# Forward Chaining in HALO: An Implementation Strategy for History-based Logic Pointcuts

Charlotte Herzeel[a], Kris Gybels[a], Pascal Costanza[a],
Coen De Roover[a], and Theo D'Hondt[a]

[a]Programming Technology Laboratory, Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussels, Belgium

In aspect-oriented programming, pointcuts are formulated as conditions over the context of dynamic events in the execution of a program. Hybrid pointcut languages also allow this context to come from interactions between the pointcut language and the base program. While some pointcut languages only allow conditions on the current execution event, more recent proposals have demonstrated the need for expressing conditions over a history of join points. Such pointcut languages require means to balance the expressiveness of the language with the additional memory and runtime overhead caused by keeping a history of join point context data. In this paper, we introduce a logic-based pointcut language that allows interaction with the base program as well as pointcuts over a history of join points. We introduce forward chaining as an implementation model for this language, and discuss possible optimization strategies for the additional overhead. [1]

## 1. Introduction

A good modular design decomposes program concerns into separate modules, each implementing a different concern. Some concerns are, however, inherently crosscutting, which means that their implementation is scattered over different modules. Aspect-oriented Programming (AOP) focuses on the modularisation of such crosscutting concerns [18]. An AOP language provides a notion of join points which are events in the execution of a program, a pointcut language to concisely describe multiple join points and advice which affect the program behavior at the join points captured by a pointcut.

Research into pointcut languages has shown that these can be made more expressive in several ways [19]. Two of them are: allowing pointcuts to be expressed over a history of join points [1,21], and allowing pointcuts to interact with the base language through a mechanism like hybrid pointcuts [9], which allows messages to be sent to objects. Several pointcut languages are also based on logic programming. In that approach, join points are represented as logic facts and pointcuts as logic queries over these facts. In this paper, we focus on how logic-based pointcut languages [13,21,14,27] can be combined with history-based and hybrid pointcuts. Most logic-based pointcut languages are based on Prolog, which uses backward chaining to evaluate logic queries. In this paper, we demonstrate

[1]ACM, 2007. This is a minor revision of the work published in Proceedings of the 2007 International Conference on Dynamic Languages (ESUG/ICDL 2007) http://doi.acm.org/10.1145/1352678.1352689

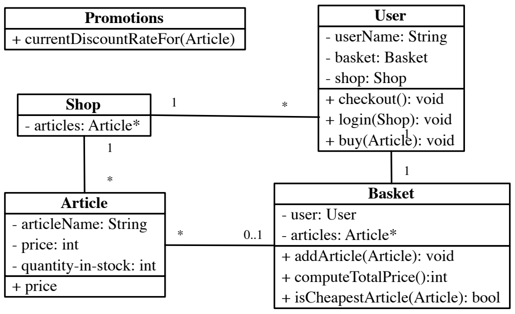


Figure 1. A small e-shop application.

why this chaining strategy fails for a combination of history-based and hybrid pointcuts, and introduce forward chaining as an alternative. The contributions of this paper are:

- We introduce a novel logic-based pointcut language, HALO, which allows both pointcuts over a history of join points as well as a mechanism to interact with the base language.
- We introduce forward chaining, in particular the Rete algorithm, as an implementation strategy for such a language, including our extension to Rete to support expressing temporal relations between join points in the history.
- We discuss how the predicates for expressing temporal relations in HALO enable space optimization of the join point history.

In the next section, we introduce a small running example demonstrating the need for a pointcut that is both history-based and interacts with the base program. In Section 3, we introduce a logic-based pointcut language that supports both features. In Section 4, we contrast backward and forward chaining, and more extensively discuss the Rete-based implementation of HALO. The final sections of the paper contrast HALO with related work, and discuss future work and our conclusions.

## 2. Running example

We use a simple e-commerce application as a running example to illustrate HALO. A UML diagram for this example is given in Figure 1. Users have an account and have to log in before they can add articles to their shopping basket of which the total price is calculated when they check out.

To attract customers, the shop occasionally engages in promotional marketing campaigns. The application therefore has a singleton class `Promotions` which gives the current rate of discount for each article. Several variations of this class are possible. It can be implemented as a simple table that stores the current discount rate for each article. The computation can, however, be more complex, for example based on the current amount of stock for the article.

In this example, one possible effect of the promotions is that banners pop up to advertise promotions. Another possible effect of the promotions is that customers get a discount on articles when they check out. Giving discounts is based on the *past* discount rate for an article. The idea is that if a promotion for an article was advertised, the user should still get the discount when she checks out, even if the promotion is no longer active. The latter can be implemented as a "discount" aspect. Depending on the shop's strategy, the aspect can give the rate from when the user logged in or from when she added the article to her basket.

This is a small example, but it is sufficient to motivate a pointcut language with two features: (1) the ability to interact with the base language to invoke the promotion object's method for determining the current discount rate, and (2) the ability to refer to past join points. While the aspect can be expressed in a pointcut language lacking either feature, it requires some effort to do so. Notably, the programmer has to provide the necessary mechanism for manually accessing and recording the past discount rate by writing two pieces of advice: The first one defines a pointcut to capture when the user logs in, and the advice body records the current discount rates of the articles. The second piece of advice defines a pointcut to capture when the checkout happens, and determines the right discount rate for an article from the recorded rates. That approach becomes more complicated the more pointcuts involve additional past join points or past data. In the next section, we introduce a logic-based pointcut language that supports both features, which allows the pointcut to be expressed more concisely. A detailed comparison of the implementation of the e-shop application solelely relying on Common Lisp and a version implemented using HALO was reported on in previous work [15]. The focus of this paper is the definition and implementation details of the HALO language.

## 3. The HALO Language

HALO ("History-based Aspects using LOgic") is a novel logic-based pointcut language for the Common Lisp Object System (CLOS) [5] that allows pointcuts to be expressed over a history of join points, as well as allowing interactions with the base language. In contrast with earlier work on logic-based pointcut languages that offer a history of join points [21], HALO imposes a fixed set of temporal predicates for expressing pointcuts over the history, drawn from temporal logic. In the next section, we discuss how imposing a fixed set of built-in predicates allows a runtime space optimization of the join point history based on the semantics of the predicates. The related work section contrasts this approach with the static analysis of the open set of predicates of earlier work. In this section, we first discuss the background on logic-based pointcut languages, and briefly give an overview of temporal logic programming. We then explain the HALO language in more detail.

### 3.1. Logic Pointcuts over Join Point Histories

#### Background on logic-based pointcut languages

Previous work on CARMA [13] and other logic-based pointcut languages [21,14,27] has demonstrated the suitability of logic programming [20] as the basis for a pointcut language. The core idea behind these languages is to represent join points as logic facts and write pointcuts as logic queries over these facts. In particular, variations of the logic programming language Prolog [8] are often used as query languages.

**Temporal logic**

To deal with temporal relations between join points in the join point history, HALO is in particular based on temporal logic programming. Between any two join points generated during the execution of a program there exists a temporal relation. This relation can henceforth be used to concisely describe sequences of join points in a pointcut. HALO offers a set of higher-order predicates to describe temporal relations between pointcuts. Higher-order temporal predicates are important for expressing hybrid pointcuts, as they indicate at what time the interaction with the base language should occur. In addition, as explained in Section 4.4, a predefined set of temporal predicates makes it possible to optimize the memory usage of a history-based pointcut language.

There are different variants of temporal logic programming [12]. These variants primarily differ in the way time is modeled (i.e. discrete or continuous, finite or infinite, etc.). A discrete time model is most suited for a pointcut language, as join points are discrete events occurring during the execution of a program. The predicates available for expressing temporal relations constitute another important difference. Most of these predicates express an ordering relation and can either be past- or future-oriented, while in some logics both are provided. For example, J-LO [6] uses linear temporal logic which provides future-oriented predicates. HALO, on the other hand, uses a past-oriented subset of metric temporal logic programming (MTL) [7].

## 3.2. HALO Advice Language

As the focus of research involving HALO is the pointcut language itself, HALO adopts the advice mechanism used in most other general-purpose aspect languages. The advice body is written in the base program language, in this case Common Lisp. Unlike other advice languages, there is no construct to distinguish between "before" and "after" advice. Instead, the join point model includes distinct "entry" and "return" join points on which the application of a piece of advice has the same effect as "before" or "after" advice in other languages.

An example to illustrate the form in which pieces of advice are written:

```
(at ((gf-call 'buy ?arguments))
   (print "Buy was invoked with arguments: " ?arguments))
```

This is an example of a straightforward logging aspect. Its pointcut comprises one condition using the predicate `gf-call`. The predicate `gf-call` is short for "generic function call". In CLOS, methods are not associated with single classes and do not have a hidden "receiver" argument as in other object-oriented programming languages. Instead, method dispatch occurs on the dynamic type of all arguments. Methods are grouped into generic functions, which perform the dynamic dispatch when called. The concept of sending a message with name $n$ to an object comes down to invoking the generic function named $n$ with that object as one of the arguments. In the pointcut, the predicate `gf-call` is used for matching calls to the generic-function named "buy". The arguments of this call will be bound to the logic variable `?arguments`.

| | | |
|---|---|---|
| *pointcut* | :: | $(< primitive\_pointcut >< escape > * < tpointcut >)$ |
| *pointcut* | :: | $(< primitive\_pointcut >< escape > *(since < tpointcut >< tpointcut >))$ |
| *tpointcut* | :: | $(\{< temporal > \mid not\} < pointcut >)$ |
| *primitive_pointcut* | :: | $< gf\_call > \mid < gf\_return > \mid < get > \mid < set > \mid < create >$ <br> $\mid < m\_return > \mid < m\_call >$ |
| *escape* | :: | $(escape\ ?variable < lisp\text{-}form >)$ |
| *gf_call* | :: | $(gf\text{-}call\ ?gfName\ ?arguments)$ <br> Generic function call join point |
| *m_call* | :: | $(method\text{-}call\ ?methodName\ ?arguments\ ?specializers)$ <br> Method call join point |
| *gf_return* | :: | $(gf\text{-}return\ ?gfName\ ?arguments\ ?rvalue)$ <br> Generic function return join point |
| *m_return* | :: | $(method\text{-}return\ ?methodName\ ?arguments\ ?specializers\ ?rvalue)$ <br> Method return join point |
| *get* | :: | $(slot\text{-}get\ ?obj\ ?slotName\ ?value)$ <br> Slot get join point |
| *set* | :: | $(slot\text{-}set\ ?obj\ ?slotName\ ?oldValue\ ?newValue)$ <br> Slot set join point |
| *create* | :: | $(create\ ?className\ ?instance)$ <br> Instance creation join point |
| *temporal* | :: | $most\text{-}recent \mid all\text{-}past \mid cflow$ <br> Temporal relations |

Figure 2. Grammar for HALO pointcut language. For conciseness we have depicted the arguments of the different predicates as logic variables preceded by a "?".

The advice body consists of a call to the Common Lisp function `print`. The example shows that it is possible to pass logic variables from the pointcut to the advice body. Logic queries, and thus pointcuts, can have multiple solutions with different values for the variables. The advice body is executed for every solution of the query.

## 3.3. HALO Pointcut Language

In a logic pointcut language, pointcuts are expressed as queries using logic predicates. The built-in predicates of HALO fall into two classes: the primitive predicates that distinguish between types of join points, and higher-order temporal predicates for dealing with temporal relationships between join points. The predicates are summarized in Figure 2.

Note that as HALO is a pointcut language for Common Lisp, Lisp-style list syntax is used for logic pointcut queries. Variables are written with a question mark, as in `?var`. For example, the expression `(gf-call 'buy ?args)` would be written in Prolog as `gf-call(buy, Args)`, where variables are written with initial capital letters.

### 3.3.1. Join Point Type Predicates

HALO's join point model, as with most other pointcut languages, consists of the key events in the execution of an object-oriented program. In the case of Common Lisp, there are seven types of join points: the instantiation of a class, the invocation of and return from a generic function, the execution of and return from a method, and the access or change of a slot (instance variable).

Figure 2 lists HALO's join point predicates. They each have a number of arguments exposing data of the join point. The `gf-call` and `method-call` predicates respectively capture invocations of generic functions and executions of specific methods of generic functions. They each expose the arguments the function is invoked with, i.e. the actual runtime objects. The name of the function is exposed as a symbol. The `method-call` predicate has an additional parameter that exposes the specializers of the method, i.e. the argument types specified in the method signature, which are used to select a specific method of a generic function. The corresponding `gf-return` and `method-return` predicates select return join points for generic functions and methods respectively. They have a similar parameter list as the `gf-call` and `method-call` predicates, but additionally expose the return value. The `slot-get` and `slot-set` predicates respectively capture slot access and change join points. They expose the object whose slot is referenced, the name of the slot, its current value, and its new value in the case of `slot-set`. The `create` predicate captures class instantiation join points and exposes the class's name and the new instance.

### 3.3.2. Temporal Predicates

**Temporal predicates overview**

The temporal predicates in HALO allow for pointcuts that express a temporal relation between past join points. This is not limited to join points which are in a control flow relationship. Rather, a history of past join points is kept which can be referred to using the temporal predicates. The temporal predicates are higher-order predicates that take pointcuts as arguments. To establish some terminology, consider the following pointcut:

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

The first condition is referred to as the outer pointcut, and the single condition used as argument to the temporal predicate `most-recent` is referred to as the inner pointcut.

The temporal higher-order predicates share the same basic semantics. An inner pointcut is evaluated against a subset of join points relative to the join points matching the outer pointcut. The actual subset, of course, depends on the particular temporal predicate. In the above example, the inner pointcut `(gf-call 'buy ?argsB)` is thus evaluated against join points in the past of the join points matching the outer pointcut `(gf-call 'checkout ?argsC)`.

In total, HALO has four temporal predicates built-in: `most-recent`, `all-past`, `since` and `cflow`. The `all-past` and `most-recent` predicates match the inner pointcut against *all* past join points relative to the join point matched by the outer pointcut. The predicates differ in that the `all-past` has solutions for all past join points that match, while the `most-recent` predicate only has a solution for the most recent join point that matches. The `cflow` predicate is a variation of the `most-recent` predicate which additionally checks that no corresponding `return` join point has occurred for the join point captured by the inner pointcut (it is therefore similar to the `cflow` construct in AspectJ [17]).

The `since` temporal predicate is the more difficult one of the four predicates as it has two inner pointcuts. The first inner pointcut is evaluated against the past join points relative to the join points captured by the outer pointcut. The second inner pointcut is evaluated against the join points in-between the two other join points. Examples for the use of `since` are given in Section 3.4.

**Variable sharing**

Variables can be shared between the inner and outer pointcuts. As the semantics of the temporal predicates is that the inner pointcut is evaluated against the past of the join point captured by the outer pointcut, variables are bound by the outer pointcut. For example, the following pointcut captures invocations of the `buy` function for a user buying an article, and yields all users that previously also bought the same article:

```
((gf-call 'buy (?user1 ?article))
 (all-past (gf-call 'buy (?user2 ?article))))
```

In this example, the outer pointcut captures a `buy` call and exposes the arguments of the call in the `?user1` and `?article` variables. The inner pointcut then matches on all previous calls to `buy` with the same article object as argument.

### 3.3.3. Hybrid Pointcuts

The `escape` predicate can be used to include Lisp code referring to logic variables in a pointcut definition. The example below shows a pointcut capturing invocations of a generic function named `buy`, where the `escape` predicate is used to ask the price of an article (second argument) and to bind the result to a logic variable `?price` (first argument). Whenever such a pointcut is evaluated, the piece of Lisp code is executed using the bindings available for the logic variables, resulting in a new variable binding which in return is used in the evaluation of the rest of the pointcut. However, if the return value of the Lisp code is `nil`, the condition has no solution.[2] In the example, the constraint `(greater-than ?price 10)` is checked for a value `?price` computed at the Lisp level – or in other words, the binding for `?price` is not logically derived.

```
((gf-call 'buy (?user ?article))
 (escape ?price (price ?article))
 (greater-than ?price 10))
```

The `escape` predicate can be in a temporal predicate, but a restriction on the variables that can be used in its condition applies: Only variables that are used in non-`escape` conditions in the same inner pointcut can be used. This is because the Lisp code of the `escape` conditions inside the `most-recent` is evaluated when `user2` buys an article. The following piece of advice is triggered to print the price of an article bought by a user, and its price when previously bought by another user:

[2]The value `nil` also denotes false in Lisp.

```
1 (gf-call 'login <kris> <shop>)
2 (gf-call 'buy <kris> <dvd>) where the current discount rate for <dvd> is 0.05
3 (gf-call 'checkout <kris>)
4 (gf-call 'login <kris> <shop>)
5 (gf-call 'buy <kris> <game>) where the current discount rate for <game> is 0.05
6 (gf-call 'buy <kris> <book>) where the current discount rate for <book> is 0.10
7 (gf-call 'buy <kris> <cd>) where the current discount rate for <cd> is nil
8 (gf-call 'checkout <kris>)
```

Figure 3. A sample history of join points (to simplify the example, only generic function calls are considered).

```
(at
    ((gf-call 'buy (?user1 ?article))
     (most-recent
        (gf-call 'buy (?user2 ?article))
        (escape ?price2 (price ?article))
        (escape ?name2 (user-name ?user2))))
 (print "Article previously bought by " ?name2 " for " ?price2 " EUR"))
```

So the variable `?price2` will refer to the past price of the article, which is possibly different from the price of the article when the second buyer purchases the article.

## 3.4. Further Examples

To clarify the way the temporal predicates are matched, we give a few further examples based on the sample execution trace shown in Figure 3: Note that we use the notation `<`*name*`>` to denote object identifiers (e.g. `<cd>` represents an object, obviously intended to be an instance of `Article`). For more complex examples, we refer the reader to the implementation of a realistic web shop application using HALO [15].

Given the sample execution history depicted in the figure, the following pointcut matches on join point 8 with one solution:

```
((gf-call 'checkout ?user)
 (most-recent (gf-call 'buy (?user ?article))
   (escape ?rate (current-discount-rate-for (singleton-instance 'promotions) ?article))))
```

For the match with join point 8, the solution gives the article the user `<kris>` last bought, and the discount rate at the time the article was bought. Given the execution history in Figure 3, this means the exposed discount rate is 0.10 for the article `<book>`. This pointcut captures join point 8 because it matches the outer pointcut `(gf-call 'buy ?user ?article)` and because of the presence of join point nr. 6 that matches the inner pointcut. Join point 7 does not match the inner pointcut, as the Lisp form in the `escape` condition evaluates to `nil`.

The following pointcut has multiple solutions for join point 8, one for each article the user checking out ever bought and for which there was a promotion (the articles `<book>` and `<game>` and `<dvd>` bought by user `<kris>`):

```
((gf-call 'checkout ?user)
 (all-past (gf-call 'buy (?user ?article))
   (escape ?rate (current-discount-rate-for (singleton-instance 'promotions) ?article))))
```

Our last example is a variation of the above one. Again, it has multiple solutions for the match with join point 8, but only those articles bought since the last `login` (only the articles `<book>` and `<game>` purchased by `<kris>`):

```
((gf-call 'checkout ?user)
 (since
   (most-recent (gf-call 'login (?user ?shop)))
   (all-past (gf-call 'buy (?user ?article))
     (escape ?rate (current-discount-rate-for (singleton-instance 'promotions) ?article)))))
```

In more detail, this pointcut is matched at join point 8, because it matches the outer pointcut `(gf-call 'login (?user ?shop))`, and exposes the discount rate of all `buy` join points (namely 5 and 6) that match the second argument of the since predicate, because the last `login` join point that matched the first argument of the since predicate (join point 4). Note that the `buy` join points and the `login` join point are again matched in the past of the `checkout` join point.

## 3.5. Defining Rules

Programmers can define rules for new predicates using the `defrule` construct. As in other logic-based pointcut languages [13,21], this mechanism can be used to define new join point predicates. This is simply a matter of using an existing join point predicate in the definition of the rule. For example, the rule definition below extends HALO with a new pointcut predicate that captures invocations of a generic function called `checkout`:

```
(defrule (checkout-gf-call ?args)
  (gf-call 'checkout ?args))
```

Note that rules do not have to define predicates about join points. Only rules based on other join point predicates define a new join point predicate. This is unlike the named pointcut mechanism in AspectJ, for example, in which the conditions of a named pointcut always have to include a primitive or user-defined pointcut.

## 4. HALO Implementation

In this section we present the implementation strategy for evaluating HALO pointcuts. We present the overall architecture of the weaving process which involves a runtime weaver for intercepting join points and a query engine for checking pointcuts for matches. We contrast the use of backward and forward chaining query engines for supporting HALO to compare with related work on logic-based pointcut languages and to demonstrate why forward chaining is necessary. This is followed by an extensive discussion of the Rete forward chaining algorithm and the necessary extensions for supporting HALO's temporal predicates and `escape` mechanism. Finally, we discuss how the semantics of the predicates is exploited to optimize the join point history, so that join points are removed from the history when they are no longer relevant for matching pointcuts.

## 4.1. HALO Weaver Architecture

A schema of the dynamic weaving process, responsible for combining HALO code and base code, is depicted in Figure 4. Note that in this schema, we assume a sequential execution of the base program – a version of HALO for concurrent programming is left

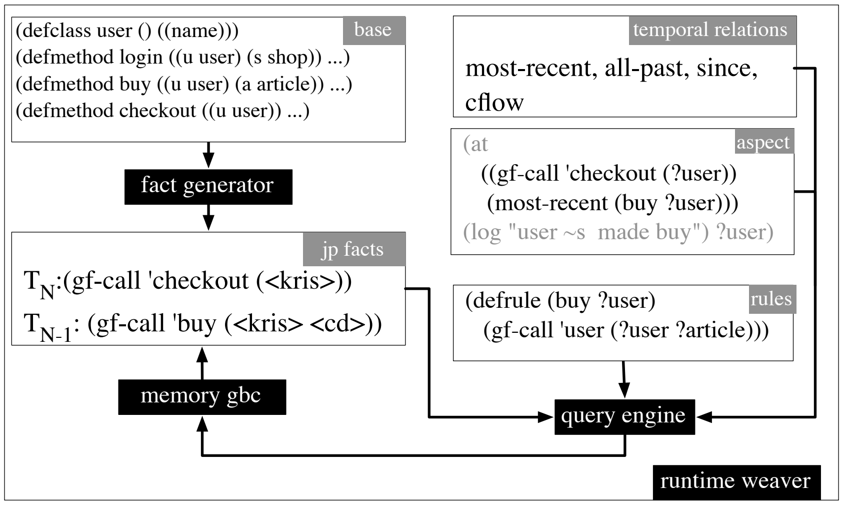

(defclass user () ((name)))
(defmethod login ((u user) (s shop)) ...)
(defmethod buy ((u user) (a article)) ...)
(defmethod checkout ((u user)) ...)
base
fact generator
jp facts
$T_N$:(gf-call 'checkout (<kris>))
$T_{N-1}$: (gf-call 'buy (<kris> <cd>))
memory gbc
temporal relations
most-recent, all-past, since, cflow
aspect
(at ((gf-call 'checkout (?user)) (most-recent (buy ?user))) (log "user ~s made buy") ?user)
rules
(defrule (buy ?user) (gf-call 'user (?user ?article)))
query engine
runtime weaver

Figure 4. HALO weaver schema.

for future work. The weaver is responsible for mapping the key events in the execution of a Common Lisp program to logic facts and storing them in a fact base. In our concrete implementation, this is achieved by wrapping the generic function call, instance creation and slot access protocols in Common Lisp through the CLOS Metaobject Protocol [16] to attach code for generating the facts. Secondly, the weaver is responsible for weaving in the proper advice code at each event. The proper advice code is computed by trying to resolve the pointcuts given the fact base. The latter is done by a *query engine*, which is basically an execution engine for our logic language HALO. In the following sections, we outline the decisions made for implementing the HALO query engine. Another issue we examine in the follow-up text involves optimization strategies for memory management of the fact base, a common problem in history-based logic pointcut languages.

## 4.2. Implementing the Query Engine

We provide a more detailed discussion of the differences between forward and backward chaining [22] to compare with related work on logic-based pointcut languages (see Section 5). Most current logic-based pointcut languages are based on Prolog, which is in turn based on backward chaining. We contrast these two approaches for evaluating logic queries and discuss why forward chaining is necessary to support a combination of hybrid pointcuts and reasoning over a history of join points as in HALO.

The two approaches to chaining can best be contrasted by representing a logic query graphically, as in Figure 5 depicting the following piece of advice. When a user checks out, he gets a discount on the total amount purchased, and that discount is based on a rate that was promoted when the user logged into the shop:

```
(at
    ((gf-call 'checkout (?user))
     (most-recent (gf-call 'login (?user ?shop))
                  (escape ?rate (current-rate ?shop))))
```

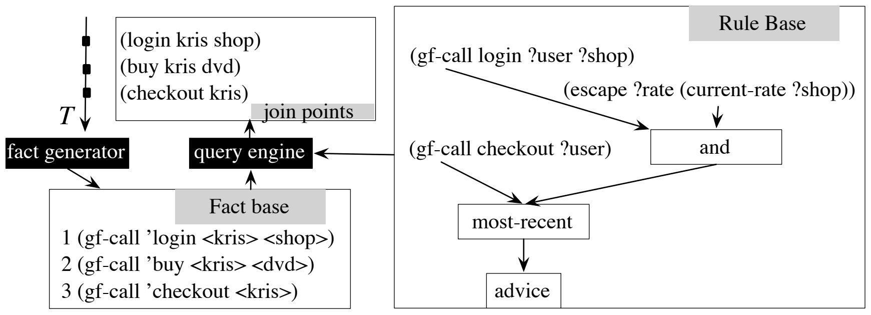

(login kris shop)
(buy kris dvd)
(checkout kris)
join points
T
fact generator
query engine
Fact base
1 (gf-call 'login <kris> <shop>)
2 (gf-call 'buy <kris> <dvd>)
3 (gf-call 'checkout <kris>)
Rule Base
(gf-call login ?user ?shop)
(escape ?rate (current-rate ?shop))
(gf-call checkout ?user)
and
most-recent
advice

Figure 5. Execution of a program represented as facts. HALO pointcut represented as a tree.

```
  (discount ?user ?rate))
```

The left upper corner of Figure 5 depicts a sample program run. As explained in the previous section, the weaver records facts for each join point. The resulting fact base is also depicted in Figure 5.

When using a weaver with a backward-chained query engine, a problem arises with evaluating `escape` conditions at the right time: At every join point, the weaver produces logic facts for describing that join point and then invokes the query engine to check if any pointcuts match. In the example, when using a backward chainer, the weaver would launch the pointcut as a query at every join point. In backward chaining, a logic query or pointcut is evaluated by finding rules to evaluate the conditions in the pointcut, and recursively finding rules for the conditions in those rules. In other words, using the graphical representation, the query is evaluated from the bottom to the top. The process stops when it can find logic facts for all of the conditions, meaning the query or pointcut follows logically from the facts. Resolving this query using backward chaining results in a bottom-up traversal of the tree depicted in rule base of Figure 5. In order to resolve the query, a `most-recent` relation must hold between the result of resolving the left-input and the right-input of the node labelled `most-recent`. This requires searching the fact base for a fact that matches the pattern `(gf-call 'checkout (?user))`, another fact that matches the pattern `(gf-call 'login (?user ?shop))` and, given those bindings, resolving the `escape` condition by executing its piece of Lisp code. Escape conditions make it possible to expose context from the base program (see Section 3.3.3). When combined with temporal operators, escape conditions expose *past* program context. Coming back to our example, this means the `escape` condition should be evaluated in relation to the program state when the `login` join point occurred. In other words, the promotional rate exposed via `?rate` through the `escape` predicate needs to be bound to the discount rate active at login time. However, backward chaining does not support such semantics. At any time between the `login` join point and resolving the pointcut, the state of the object `<shop>` might have changed.

Supporting the evaluation of `escape` conditions at the right time fits better in the model of forward chaining, particularly the Rete forward chaining algorithm. When using a forward-chained query engine, the relationship between weaver and query engine is reversed. Rather than the weaver invoking the query engine to check if a pointcut matches, the query engine responds to changes in the fact base and informs the weaver if there are any new matches for pointcuts. In the Rete forward chaining algorithm, a representation for pointcuts similar to the one in Figure 5 is used. This representation is extended with memory. The memory serves to remember partial matches for pointcuts. Overall, the algorithm works as follows: When a fact is inserted in the fact data base, find all rules for which the fact matches a condition and try to resolve the rule given the fact base at that time. In addition, the algorithm records all conclusions found in-between in the fact base. So in the example depicted in Figure 5, when the fact `(gf-call 'login <kris>)` is inserted in the fact base, the `escape` condition of the rule depicted in the same figure is evaluated and asserted in the fact base. At a later time, when the fact `(gf-call 'checkout <kris>)` is asserted, it is combined with the solution memorized for the match to the partial pointcut `((gf-call 'login (?user ?shop)) (escape ?rate (current-rate ?shop)))`. Note that the `escape` condition is thus evaluated at the time the join point happens that matches this part of the pointcut, thus implementing the HALO semantics. In the next section, we discuss how the Rete algorithm is further extended to support HALO's temporal operators and discuss how its apparent drawback on memory usage can be optimized in HALO.

## 4.3. Temporal Extensions to Rete

In this section, we discuss how the Rete algorithm can be extended to support HALO's temporal predicates and its `escape` mechanism. We begin by briefly discussing the standard Rete algorithm.[3]

### Basic Rete

Rete represents rules – or pointcuts in HALO – as a network of nodes with memory tables. For each condition in a rule, the network contains a "filter" node. For each logical "and" between conditions in rules, it contains a "conjunctive" join node. When new facts are added, they are inserted in the filter nodes. A filter node checks whether the fact unifies with its condition, and if so, memorizes it in its memory table and notifies the join node that it is linked to. For example, the filter node for the condition `(gf-call 'checkout ?x)` will memorize a fact `(gf-call 'checkout <pascal>)` but not a fact `(gf-call 'login <pascal>)`. Conjunctive join nodes have an incoming link from one filter node and another join node or filter node. When a conjunctive join node is notified that a new fact was memorized, it checks whether this fact matches with the facts memorized by the other incoming node. Specifically, it checks whether they have the same values for common variables. If this is the case, this combination is memorized and the next join node is notified.

As an example, consider the query given below. The Rete network for the query is shown in Figure 6. The figure shows the state of the memory tables after adding the fact

[3]We discuss the original Rete algorithm. Improved versions, like Rete II and Rete III exist, but unlike the original Rete, are proprietary algorithms that have not been published.

shown on the left of the figure. Note that the notation <*name*> is again used for an object identifier (a reference in the actual implementation).

```
((gf-call ?operation (?arg1 ?arg2))
 (gf-call 'browse (?arg1 ?arg2))
 (gf-call ?operation (1 ?arg2)))
```

As the rule has three conditions, the Rete network contains three filter nodes (the circles). These filter nodes are connected to one another by means of conjunctive join nodes (the squares). The bottom node (the triangle) is a query conclusion node. When it is notified of new facts, it means a solution for the query was derived. This is the case in this example, where the bottom node is triggered for the match where `?operation` has the value `browse`, `?arg1` the value 1 and `?arg2` the value `<lotte>`.

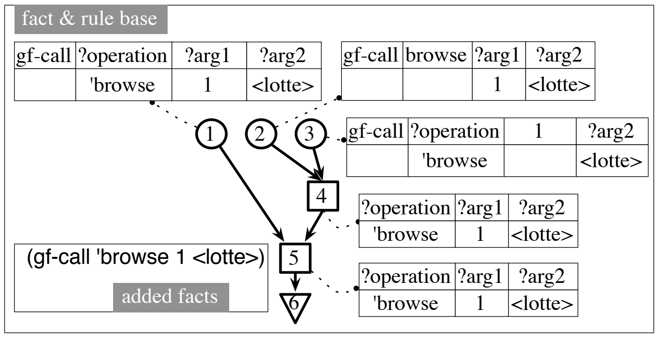


Figure 6. A standard Rete network.

## Temporal frames in memory tables

In standard logic, any fact is unambiguously "true." In temporal logic however, facts are true in a temporal frame, a certain moment in time.[4] To support this, memory tables are extended to record in which temporal frame their entries are considered true. In Figure 7, this is the gray column in the left of memory tables.

## Temporal join nodes

Supporting the temporal operators in Rete is done by introducing new types of join nodes. One new type of join node is added for each of the temporal operators `most-recent`, `all-past`, `since` and `cflow`. When temporal join nodes are notified of new incoming facts, they combine the new facts with those in the memory table of its other incoming node, similar to conjunctive join nodes in regular Rete. The join nodes in the

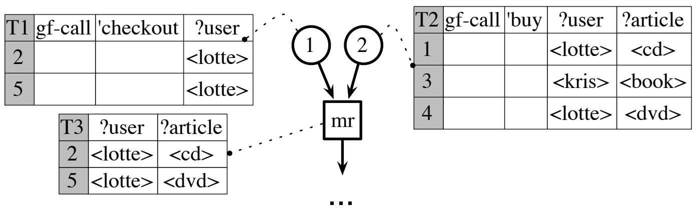
[4]The term "temporal context" is used in literature, but we use "temporal frame" to avoid confusion with "context" in the sense of join point context data.

extended Rete are restricted to combining entries that meet constraints on the temporal frames. Figure 8 displays the different temporal constraints for the different types of join nodes: $T_{left}$ and $T_{right}$ refer to the temporal frames associated with the outer and inner pointcut respectively (which are always depicted as respectively the left and right inputs of the temporal join node). In the case of the `since` operator, $T_{left}$ refers to the outer pointcut, $T_{right}$ points to the second argument of `since` and $T_{middle}$ points to the first argument. The behavior of the `most-recent` and `all-past` join nodes further differs in that an `all-past` passes all matches to its output node, while a `most-recent` join node only passes one match. Specifically, when a new entry is made in its left input node, it tries to match it with the entries in its right memory table, starting from the most recent entry, and only passes the first successful match.

For example, Figure 7 shows the network for the following pointcut:

```
((gf-call 'checkout (?user))
 (most-recent (gf-call 'buy (?user ?article))))
```

The network consists of one temporal join node for the `most-recent` condition, its left input is a filter node for the one condition in the outer pointcut `(gf-call 'checkout (?user))`, and its right input is a filter node for the one condition in the inner pointcut `(gf-call 'buy (?user ?article))`.



| T1 | gf-call | 'checkout | ?user |
|---|---|---|---|
| 2 | | | <lotte> |
| 5 | | | <lotte> |

| T2 | gf-call | 'buy | ?user | ?article |
|---|---|---|---|---|
| 1 | | | <lotte> | <cd> |
| 3 | | | <kris> | <book> |
| 4 | | | <lotte> | <dvd> |

| T3 | ?user | ?article |
|---|---|---|
| 2 | <lotte> | <cd> |
| 5 | <lotte> | <dvd> |

Figure 7. Rete containing a square-shaped temporal join node.

Figure 7 also shows the state of the memory tables after processing the following series of fact insertions. At time 1, a fact `(gf-call 'buy <lotte> <cd>)` is inserted and memorized which matches only the right filter node. The join node is notified, but as the memory table of its left input is empty, it does not do anything. At time 2, a fact `(gf-call 'checkout <lotte>)` is inserted and memorized which matches only the left filter node. As the join node is notified, it combines the new entry of its left input with the most recent matching entry in the memory table of the right input node. The entries match if they have the same values for the common variables and the constraint $T_{right} < T_{left}$ between the temporal frames of the entries is met. The entry that was made at time 1 in the right filter node matches, as it also has the object `<lotte>` as value for the variable `?user`. Thus, this combination is memorized in the join node's memory table. At time 3, the

- `and`: $T_{left} = T_{right}$
- `most-recent` $T_{right} < T_{left}$
- `all-past`: $T_{right} < T_{left}$
- `since`: $T_{right} > T_{middle}$ and $T_{right} < T_{left}$ and $T_{middle} < T_{left}$

Figure 8. Constraints in temporal join nodes.

fact `(gf-call 'buy <kris> <book>)`, and at time 4, the fact `(gf-call 'buy <lotte> <dvd>)` is inserted. At time 5, the fact `(gf-call 'checkout <lotte>)` is inserted. It matches the left filter node, so the join node is notified. The join node combines the new entry with the right filter node's memory table. Two combinations are possible: one with temporal frame 4 and one with temporal frame 1, because both have the object `<lotte>` as value for `?user`. However, due to the recent matching semantics of the `most-recent` operator, only the combination with the entry of temporal frame 4 is made. A new entry is thus made in the join node's memory table which is true at temporal frame 5, with the object `<dvd>` as value for the variable `?article`. Conversely, were the operator `most-recent` replaced by the operator `all-past`, all matching combinations would have been memorized.

### Control flow join nodes

Control flow join nodes operate slightly differently from `most-recent` join nodes. Figure 9 shows the Rete network for the following pointcut:

```
((gf-call 'update-line ?args1)
 (cflow (gf-call 'update-figure ?args2)))
```

The memory table for a `cflow` join node's right input node is extended to record the time at which the return of the captured join point occurs. When a return join point is encountered, the weaver notifies control flow join nodes of the time of the corresponding invocation join point. The nodes that have an entry for that time in their right input node's memory table add the time at which the return occurred. The entry will no longer be used to make combinations with entries coming from the left node.

Figure 9 reflects the Rete after the insertion of the following conclusions. At time 1, a call to the generic function `update-figure` occurs and a fact `(gf-call 'update-figure <fig1>)` is inserted in the network, making an entry in the memory table of the first filter node. Immediately thereafter, the weaver detects the return of that same generic function call and notifies the control flow join node: The return time, namely 2, is added to the entry for the generic function call in the memory table of the temporal join node's right input node. If subsequently at time 3, a fact `(gf-call 'update-line <line1>)` is added, this is memorized in the first filter node and the control flow join node is notified. However, as the control flow join node cannot find an unfinished generic function call entry in its right input node for which the constraint $T2 < T1$ succeeds, it cannot memorize a conclusion. Assume that next, at time 4, the fact (gf-call 'update-figure <fig2>) and at time 5, the fact (gf-call 'update-line <line2>) are inserted in the network. This time, when the temporal join node is notified, it is able to derive a conclusion as it can combine the entry memorized at time 4 in its left input node with the entry memorized at time 5 in its right input node. As such, the Rete network concludes that the invocation of the generic function update-line with argument <line2> is in the control flow of the call to the generic function update-figure with an argument <fig2>.

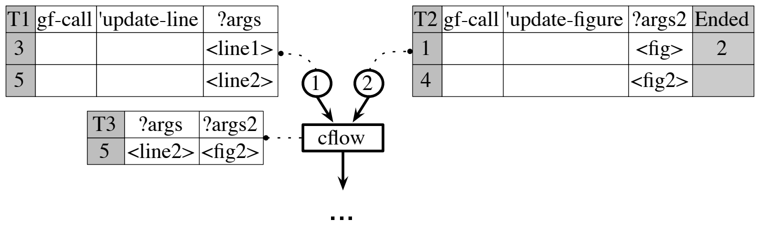


Figure 9. Rete containing a control flow join node.

### Escape nodes extension

Another extension to the Rete algorithm is used to handle the escape conditions. Such conditions are represented as nodes in the Rete network similar to join nodes, though they only have one input. Figure 10 shows the Rete network for the following pointcut:

```
((gf-call 'checkout (?user))
 (most-recent
   (gf-call 'buy (?user ?article))
   (escape ?rate (current-discount-rate (singleton-instance 'promotions) ?article))))
```

When an escape filter node is notified of a memorization in its input node, the Lisp form is executed after all logic variables are replaced by the values from the received notification. Subsequently, if the result of this evaluation is different from nil, it is memorized in the escape filter node and the escape filter node's output node is notified. For example, if a fact (gf-call 'buy <kris> <book>) is inserted in the network from Figure 10, the escape filter node is notified and evaluates the Lisp form (current-discount-rate (singleton-instance 'promotions) <book>).

## 4.4. Memory Table Garbage Collection

It is not very economical to keep all the entries in the memory tables in the Rete network during the entire run of the program. Due to the semantics of the temporal predicates, certain entries in the memory tables can become irrelevant as they will never produce new combinations. This information can be exploited to provide automatic garbage collection of superfluous entries. However, removing join point facts from the history also implies

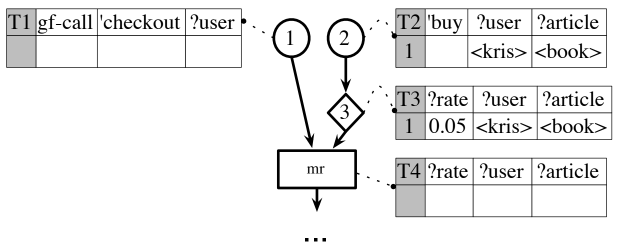


Figure 10. Rete containing an diamond-shaped escape filter node.

that they cannot be used anymore to match pointcuts added dynamically. Nonetheless the considerable improvements with regard to memory usage favor the use of our garbage collection strategy.

**Entries no longer most recent**

Entries in the right input memory table of `most-recent` temporal join nodes can be removed when new entries with the same values for the variables are added. In fact, only the values for the variables that are in common with the left input memory table need to be the same. This is because when an entry is added to the left input's memory table, the join node will combine it with the most recent matching entry in the right input node. The match requires that the values for the variables that the two input nodes have in common are the same. Thus, if there is an older entry in the right memory table that also matches with the new entry in the left, it will still not produce a combination. Thus, such entries can be removed.

Consider the example of Figure 7 discussed previously. At time 4, an entry is made in the memory table of the join node's right input node. The entry of time 1 can then be removed because it has the same value for the variable `?user`. Note that the values for the variable `?article` are different, but this variable is not used in the join node's left input node.

Figure 11 gives an example with nested `most-recent` predicates for the following piece of advice:

```
(at ((gf-call 'checkout ?user1)
     (most-recent (gf-call 'checkout ?user2)
       (most-recent (gf-call 'buy ?user2 ?article2))))
  (print ?user2 " just bought " ?article2))
```

A sample program run is depicted in the same figure. In addition, the figure displays tables labelled `LT` (life time): The intervals stored by these tables indicate the begin and end point for the interval during which entries in the memory tables are kept. Note that though the entries in the third filter node are removed as new entries are made, the derived conclusions are not also removed at the same time: At time 7, for example, when the entry made for `(gf-call 'buy <lotte> <dvd>)` is removed, the derived conclusion for time 5 in the first `most-recent` join node is kept. This ensures that at time 8, it can be used to match the pointcut. However, this does not mean the derived conclusion is kept forever. The first `most-recent` join node is itself the input of another `most-recent` join node. The input nodes of this second join node share no variables. So the entry for time 5 in the output memory table of the first join node is removed when any other entry is made, which in this example will happen the next time a user checks out if he bought something (e.g. if the user `lotte` does another checkout).

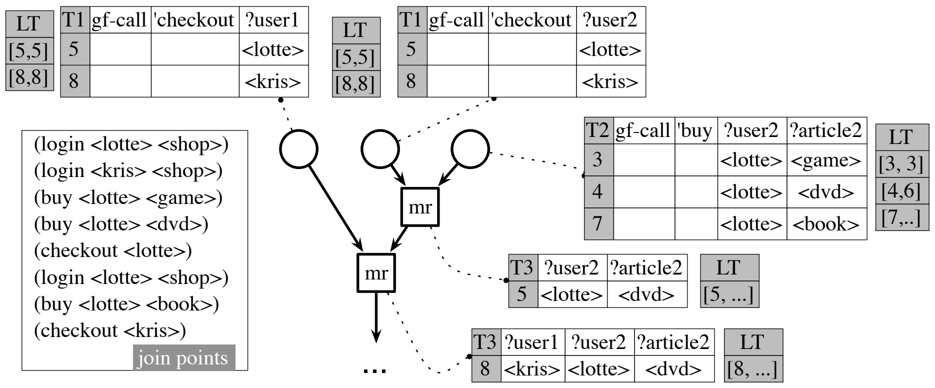


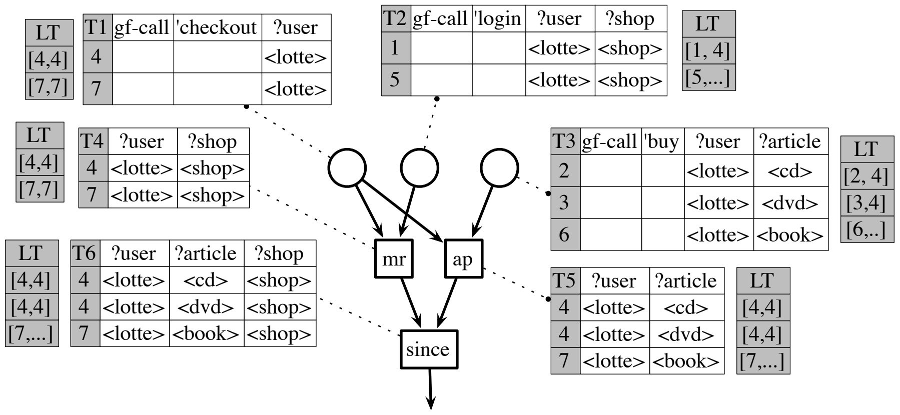
Figure 11. Garbage collection of nested temporal join nodes.

## Combinations of `since` and `most-recent`

When the first argument pointcut of a `since` condition is a `most-recent` condition, the memory tables for the second argument pointcut's network can be garbage collected whenever entries are removed from the `most-recent` node's memory table. For example, consider the Rete network shown in Figure 12 for the following piece of advice that makes sure a discount is given for each article bought during a single shopping session:

```
(at ((gf-call 'checkout (?user))
     (since (most-recent (gf-call 'login (?user ?shop)))
       (all-past (gf-call 'buy (?user ?article)))))
  (discount ?article (current-discount-rate (singleton-instance 'promotions) ?article)))
```

Intuitively, in this pointcut, the join points in the history for the buy calls of a user can be removed once she logs in again. This is illustrated for the sample program depicted in the figure, and the Rete network is shown after the execution of the entire program. When the second call `(login <lotte> <shop>)` at time 5 happens, the entry labelled 1 is removed in the right input node of the `most-recent` join node (table T2). This also implies that we can safely remove all entries memorized in the right input network of the `since` join node, which have the same binding for the variable `?user`, namely `<lotte>`, which were made before time 5. This is because the temporal constraint of the `since` temporal join node, which is $T_3 > T_2$ in Figure 12, will never be fulfilled for those entries anymore.



| LT | T1 | gf-call | 'checkout | ?user |
|---|---|---|---|---|
| [4,4] | 4 | | | <lotte> |
| [7,7] | 7 | | | <lotte> |

| T2 | gf-call | 'login | ?user | ?shop | LT |
|---|---|---|---|---|---|
| 1 | | | <lotte> | <shop> | [1, 4] |
| 5 | | | <lotte> | <shop> | [5,...] |

| LT | T4 | ?user | ?shop |
|---|---|---|---|
| [4,4] | 4 | <lotte> | <shop> |
| [7,7] | 7 | <lotte> | <shop> |

| T3 | gf-call | 'buy | ?user | ?article | LT |
|---|---|---|---|---|---|
| 2 | | | <lotte> | <cd> | [2, 4] |
| 3 | | | <lotte> | <dvd> | [3,4] |
| 6 | | | <lotte> | <book> | [6,..] |

| LT | T6 | ?user | ?article | ?shop |
|---|---|---|---|---|
| [4,4] | 4 | <lotte> | <cd> | <shop> |
| [4,4] | 4 | <lotte> | <dvd> | <shop> |
| [7,...] | 7 | <lotte> | <book> | <shop> |

| T5 | ?user | ?article | LT |
|---|---|---|---|
| 4 | <lotte> | <cd> | [4,4] |
| 4 | <lotte> | <dvd> | [4,4] |
| 7 | <lotte> | <book> | [7,...] |

Figure 12. Rete network illustrating automatic garbage collection for `since`.

### Nodes without memory

Not all nodes need to keep a memory table. The exceptions are: nodes that are the left input of a temporal join node (thus not conjunctive join nodes), the last node that triggers the advice code, and nodes that are the input of an escape node. Keeping a memory table for the left input of a temporal join node is not necessary: new entries coming in from the right conceptually need to be matched with entries from the left, but they can only match if the left entries are in a temporal frame which is in the future of the right entry. Due to the order in which facts are added by the weaver, such entries cannot exist yet. An escape node never consults the memory of its input node, rather whenever new facts come in through its input, it executes Lisp code and records the result in its own memory table. Thus the input node of an escape does not actually need to keep a memory table. The last node that triggers the advice code does not need to keep a memory table: these are never joined to other nodes for deriving conclusions.

As an example, the Rete network of Figure 10 is depicted again in Figure 13, annotated with the life time of facts. The two filter nodes no longer have a memory. The second figure shows the network after a different series of insertions: the facts `(gf-call 'buy <lotte> <book>)` at time 1 and `(gf-call 'buy <lotte> <cd>)` at time 2. When the first fact is inserted, assuming the discount rate of the `<book>` is 0.05, this is memorized in the escape node . Similarly, as the second conclusion is inserted and if the discount rate of the `<cd>` object is 0.10, this is also memorized. In addition, the previous conclusion of time 1 memorized in the escape node is deleted, as this conclusion will not be used anymore to derive conclusions by the most-recent join node.

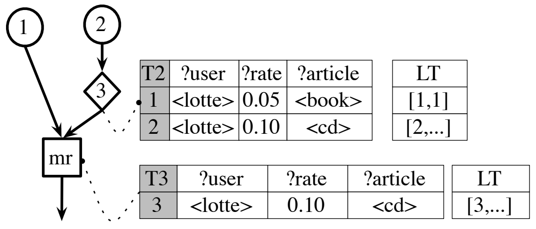


Figure 13. Rete network from Figure 10 annoted with life time of facts for garbage collection.

**Entries marked as no longer used**

Obviously, in the case of a `cflow` join node, the entries that are marked as “no longer used” can in fact simply be removed. Note that this means that older entries in the memory table can become the most recent match again. This is why `cflow` nodes are an exception to the first case for garbage collection described above.

## 4.5. Evaluation on E-Shop Example

Table 1 shows the results of an evaluation of the garbage collection of memory tables. The evaluation was performed using an e-commerce application based on the running example of this paper (Section 2). The application was reported on in our earlier work [15] and is implemented in Lisp using the Hunchentoot web application framework [26]. The aspects installed on the application include one for giving discounts, another for popping up banner advertisements as a customer logs into the shop, and an additional aspect which implements a recommendation system as found on sites such as Amazon (i.e. “other users who bought this also bought ...”). The application was run with scripts simulating a series of users logging in, buying articles and checking out.

Table 1 shows statistics on the number of memory table entries for three scripts. The scripts differ in the number of operations done by the simulated users. The column labeled JPF shows the total number of join point facts produced by the weaver during the entire run of the application. The column labeled TFNE shows the total number of filter node entries that were created during the run of the application. Note that there is a large difference between the two numbers. Many join points were intercepted that did not match any conditions of any pointcut at all, and hence were all rejected by the filter nodes implementing these conditions. Hence no information about these join points is stored at all. The large difference between the numbers stems from the fact that, as further discussed in Section 4.6, our implementation of HALO does not currently employ any static analysis techniques, such as shadow weaving, to limit the amount of intercepted join points (see future work).

The column labeled TJNE shows the total number of entries made in the join nodes, in other words, the number of partial derivations. This number is naturally much larger than the number of filter node entries, as each entry from either input node of the join

| **JPF** | **TFNE** | **TJNE** | **RFNE** | **RJNE** | **RTH (s)** | **RT (s)** |
|---|---|---|---|---|---|---|
| 5948 | 294 | 9663 | 164 | 329 | 2.680 | 0.116 |
| 11799 | 378 | 17803 | 158 | 329 | 5.036 | 0.156 |
| 28853 | 731 | 49800 | 156 | 339 | 31.137 | 1.494 |

Table 1
Benchmarks for memory table garbage collection on a Lisp E-commerce application.

node can be combined with several other ones from the other input node.

The columns RFNE and RJNE show the number of remaining entries at the end of the application run in respectively the filter nodes and join nodes. All other entries have been already removed at some point. The remaining entries are those that could still be needed for matching pointcuts if the application continued to run. As shown, many of the filter node entries, and particularly very many of the join node entries, are removed. The entries that remain at the end of the run remain fairly constant and consist mostly of entries that, because of the pointcuts used in this application, need to be remembered for the entire run of the application anyway.

Finally, the columns RTH and RT show an average runtime of the application for each script, respectively with aspects installed (RTH), and without any aspects installed and HALO fully deactivated (RT). This shows that our current implementation of HALO causes a large runtime overhead. This is in part because of the prototypical nature of the implementation. Our implementation of the Rete algorithm is currently designed for extensibility rather than efficiency. For example, it does not employ any compilation to machine form as described in Forgy's original work [11]. Also, as explained above, many join points are generated and then matched against all filter nodes. We expect a static analysis technique using shadow weaving to improve this, as explained in the next section.

To summarize, we can learn the following from Table 1: Our current implementation generates a high number of join point facts that are not recognized by any filter nodes. This number can potentially be reduced by employing static analysis techniques, such as shadow weaving. This should also reduce the runtime overhead, as discussed below.

On the other hand, there is also a relatively high number of entries in join nodes created during runtime, which are conceptually necessary and cannot be reduced by further optimization techniques – the possible combinations of input nodes simply induces this level of complexity. However, as the distinctive feature of our approach, many filter node entries are effectively removed at runtime when they become obsolete, keeping the total number of entries to a reasonably low size.

## 4.6. Further Optimization Strategies

The memory table garbage collection optimization strategy outlined in the previous sections is orthogonal to the shadow weaving optimization strategy performed in other logic-based pointcut languages [13,21]. In that optimization, which is based on abstract interpretation and partial evaluation, the pointcuts are statically analyzed to determine which join points never affect the matching of a pointcut so that the weaver does not need to intercept these join points. That optimization can be adapted to HALO as well:

For each shadow point one can record which filter nodes the join points from that shadow potentially match against. In many cases, this will actually be zero filter nodes, and thus no join point fact needs to be generated at all. In other cases, only a small subset of the filter nodes will have to be checked. In any case, we note that the shadow weaving optimization strategy is orthogonal to optimizing the join point history dynamically. Shadow weaving optimizes the join point history so that join points that are *never* relevant are not intercepted. The technique discussed in this paper optimizes the join point history so that join points that *are* relevant are removed when they are *no longer* relevant. This is further explained in the related work section in comparison with Alpha.

## 5. Related Work

### OReA & Hybrid Pointcuts

We have simplified the "hybrid pointcuts" mechanism [9] in this paper to an explicit `escape`. In the work of D'Hondt on OReA [9], the goal of "hybrid aspects" is to be transparent: A condition in a logic pointcut can be re-defined as a method, and vice versa. The pointcut language and base language are redefined so that when no rule is defined for a logic condition, the condition will be evaluated by sending a message instead. This can be easily achieved in HALO as well: If no rule exists for a logic condition, it is translated to an `escape` condition. However, we have not demonstrated this in this paper in order to focus on the issue of supporting "hybrid pointcuts" in a language like HALO that supports pointcuts over a history of join points. OReA also supports interaction from the base and advice languages with the rule language, which we have not considered in HALO so far. OReA is actually a family of logic pointcut languages, which includes a variant based on forward chaining. However, that variant of OReA is not based on the Rete network and lacks the necessary support for memorizing past evaluations of hybrid pointcuts. While OReA supports hybrid pointcuts in both directions in a transparent manner, it does not support pointcuts over a history of join points.

### Alpha

A closely related approach to our work is Alpha [21], a logic-based pointcut language for expressing pointcuts over a history of join points. Alpha includes information about the state of objects and the static structure of the program in the fact base. Full Prolog can be used to write pointcuts as logic queries over the historic fact base. A pre-defined set of logic rules for expressing temporal relations is provided, but this can be extended by the programmer. While Alpha also has a mechanism for letting the pointcut language interact with the base program, as discussed in Section 4.2, the use of standard Prolog only allows interaction with the base program at the current join point. So (as discussed in Section 2), this means the "past rate discounting" aspect must be expressed as two pointcuts and pieces of advice. Thus, while Alpha is more expressive than HALO in terms of providing a richer join point model and the use of full Prolog to reason about the past history of join points, it is also less expressive with regard to the extent to which hybrid pointcuts can interact with the base program. Because of the open set of temporal predicates, partial evaluation of pointcuts is used to optimize the memory required to keep the historic fact base. The analysis is done statically and determines which join points may possibly affect pointcuts. For these, the shadow weaving technique well-known in aspect weaver construction is applied so that only those join points are actually intercepted. A similar technique can be used in HALO, though this is an area of future work. In Alpha, the join points that are intercepted are kept in the fact base indefinitely, except if the static analysis can determine that they are only used in matching pointcuts as the current join point and not as past join points. Thus, if a pointcut expresses the equivalent of HALO's `most-recent` predicate, information about all join points matching the `most-recent` condition is kept indefinitely. In contrast, in HALO, the set of temporal predicates is fixed, which means the implementation knows about the semantics of the predicates, which is exploited to perform a dynamic analysis of the fact base so that matches are removed from memory tables if they are no longer relevant.

### Context-Aware Aspects in Reflex

Reflex is a kernel for multi-language aspect-oriented programming, implemented as an object-oriented framework. In earlier work, the framework was extended with the necessary support for context-aware aspects, which also allows embedding base code in pointcuts, as well as referring to past join points [24]. In that framework, context definitions can be implemented as objects with a method that indicates whether the context is active. The proposed pointcut language does not allow pointcuts over past join points. Rather, the framework provides support for defining "context restrictors" that can be used in a pointcut to restrict it not just based on the current join point but also on past activations of a context, for example, depending on whether a context was active during the creation of the object in which the current join point occurs. Internally, these restrictors add additional pointcuts and advice to the program to capture the state of the context objects for later reference. In HALO, this splitting of pointcuts into parts that are evaluated at different times, and keeping the past state exposed by contexts automatically, arises from the Rete network.

### EAOP, J-Lo & Tracematches

In several other approaches that allow expressing pointcuts over a history of join points, including EAOP [10], Tracematches [1] and J-LO [6], implementation strategies based on state machines are investigated. The state machines are used to evaluate temporal relations between pointcuts, which in the Tracematches and J-LO approaches are expressed in AspectJ. The state machine formalism inherently does not support a memory, thus when variable sharing is allowed between the non-temporal AspectJ pointcuts, this requires an additional form of memory. On the other hand, the logic chaining formalism we have started from in this paper inherently uses such a memory. As for interaction with the base language, current versions of Tracematches extend AspectJ with a `let` pointcut similar to the `escape` discussed in this paper [23], but this mechanism is not covered in [1] and [2], and only examples for accessing the current join point reflection object are discussed in [3]. Furthermore, a `let` pointcut condition is limited to using variables from its enclosing symbol, i.e. only variables defined at the current join point, in contrast with HALO's `escape` predicate which also allows use of variables defined at past join points.

**Rete**

Work by Teodosiu and Pollak [25], and more recent work by Berstel [4], propose extensions of the Rete algorithm for temporal event management. No foundation based on temporal logic is considered, i.e. temporal constraints are expressed only over explicit timestamps, and no higher-order predicates for expressing temporal relations are provided. Furthermore, the temporal constraints always involve a fixed interval of past events, which is motivated by the need to garbage collect memory table entries. In contrast, we have shown how an appropriate most recent join point matching semantics for the temporal predicates still allows for garbage collection.

## 6. Conclusions and Future Work

As stated in the introduction, the contributions of this paper are three-fold. Firstly, we introduced a novel temporal logic-based pointcut language which has features for expressing pointcuts over a history of join points and allowing interaction with the base language. The language is dubbed "HALO".

Secondly, we introduced forward chaining as an implementation mechanism for such a language. Earlier work on similar logic-based languages, such as Alpha and OReA, support either feature but not a combination of both. As we have shown, the backward chaining evaluation strategies used in those approaches are insufficient for supporting the combination. We demonstrated the Rete algorithm as a particular forward chaining algorithm that can support a language that combines both features. We showed how Rete can be extended with support for verifying temporal relations between facts and interacting with the base language.

Thirdly, we demonstrated how the Rete network can be further optimized such that keeping a full history of join point facts is not necessary. Only nodes that are the right input of a temporal node actually need a memory table. Furthermore, because the set of temporal predicates is built-in in the language (rather than an open set), the known semantics of these predicates can be exploited to perform a dynamic analysis of the memory tables. Certain nodes can perform a garbage collection of their previous entries in the memory table when new entries are made.

As future work, we consider extending HALO with predicates that offer a static model of the base application, as in other logic-based pointcut languages [13,21]. Predicates for such a model could be easily added, and would exist in the temporal Rete network as facts that are eternally true. HALO does not currently support recursive rules, which have so far not been proven useful in our examples, and the restriction allows rules to simply be inlined. In previous work, recursive rules have been proven useful for writing pattern-based pointcuts that detect recursive patterns in the static model of the base application [13]. Rete can support recursive rules, but the impact of our additions, especially the `escape` predicate, needs to be further investigated. Furthermore, while the temporal relations expressed by the current set of built-in higher-order temporal operators are similar to the pre-defined time stamp comparison predicates used in Alpha [21], the use of full Prolog in the latter potentially allows additional temporal relations to be expressed. We are currently investigating additional temporal predicates for expressing more interesting temporal relations.

**Acknowledgements**

This work was supported in part by the AOSD-Europe Network of Excellence, European Union grant no. FP6-2003-IST-2-004349. Charlotte Herzeel and Coen De Roover are funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium. Pascal Costanza is funded by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

## REFERENCES

1. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, 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. Pavel Avgustinov, Eric Bodden, Elnar Hajiyev, Laurie Hendren, Ondrej Lhotak, Oege de Moor, Neil Ongkingco, Damien Sereni, Ganesh Sittampalam, Julian Tibble, and Mathieu Verbaere. Aspects for trace monitoring. In *Invited paper at FATES/RV 2006*, 2006.
3. Pavel Avgustinov, Julian Tibble, Eric Bodden, Ondrej Lhotak, Laurie Hendren, Oege de Moor, Neil Ongkingco, and Ganesh Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, ABC Group, 2006.
4. Bruno Berstel. Extending the rete algorithm for event management. In *Proceedings of the Ninth International Symposium on Temporal Representation and Reasoning*, page 49, Washington, DC, USA, 2002. IEEE Computer Society.
5. Daniel Bobrow, Linda DeMichiel, Richard Gabriel, Sonya Keene, Gregor Kiczales, and David Moon. Common lisp object system specification. *Lisp and Symbolic Computation*, 1(3-4):245–394, January 1989.
6. Eric Bodden. J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen university, 2005.
7. Christoph Brzoska. Temporal logic programming with bounded universal modality goals. In *Proceedings of the Workshop on Executable Modal and Temporal Logics*, pages 21–39, London, UK, 1993. Springer Verlag.
8. Jacques Cohen. Describing prolog by its interpretation and compilation. *Commun. ACM*, 28(12):1311–1324, 1985.
9. Maja D'Hondt and Viviane Jonckers. Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development*, pages 132–140, New York, NY, USA, 2004. ACM.
10. Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 170–186, London, UK, 2001. Springer-Verlag.

11. Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, September 1982.
12. Manolis Gergatsoulis. Temporal and modal logic programming languages. In A. Kent and J. G. Williams, editors, *Encyclopedia of Microcomputers*, volume 27, pages 393–408, New York, 2001. Marcel Dekker, Inc.
13. Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the Second International Conference on Aspect-Oriented Software Development*, pages 60–69, New York, NY, USA, 2003. ACM.
14. Wilke Havinga, Istvan Nagy, Lodewijk Bergmans, and Mehmet Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *Proceedings of 5th International Conference on Aspect-Oriented Software Development, AOSD2006*, pages 214 – 225, New York, NY, USA, 2006. ACM.
15. Charlotte Herzeel, Kris Gybels, and Pascal Costanza. Modularizing crosscuts in an e-commerce application in Lisp using HALO. In *Proceedings of the International Lisp Conference 2007*, 2007.
16. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
17. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.
18. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the European conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, jun 1997. Springer-Verlag.
19. Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *European Conference on Object-Oriented Programming, ECOOP 2005*, pages 195 – 213. Springer-Verlag, 2005.
20. Robert Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974. Reprinted in Computers for Artificial Intelligence Applications, (eds. Wah, B. and Li, G.-J.), IEEE Computer Society Press, Los Angeles, 1986, pp. 68–73.
21. Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *European Conference on Object-Oriented Programming*, pages 214–240, 2005.
22. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.
23. Ganesh Sittampalam. Abc version 1.1.1 release announcement. http://abc.comlab.ox.ac.uk/archives/announce/2006-Mar/0000.html.
24. Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. *Lecture Notes in Computer Science, Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, 4089:227–242, 2006.
25. Dan Teodosiu and Gunter Pollak. Discarding unused temporal information in a production system. In *Int. Conf. on Information and Knowledge Management, Baltimore*, 1992.

26. Edi Weitz. Hunchentoot - the common lisp web server formerly known as tbnl. http://weitz.de/hunchentoot/.
27. Tobias Windeln. Logicaj - eine erweiterung von aspectj um logische meta-programmierung. Diploma thesis, CS Dept. III, University of Bonn, Germany, Aug 2003.