lotte")`.

```
def compile-pattern(level, pattern):
  type = pattern-type(pattern)
  case
   sometime-past-pattern?(type) then return compile-conjunction(level+1, arguments(pattern))
   sometime-interval-pattern?(type) then return compile-conjunction(level+1, arguments(pattern))
   previous-pattern?(type) then return compile-conjunction(level+1, arguments(pattern))
   escape-pattern?(type) then
     return escape-filter-node(lambda=lambda(pattern),memory-table=var(pattern))
   create/escape?(type)
      node = filter-node(label=predicate(pattern),
                         memory-table=union((generate-temporal-variable), arg(pattern)))
      *root* add= node
      return node
  else /* a regular pattern*/
      node = filter-node(label=predicate(pattern), memory-table=arg(pattern))
      *root* add= node
      return node
```

Figure 5.13: Pseudo code for compiling a single HALO pattern.

1. Joining a node and an escape filter node

There are two cases we need to consider: (1) there are *no* variables used in the escape filter nodes' lisp-form that are not matched by a variable in the header of the first node or (2) there are variables used in the escape filter nodes' lisp-form that are not matched by a variable in the header of the first node.

When the first case applies, we put the escape filter node as output of the other node and we update the header of the escape filter node by adding the header of the memory table of the first node (e.g. Figure 5.14).

When the second case applies, we can't just put the escape filter node as output of the other node, because the escape filter nodes' lisp form cannot be evaluated when there are unbound logic variables (see section 5.1.3). Instead, we put a special type of node as output of the first node, namely a "copy node", which has the same memory table of the first node (e.g. Figure 5.15). The escape filter node itself is put away and will be joined to another node at a later point in the compilation (see pseudo code in Figure 5.16).

2. Joining a node and node, where the second node represents a pattern with predicate name = temporal operator

The general idea is that for each temporal operator, we have a special join node. For example, if we join a node with a node that was the result of compiling a pattern with predicate name = `sometime-past`, we get the following network.

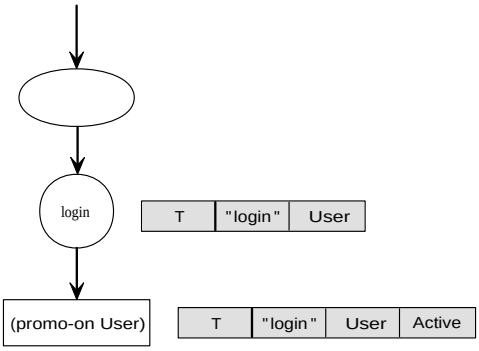

Figure 5.14: Rete for the conjunction `((call "login" "User") (escape ?Active (promo-on ?User)))`.

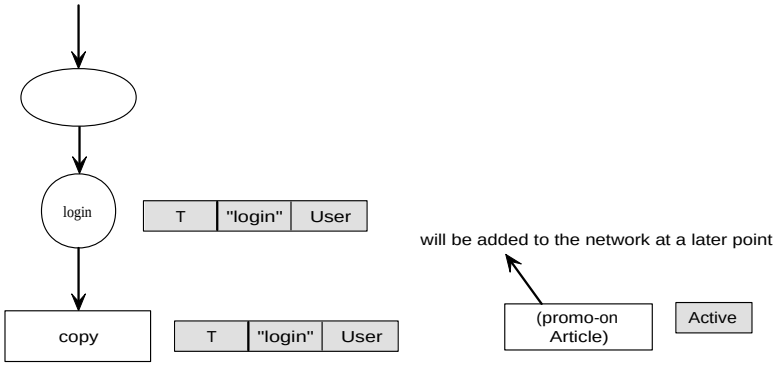

Figure 5.15: Rete for the conjunction `((call "login" "User") (escape ?Active (promo-on ?Article))`.

We create a join node of type "sometime-past" and its memory table's header is the result of taking the union of both the nodes that are being joined their memory table. In addition we join the resulting join node with an escape pattern of the form `(< T K)` where `T` = the temporal variable of the (most left) filter node connected to the first node and `K` = the temporal variable of the (most left) filter node connected to the second node. Next to that, we try to join some of the escape filter nodes that couldn't be joined previously (e.g. as in 5.4.1). The pseudo code is depicted in Figure 5.18 and an example in Figure 5.19.

Similarly, a special node is used to join two nodes, where the second node was the result of compiling a pattern with predicate name = `previous` or `sometime-interval` and similarly the resulting join node is joined with an escape pattern of the form `(= K (- T 1)` and `((> K (+ T left-interval)) (< K (+ T right-interval))` respectively.

```
def join-escape(level, left, right):
  jnode = left
  if(there are free variables in the action of the right
     node (= escape node) that are not present in the
     variable list of the memory table of the left node)
  then /* add a promis to join the right node later in
       compilation phase */
     *map-of-nodes-to-be-added* join= (level, right)
     /* make a node to take a copy of variables that are bound in left*/
    a-copy-node(variables(memory-table(left)))
    input(a-copy-node) = left
    output(left) = a-copy-node
  else
    input(right) = left
    output(left) = right
    /* update memory table of right to contain header of left */
    header(memory-table(right)) union= header(memory-table(left))
    jnode = right
  return
    jnode
```

Figure 5.16: Pseudo code for joining a node and an escape filter node

```
def join(level, fnode1, snode, type):
  jnode = nil
  case
    sometime-past-pattern?(type) then jnode = join-sometime-past(level, fnode, snode)
    sometime-interval-pattern?(type) then jnode = join-sometime-interval(level, fnode, snode)
    previous-pattern?(type) then jnode = join-previous(level, fnode, snode)
    escape-pattern?(type) then jnode = join-escape(level, fnode, snode)
    not-pattern?(type) then jnode = join-not(level, fnode, snode)
  else /* a regular pattern*/
    fnode = join-node(level, fnode, snode)
  return
    /* return created join node */
    jnode
```

Figure 5.17: Pseudo code for joining two nodes. Depending on the given type, a different join function is used to join the nodes.

3. Joining a node and a node

If we try to join two nodes, where the second node's predicate name is neither `create`, `call`, `sometime-past`, etc, we simply use a plain Rete join node (see section 5.3).

5.4.2 Interpreter

Inserting a fact in the network, means the fact is propagated through the different nodes in the network. Depending on the type of node a fact is inserted in, the fact is either discarded, stored in a memory table or passed along to the node its output node. We next review the behavior for the different types of nodes.

Root node

If a fact is inserted in the root node of the network, it is just inserted in all the root node's output nodes (ergo the fact is inserted in all the filter nodes).

```
def join-sometime-past(level, left, right):
    jnode = sometime-past-join-node(left-input = left,
                            right-input = right,
                            memory-table = union(memory-table(left),
                                                 memory-table(right))

    /* set time stamp constraints, i.e. join a escape constraint to
    the created join node */
    T1 = leftest-timestamp(left)
    T2 = leftest-timestamp(right)
    jnode = join-escape(level, jnode, compile-pattern(< T2 T1))
    /* try to join one of the unjoined escape nodes*/
    join-escape-nodes(level, jnode)
    return
      jnode

def join-escape-nodes(level, jnode):
  tryouts = fetch all pairs in *map-of-nodes-to-be-added*
            where the level of the pair >+ level
  for tryout in tryouts
    join-escape(level(tryout), jnode, tryout)
```

Figure 5.18: Pseudo code where the second node represents a pattern with predicate name = temporal operator

Filter node

The result for inserting a fact in a filter node is as follows. If the fact matches the filter node, the fact is stored in the filter node by saving all the arguments of the fact in the memory table of the node, and the fact is inserted in the output node of the filter node. A fact matches a filter-node if: 1) the predicate name is the same as the node's label and 2) the argument list unifies with the header the memory table. For example, the fact `(call "login" jeff)` matches the filter node depicted in Figure 5.20, because the label of the filter node is equal to the predicate name of the fact, namely `call`, and the arguments of the fact, namely `("login" jeff)` unify with the filter node's memory table's header, namely `("login" ?User)`. Consequently, an entry "jeff" is made for the variable `?User` in the memory table and the fact is inserted in the output node of the filter node. For example, the fact `(call "logout" lotte)` does not match the filter node depicted in Figure 5.20, because the arguments of the fact, namely `("logout" lotte)` do not unify with the filter node's memory table's header, being `("login" ?User)`, and consequently no entries are made in the fact isn't saved in the filter node nor inserted in the output node of the filter node.

Escape filter node

When a pattern is inserted in an escape filter node, the lisp-form of the escape filter node is evaluated, using the variable bindings in the memory table from in its input node to bind the logic variables in the the lisp-form, and the returned value is entered in the memory-table of the escape filter node. Next, the pattern is inserted in the escape filter node's output node.

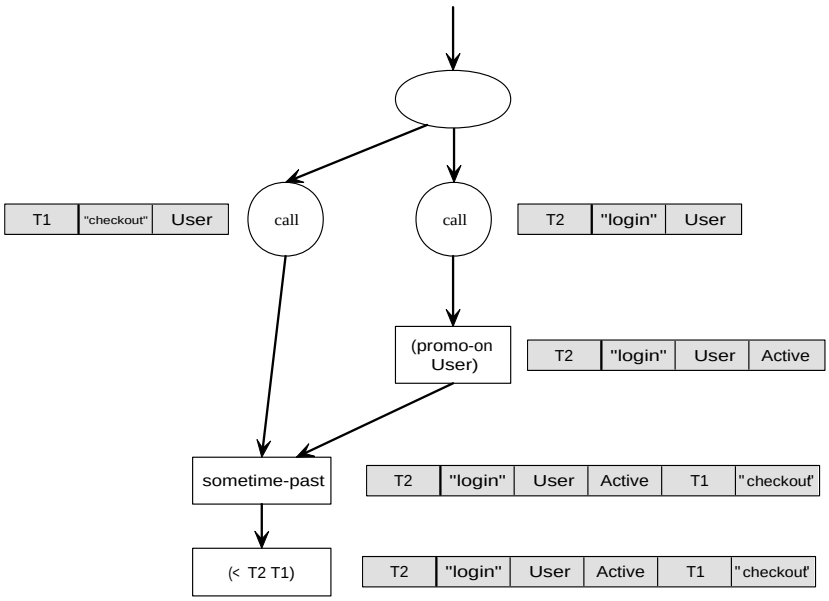

Figure 5.19: Rete for the conjunction `((call "checkout" (User)) (sometime-past (call "login" (User)) (escape ?Active (promo-on ?User)))`.

Copy filter node

When a pattern is inserted in a copy filter node, a deep copy[4] of all the pattern arguments is taken and entered in the memory table (this implements 5.1.3).

Finally for all types of filter nodes, if the inserted pattern matches the filter node, it is inserted in the output node of the filter node. Next, the pattern is inserted in the copy filter node's output node.

Join nodes

The behavior strongly depends on the type of join node one is inserting a pattern in. We define the behavior per join node type:

join node : if the pattern is inserted in the left input: For each pattern stored in the memory table of the right input, see if you can combine it with the inserted pattern into one pattern – if unification for the common variables succeeds – to store in the memory table of the join node; Else if the pattern is inserted in the right input, do the same, but for the tupels in the memory table of the left input. Insert each combination in the output node of the join node.

sometime-past join node : if the pattern is inserted in the left input: For each pattern stored in the memory table of the right input, if we can

[4]In our current implementation, deep copies are taken using the generic function `copy`. The HALO programmer can implement a method `copy` for the types of objects for which copies are taken; This way, the programmer can avoid to copy irrelevant state.

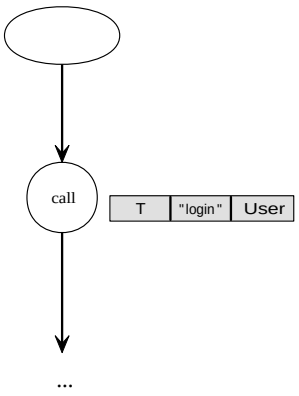

Figure 5.20: Rete filter node for the pattern `(call "login" ?User)`

combine it with the inserted pattern into one pattern – if unification for the common variables succeeds – to store in the memory-table of the join node and insert each combination in the output node of the join node. Otherwise, if the pattern is inserted in the right input, we do not try to combine it with the tuples already inserted in the left input. This is because tuples in the right input represent facts that are supposed to be inserted *before* any fact inserted in the left input, and consequently if we insert a pattern in the right input, all the facts inserted in the left input were inserted *before* the one we are inserting now. In other words, we know that in this case that the condition comparing the time stamps of patterns we are trying to combine, will always fail. E.g. upon inserting the fact `(call 3 buy lotte cd)` in the right input of the sometime-past join node depicted in Figure 5.21, we do not try to combine it with any of the patterns that were inserted *before* in the join node's left input.

previous join node : if the pattern is inserted in the left input, for a pattern stored in the memory table of the right input, see if you can combine it with the inserted pattern into one pattern – if unification for the common variables succeeds and the time stamps are only one time unit apart – to store in the memory table of the join node; if it comes from the right input, discard it. Insert the combination in the output node of the join node.

Production nodes

When a fact is inserted in a production node, the advice is executed, by applying the function that implements the advice on the arguments bound in the inserted pattern. When the insertion of a fact in a network, results in inserting multiple patterns in a production node, the production node is fired for the first pattern inserted.

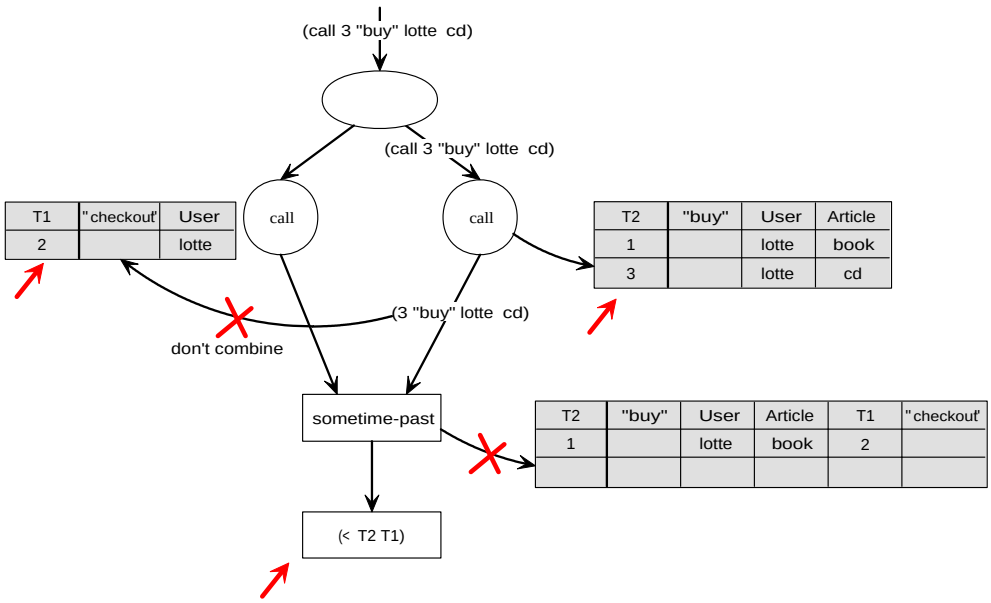

Figure 5.21: Inserting the fact `(call 3 "buy" lotte cd)`

5.4.3 Combining insertion and garbage collection

We can combine the garbage collection described in section 5.1.4 and the insertion mechanism: this involves deleting facts from the network, when a new fact is inserted.

Rule 5.1.4 says we can delete all facts (for a rule) that do not match with any pattern in the rule that is the argument of a temporal operator. We can mark the filter nodes during the compilation process that are captured by a temporal operator (e.g. extend the pseudo code for compiling a pattern by adding a line that marks the nodes). Now, whenever a call or create fact is inserted in an unmarked filter node, we immediately delete the fact from the network, once the network has stopped processing the insertion of the fact.

Rule 5.1.4 - 5.1.4 defines when we can delete a fact (for a rule) that matches with any pattern in the rule that is the argument of a temporal operator. Basically, we can delete a fact, if we insert a new fact that matches all the conditions placed on the arguments of the pattern except if there are conditions that involve variables bound outside the scope of the temporal operator. When we can insert a fact in a sometime-past join node, we immediately delete the previously inserted facts that were inserted in the sometime past join node, if there are no escape filter nodes as output of the sometime past join node, because that implies the exception from section 5.1.4.

For example, imagine we have only one HALO rule, namely:

```
(defrule (advice "discount" discount (?User ?Article 0.10))
  (call "buy" (?User ?Article))
  (sometime-past
    (call "login" (?User))
```

```
(escape ?On (promo-active (?User)))))
```

This rule is compiled to the Rete network depicted in Figure 5.22. Consider now that we have the following execution history:

```
1. (login lotte)
2. (login lotte)
3. (buy lotte book)
4. (buy lotte comic)
5. (buy lotte cd)
```

A user `lotte` logs in to the shop, logs in again, and next buys a `book`, `cd` and a `comic`[5], which results in inserting the facts:

```
1. (call 1 "login" lotte)
2. (call 2 "login" lotte)
3. (call 3 "buy" lotte book)
4. (call  4 "buy" lotte comic)
5. (call 5 "buy" lotte cd)
```

in the network depicted in Figure 5.22.

The advice we defined earlier, gives a discount at every "buy" event if the user logged in when the promotions were active. According to the garbage collection rules described in section 5.1.4, we can delete all of the facts, except the fact `(call 2 "login" lotte)` (the crossed rows in Figure 5.22 reflect deleted facts). The facts nr. 3 - 4 are immediately deleted after they are inserted in the network and the fact nr. 1 is deleted after the fact nr. 2 is inserted.

5.4.4 Encountered difficulties

There are a number of difficulties for implementing HALO by means of Rete networks, we initially did not consider. We next discuss them one by one and explain how they are dealt with currently.

Rule definitions

The original Rete algorithm is written for production systems, where there are only productions, a sort of *if-then* rules that *fire* when certain facts are added in the logic repository, which results in adding/removing a new fact in the logic repository. We can try to see HALO rules as productions, however, there are some problems with HALO rules that make use of the `escape` predicate.

Consider for example the rules (remember we call all rule definitions that do not extend the definition of the predicate `advice` contexts, whereas the rules that do extend the predicate `advice` are called advices):

```
1. (defrule (stock-promo-active (?User ?Article))
2.   (escape ?Shop (shop ?User))
3.   (escape (stock-overflow ?Shop ?Article)))

4. (defrule (advice "discount" discount (?User ?Article 0.10))
```

[5]The symbol `lotte` refers to an instance of the class `User` and the symbols `book`, `cd` and `comic` refer to instances of the class `Article`.

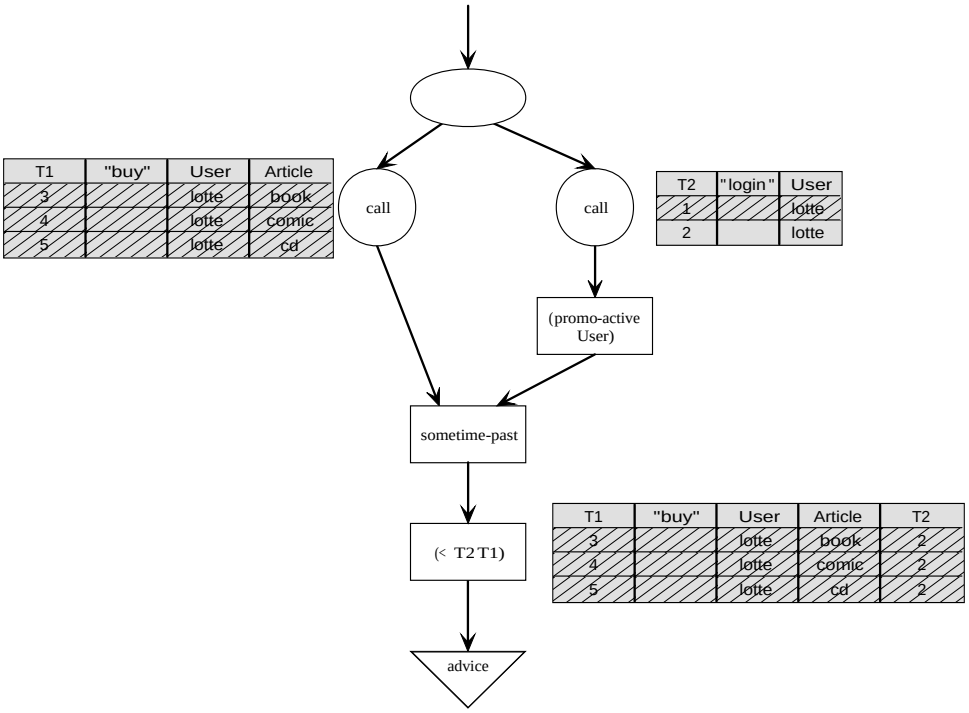

Figure 5.22: Crossed rows = deleted facts

```
5.    (call "buy" (?User ?Article))
6.  (sometime-past
7.    (call "login" (?User))
8.    (stock-promo-active (?User ?Article))))
```

How do we compile this to Rete? Do we simply compile the advice and the context rule as usual (Figure 5.23)? Then how do we make sure `(stock-promo-active (?User ?Article))` facts are generated? Well, we can assure this if we say that firing the context rule, results in adding such a fact to the network. However, to what node do we connect the Rete gotten from compiling the context rule? We can't just attach it to the root node. The problem is that the Rete gotten from compiling the context rule needs input for the variables variables `?User` and `?Article`...

We have opted to solve this as follows. All the advices are recompiled so that all occurrences of patterns that refer to context rules are replaced by the body of the context rule (pseudo code in Figure 5.25). If there are multiple rules defined for the same context predicate, we generate multiple new advice rules. For the example discussed earlier, we replace the context rule and advice by one rule:

```
(defrule (advice "discount" discount (?User ?Article 0.10))
   (call "buy" (?User ?Article))
 (sometime-past
   (call "login" (?User))
```

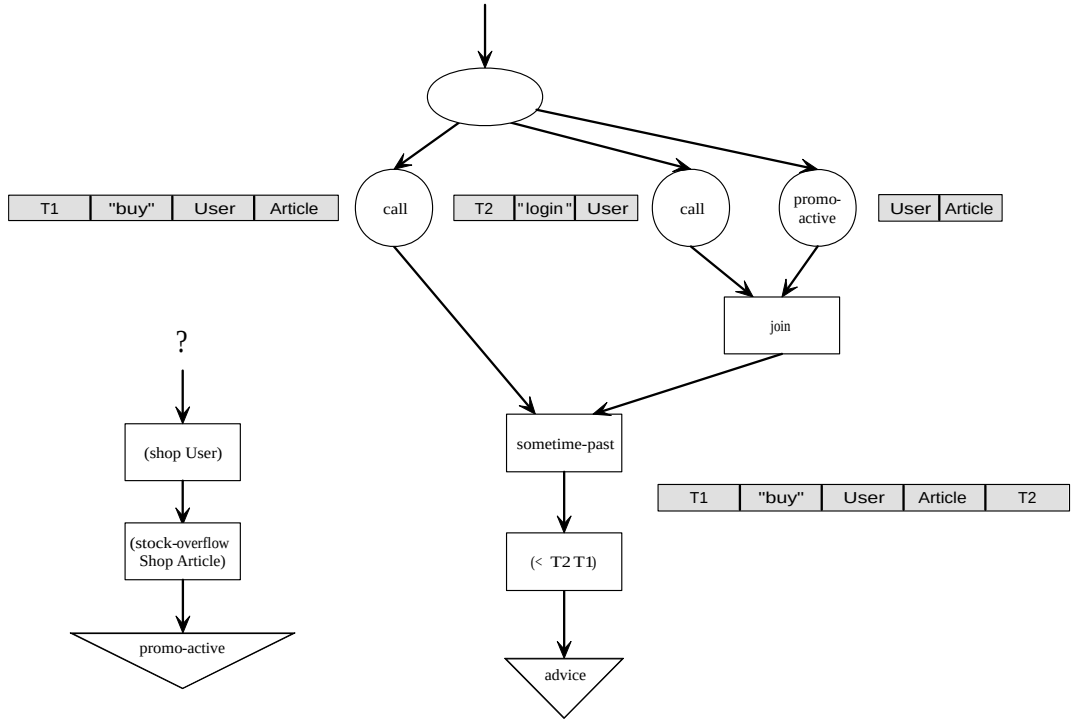

Figure 5.23: To what node do we connect the Rete generated for a context rule?

```
(escape ?Shop (shop ?User))
(escape (stock-overflow ?Shop ?Article))))
```

The advantage of dealing with context rules this way, is that we don't need to change anything to the compiler defined in Section 5.4.1 (see Figure 5.24). A drawback however, is that we can't deal with recursive rules this way.

5.4.5 Future work

The presence of certain facts prevents objects from being garbage collected

Events are stored under the form of facts in the logic repository. E.g. executing the piece of code `(buy *lotte* *book*)` results in storing a fact `(call "buy" *lotte* *book*)`where `*lotte*` and `*book*` represent variables bound to an instance of the class `User` and `Book` respectively (strong references). Normally, the fact garbage collector makes sure a fact is deleted from memory when the fact can't possibly be needed to resolve a query, assuring that the objects `*book*` and `*lotte*` can be collected by the (Lisp) garbage collector. However, there are rules for which it cannot be decided by the fact garbage collector if the fact is garbage (i.e. if it isn't needed to resolve any possible query (see section 5.1.4)).

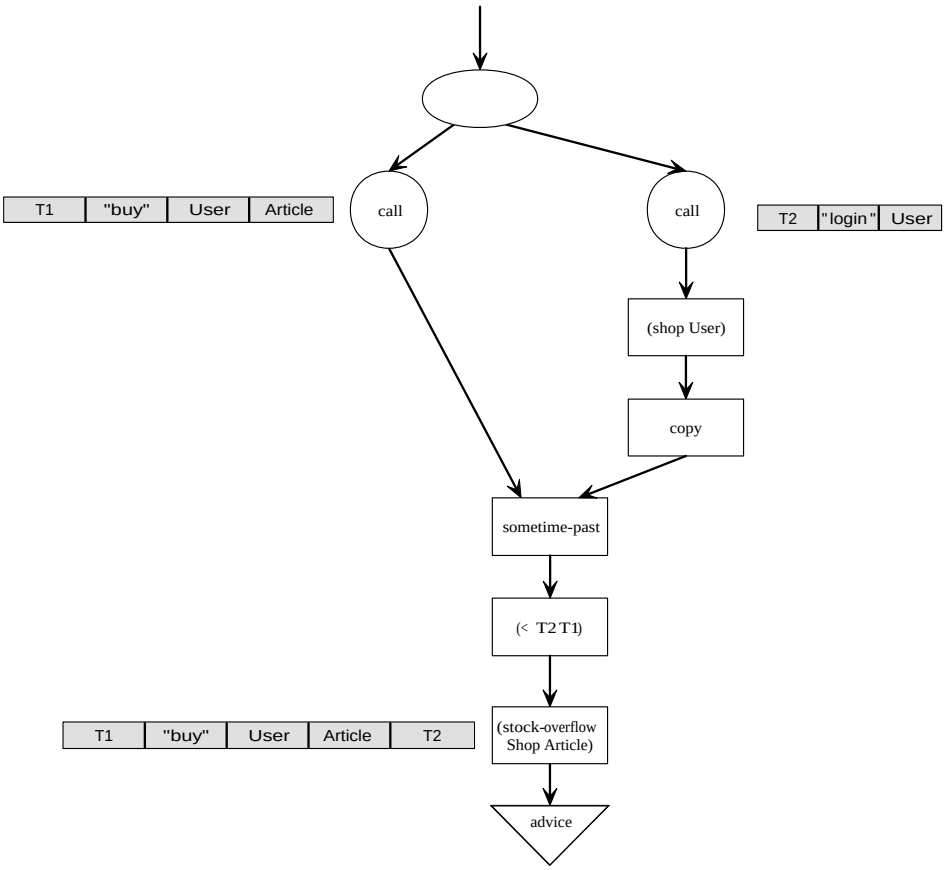

Figure 5.24: Rete for Figure 5.23

Take for example the following rule and execution history [6]:

```
(defrule (advice C c (?X ?Y))
    (call A ?X)
    (sometime-past (call B ?Y) (escape ?Greater (> ?Y ?X))))
```

```
1. (call A *number1*)
2. (call A *nulber2*)
3. (call B *number3*)
```

The variables `number` refer to instances of a class `Number`. The fact garbage collector cannot decide whether the fact `(call A *number1*)` is garbage, because there might come a fact `(call B *someNumber*)` for which the query `(advice C c (?X *someNumber*))` yields a different result depending on whether `(call A *number1*)` or `(call A *number2*)` is used to resolve the query. In short, the facts `(call A *number1*)` and `(call A *number2*)` are never removed

[6] Whether or not this example makes "sense" is not its purpose. It is just here to show a problem that *could* occur.

```
compile-advice(advice):
  repeat
    advice = replace-context-patterns(advice)
  until
    there are no more context patterns in the advice
  return
    advice

replace-context-patterns(advice):
  for all context-pattern in advice
  do
    advice = replace-pattern-by-context-body(advice, context-pattern)
  return advice
```

Figure 5.25: Pseudo code for compiling context-patterns out of an advice rule, where a context-pattern is a pattern that unifies with a context rule's head.

from the logic repository and consequently there exists forever a reference to the objects bound to `*number1*` and `*number2*` such that these objects will never be garbage collected. This is something the HALO programmer should be aware of, as rules like this can be very memory consuming. Perhaps it can be useful to give the HALO programmer control over the fact garbage collector, the same way as control over the (Lisp) garbage collector is given by means of *weak references* [31], so that facts like this *can* be deleted. However, for this dissertation, we did not consider to grant user control to the fact garbage collector. Furthermore, we should make an estimate of how often this problem occurs.

5.5 Summary

In this chapter we presented the ideas behind the implementation of HALO. We started the chapter by giving an overview and description of the (HALO specific) difficulties for implementing the weaver.

We dedicated a next section to explaining the Rete algorithm, as we chose to implement the HALO engine using a forward chainer. We proposed to represent the logic repository by means of Rete networks. Next we introduced the changes we needed to make to the Rete algorithm, involving a compilation scheme dedicated to HALO rules; We introduced new types of nodes that can be used to represent HALO specific patterns, such as call, create, escape patterns and new types of join nodes to represent the temporal operators in a HALO rule, and a special copy node. Specific behavior was introduced for inserting facts in these new types of rules, reflecting the HALO semantics for evaluating a rule.

Finally, we concluded the chapter with an overview of encountered difficulties involving garbage collection and implementing the support of abstraction by means of rule definition in HALO.

Chapter 6

Conclusions

6.1 Summary

Full separation of concerns through modularisation using a traditional object-oriented, functional or procedural programming language is difficult to achieve, because a program can be modularised in only one way at a time. Concerns that do not align with a particular modularisation are called croscutting concerns: their implementation is scattered over different modules, leading to tangled code, which is hard to understand, reuse and maintain.

In Chapter 2, we discussed aspect-oriented programming (AOP), a novel paradigm aiming to modularise crosscutting concerns. In AOP, crosscuts are defined in terms of join points, where join points are well-defined points in (the execution of) a base program affected by (the implementation of) a crosscutting concern. An AOP language is a language extension for a base language that makes it possible to implement a crosscutting concern in a separate module, called an aspect. We discussed two different aspect languages, namely AspectJ and CARMA to illustrate AOP. In both languages, the idea is to declare pointcuts, which describe a set of join points, and connect these pointcuts to a piece of advice code. This advice code is then assured to be executed – by a weaver program – at any join point described by the pointcut. The major difference between CARMA and AspectJ, is that CARMA is a logic meta programming approach to AOP, introducing the idea of using of a full-fledged logic meta language as pointcut language, which leads to a very declarative and open pointcut language. Next, we explained that past AOP research focused on expressing aspects that are triggered on the occurrence of a single join point.

Recent research evolved towards aspects that are triggered on the occurrence of a sequence of join points: they were dubbed stateful aspects, event-based aspects and context-aware aspects. We discussed AOP environments that offer some support to implement these kinds of aspects (EAOP, Reflex, JAsCo). We concluded that so far, no pointcut language based on temporal logic, or adequate in reasoning about past program state has been proposed. However we believe that a dedicated pointcut language can improve the expressiveness of such an AOP system.

We designed our pointcut language after the following description of program execution. Program execution generates a (dynamic) join point at each

computational step. All the join points that are generated this way, can be placed on a time line, reflecting the order in which they occurred and we observe that there exists a temporal relation between any two join points on this time line. If we can express this temporal relation, we can describe the sequence of events in a pointcut.

In Chapter 3 we discussed logic meta programming in full detail. First we discussed the origins of logic programming, by introducing the reader to Prolog and then we continued the chapter by introducing the ideas behind logic meta programming, namely to reason about programs, written in some (object-oriented) base language by means of logic programs. We covered some applications of logic meta programming, including aspect-oriented programming and noticed that all logic meta programming approaches to aspect-oriented programming, currently being developed propose a logic meta language based on Prolog. We tried to solve our problem using Prolog as well. The solution consisted of introducing an extra variable to each predicate, which saves a spot for adding a time stamp; Then we could express temporal relations by putting constraints on these temporal operators. However, manipulating time stamps is very tedious and error-prone. Therefore, we explored temporal logic, a formalism dedicated to reasoning about time and temporal relations. Temporal logic is an umbrella term for a set of logic languages that allow the representation of temporal information. These logics introduce new types of logic connectives, called temporal operators, into an existing logic and temporal logic formulas are evaluated in relation to an implicit temporal context. Depending on the definition of time applied, these temporal operators have different meanings. A form of metric temporal logic we investigated, defines time as a linear sequence of time points and the temporal operators define temporal relations in terms of intervals of time points. This definition matches well on our problem description, and consequently we used a subset of this form of metric temporal logic as a basis for designing our pointcut language.

In Chapter 4 we covered the syntax and semantics of our pointcut language, which we named HALO – short for history-based aspects using logic. HALO is a full-fledged logic language that comes with a library of predicates to describe single join points in the execution of a program (method call, instance creation) and temporal predicates that can be used to express a temporal relation between different pointcuts. Evaluating a pointcut that relates different join points in the execution of a program is done in respect to the state the program was in when each such join point occurred. This makes HALO suitable to reason about the past and current state of a program. In addition, you can put conditions on a variable that will be bound only later. To evaluate HALO – and hence validate our thesis – we implemented the e-commerce application used in several papers about context-aware and stateful aspects as an example application, as well as some other examples.

Chapter 5 discusses the implementation of HALO. First, we explained the general idea behind the HALO weaver. The weaver is responsible for generating/storing a logic fact for each join point and for querying the logic repository at each step in the execution of a program, in order to find applicable advice code. We carefully analyzed the problems we needed to tackle due the semantics of HALO and outlined the solutions:

- *generating and advising join points* through CLOS meta object protocol

- *interpretation of HALO programs* through translation of HALO programs to Prolog
- *saving object state* at well-defined moments in time
- *garbage collection of facts* based on the idea that some facts can't be used to resolve a query after a while

We defended the use of a forward chainer to implement the HALO interpreter and we chose to use the Rete algorithm, where logic rules are represented as networks of nodes. However, the logic language for which the Rete algorithm was originally written, is much less complicated than HALO and hence Rete doesn't support the HALO-specific properties such as rule definitions, the problems discussed above, higher-order predicates such as HALO's temporal operators etc. . Therefore we needed to extend the Rete algorithms with new types of nodes and write a HALO-specific rule compiler, taking these HALO-specific needs into account.

6.2 Contributions

In this dissertation we showed that it is possible to reason about the execution history and past program state, in order to implement certain crosscutting concerns, through:

- the use of a logic meta language,
- the use of temporal logic,
- the use of a forward chainer, our own extension of the Rete algorithm.

We noted that the lack of a dedicated pointcut language makes it hard to write aspects to implement crosscuts that affect join points in the execution of a program, based on the occurrence of other join points in the past and depending on the current and past program state, though crosscuts like these are omnipresent. We designed a pointcut language, named HALO, to cope with this need.

HALO is a logic pointcut language – in the spirit of CARMA, but based on temporal logic – where aspects are defined in terms of logic rules. We defined a set of primitive predicates to capture join points in the execution of a program and a set of higher-order predicates, based on the temporal operators in metric temporal logic, to define a temporal relation between pointcuts. Evaluating a pointcut about multiple (past) join points in HALO, is done in respect to the the program state at these (past) join points, so that constraints about past join points are checked against the program state at the occurrence of a past join point. This makes it possible to reason about current and past program state in HALO, and furthermore, it is possible to put constraints on a past join point in a pointcut, that involve values from a later join point.

We have implemented HALO for Common Lisp. This implementation involves the design of a dedicated forward chainer, for which we needed to extend the Rete algorithm. In short, the design and implementation of HALO are the main contributions of this dissertation.

6.3 Future work

We based HALO on a subset of the temporal operators in metric temporal logic, being the past temporal operators. Perhaps it might be useful to include the future-oriented temporal operators as well. Of course we need to think about a well-defined semantics for these operators. We could for example define them as follows. We could introduce the operators `sometime-next` and `next` and define their semantics in terms of the current execution history. For example, a pointcut `sometime-next (call "logout" (User))` would be false, if this is evaluated "now", i.e. at the current time point. However this does not imply the future operators would be useless: they could be useful when used *in combination with a past temporal operator*, so that such a *future* pointcut evaluates against that past context. For example, this could be used in a pointcut

```
(sometime-past (call "login" (User))
               (not (sometime-next (call "logout" (User)))))
```

that only matches the last `login` event if it was *not* followed by a `logout` event. It is arguable that this is more expressive than writing a pointcut

```
((sometime-past (call "login" (User))) (logged-in (User)))
```

that matches the most recent `login` event if that implies the `User` is still logged in now, because this requires the presence of some state variable to keep track of the "login" state of a user.

There are other temporal logics out there that introduce different kinds of operators, that might be worthwhile to include in HALO. For example, in [26] a new temporal operator `N` is introduced. The authors of that paper explain that `N` should be read as "from now on" and that this temporal operator can be used to evaluate temporal formulas that ignore parts of the past in their evaluation. Maybe we can turn this construct into a means of control over the fact garbage collector (we discussed in chapter 5 that giving the programmer control over the fact garbage collector might be beneficial).

Currently, we implemented HALO from scratch for Common Lisp, but perhaps it is interesting to investigate an implementation of HALO on top of an existing AOP approach, such as Reflex for Java. Nevertheless, we should pursue to make the current HALO implementation based on our extension of the Rete algorithm more efficient. For example, there certainly is room for improvement for the fact garbage collector. For example, we identified that we cannot decide if certain facts ever become garbage, because they can be used to resolve pointcuts that rely on the occurrence of a future event. Nevertheless, we could offer some support for warning the HALO programmer when she is writing such pointcuts and offer some form of control over the fact garbage collector. In addition, the compile-away tactic used to implement rule definitions should be rethought, to make recursive rule definitions possible in HALO, as this would greatly improve HALO's power. In this light, it is maybe worthwhile to investigate the use of a mixed resolution strategy.

In this dissertation we evaluated HALO on a few example applications, but we really should evaluate the use of HALO in a large-scale system, for example to find out if HALO's expressiveness can be improved and how well the current implementation based on an extended version of the Rete algorithm performs.

6.4 Related work

In this section we cover related work: we take a look at AOP languages that allow the implementation of aspects based on multiple join points in the execution history of a program we highlight the differences with our approach. Of course, we point out that already several of these approaches were discussed in Chapter 2, such as EAOP, JAsCo and Reflex. Remember that all these approaches lack a (sufficiently) declarative pointcut language.

6.4.1 Alpha

Alpha [30] is a logic-programming-based approach to AOP for a prototype Java-like language, whereas HALO is implemented for Common Lisp. It is possible to reason about the execution history of a program in Alpha: events such as method calls are represented by relations that have explicit time stamps. A number of predicates is defined to express the order between time stamps and these are used to express the order of events in a pointcut. In our approach however, temporal operators are higher order predicates: this allows us to easily express that certain conditions must hold at the time a specific event occurs and the conditions inside a temporal operator can refer to variables bound by a later event. Our mechanism assures to evaluate pointcuts in relation to *past* program state automatically.

6.4.2 Tracematches

Tracematches [1] is an extension to AspectJ that enables the programmer to trigger the execution of advice code by specifying a regular pattern of events in a computation trace. Tracematches includes a free-variables mechanism in which events can be matched not only by the event kind, but also by the values associated with the free variables (pointer equal), whereas in our approach the past program state is taken into account. Furthermore, in this dissertation our main goal, was expressiveness. Our Logic pointcut language makes it possible to abstract pointcuts by rule definition and make use of unification, which is much more expressive than a pointcut language based on regular expressions.

6.4.3 J-Lo

J-LO [4] is a tool for checking temporal assertions in Java programs. These temporal assertions are written down as LTL formulas in the form of Java annotations in the source code. LTL [21] is a logic that extends propositional logic with (future-oriented) temporal operators to arrange propositions on a time line. In J-LO, AspectJ pointcuts are the propositions. J-LO also supports a free-variables mechanism, which can be used to refer to variables bound earlier in the execution history, but not to variables bound at a later stage in the execution history, as is possible in HALO. Another difference with HALO is that J-LO uses a future-oriented logic, whereas HALO is based on a past temporal logic, which requires a different way of thinking about program execution. Furthermore, J-LO focuses on verification (model checking) and is not used to implement aspects or insert code: J-LO is *not* used as a pointcut language. In addition, J-LO doesn't offer a mechanism to refer to past program state, as HALO does.

6.4.4 Extending the RETE Algorithm for Event Management

In [3] an extension of the Rete algorithm is proposed to integrate event management in a rule engine. The author introduces the concepts of "events" and of "temporal constraints" between events and extends the OPS5 language to include constructs to define these. Events are a special type of facts that are time stamped and the idea is to use temporal constraints, to express a temporal relation between two events in terms of time intervals: e.g. event A occurs within 5 time steps from event B. These time intervals are used to derive "expiry dates" for events and the Rete algorithm is extended so that these expiry dates are checked at each time step to see if an event can be deleted. However, the language extension proposed in that paper is very different from HALO. For example, temporal constraints can only express that certain events happen within a limited number of time steps. The author claims that this is necessary to make it possible to throw away events. However, in HALO, we allow pointcuts to refer to past events (unrestricted how far in the past), but this does not make it impossible to throw away events. Also, in [3], no extension of Rete is discussed to support a language symbiosis mechanism between the rule language and another language such as our extension (e.g. see the copy mechanism and reordering strategy of constraints to implement the `escape` predicate).

Bibliography

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondej Lhotk, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM Press.

[2] Christine Bailey and Michael Katchabaw. An experimental testbed to enable auto-dynamic difficulty in modern video games. In *In Proceedings of the 2005 GameOn North America Conference*, 2005.

[3] Bruno Berstel. Extending the rete algorithm for event management. In *TIME*, pages 49–51, 2002.

[4] Eric Bodden. J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen university, 2005.

[5] Jonas Bonér. What are the key issues for commercial aop use: how does aspectwerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 5–6, New York, NY, USA, 2004. ACM Press.

[6] Johan Brichau. Declarative meta programming. 2002.

[7] De Fraine Bruno, Vanderperren Wim, Suvee Davy, and Brichau Johan. Jumping aspects revisited. *DAW 2005, Chicago, USA*, 2005.

[8] Christoph Brzoska. Temporal logic programming with bounded universal modality goals. *In proc. of the 10th International Conference of Logic Programming, Budapest*, 1993.

[9] Christoph Brzoska. Temporal logic programming with metric and past operators. *Lecture notes in computer science, p. 21-39*, 879, 1995.

[10] B. Jack Copeland. Arthur prior. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 1999.

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

[12] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. *Lecture Notes in Computer Science*, 2192:170–184, 2001.

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

[14] Charles L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. pages 324 – 341, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

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

[16] Manolis Gergatsoulis. Temporal and modal logic programming languages. In A. Kent and J.G. Williams, editors, *Encyclopedia of Microcomputers*, volume 27, pages 393–4O8, New York, 2001. Marcel Dekker Inc.

[17] K. Gybels. Using a logic language to express cross-cutting through dynamic joinpoints. *In Second Workshop on Aspect-Oriented Software Development, Bonn, Germany*, 2002.

[18] Kris Gybels. Soul and smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis. *In proceedings of the workshop on declarative programming in the context of object-oriented languages*, 2003.

[19] Kris Gybels. Research: Aspect-oriented programming and carma, 2006.

[20] 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 Press.

[21] Keijo Heljanko. Networks and processes: Temporal logic ltl. 2003.

[22] Gregor Kiczales Hidehiko Masuhara and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. *In compiler construction, volume 2622 of Lecture Notes in Computer Science*, pages 46–60, 2003.

[23] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[24] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[25] Roel Wuyt Kim Mens, Isabel Michiels. Supporting software development through declaratively codified programming patterns. *SEKE 2001 Special Issue of Elsevier Journal on Expert Systems with Applications*, 2002.

[26] Francois Laroussinie, Nicolas Markey, and Ph. Schnoebelen. Temporal logic with forgettable past. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 383–392, Washington, DC, USA, 2002. IEEE Computer Society.

[27] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. *In Foundations of Aspect-Oriented Languages (FOAL), Workshop at AOSD 2002.*, 2002.

[28] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *SEKE 2001 Special Issue of Elsevier Journal on Expert Systems with Applications*, 2001.

[29] Tom Mens and Serge Demeyer. Evolution metrics. In *Proc. Int. Workshop on Principles of Software Evolution*, 2001.

[30] Klaus Ostermann, Mira Mezini, and Christoph Bockish. Expressive pointcuts for increased modularity. In *European Conference on Object-Oriented Programming*, 2005.

[31] Monica Pawlan. Reference objects and garbage collection, 1998.

[32] Frederic Portoraro. Automated reasoning. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2005.

[33] Mario Südholt Rémi Douence, Olivier Motelet. Sophisticated crosscuts for ecommerce. *Proceedings of the workshop on Advanced Separation of Concerns*, 2001.

[34] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[35] Mario Südholt. Eaop tool. 2004.

[36] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. *Lecture Notes in Computer Science (accepted at Software Composition symposium 2006, to be published)*, 2006.

[37] Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed aop. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025, pages 316–331, Bologna, Italy.

[38] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

[39] Aspectj Team. *Aspectj programming guide: semantics*, 2005.

[40] AspectJ Team. Aspectj 5 language changes, 2006.

[41] Wim Vanderperren. *JasCo quickreference*, 2006.

[42] Wim Vanderperren. Jasco website, 2006.

[43] Kris De Volder. Jquery: A generic code browser with a declarative configuration language. In *PADL*, pages 88–102, 2006.

[44] Kris De Volder and Theo D'Hondt. Aspect-oriented logic meta programming. *Lecture Notes in Computer Science*, 1616:250–260, 1999.

[45] Edsger W.Dijkstra. On the role of scientific thought. 1974.

[46] J. Wielemaker. *SWI-Prolog Reference Manual*, 1997.

[47] Johan Brichau Wolfgang De Meuter and Kim Mens. *SOULManual*, 2003.