Symbiosis in Logic-based Pointcuts over a History of Join Points

Kris Gybels, Charlotte Herzeel* and Theo D'Hondt
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
{kris.gybels|charlotte.herzeel|tjdhondt}@vub.ac.be

ABSTRACT

Within aspect-oriented programming, the quality of aspect code depends on the readability and expressiveness of pointcut languages. Readability is increased by using specialized, declarative pointcut languages. For such languages, their expressiveness is increased if they offer an integration with the base code language. As has previously been shown, offering access to the past history of the base program also increases pointcut expressiveness. A logical desire then is creating pointcut languages that combine both features, but taken to the extreme this is not implementable. We discuss the unimplementable ideal model of declarative history-based logic pointcut languages, and the possible approximations that can be made that are still implementable and what limits they impose on the ideal expressiveness.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

Keywords

Aspect-oriented programming, Multi-paradigm programming, Logic-based pointcuts, History-based pointcuts, Linguistic symbiosis

1. INTRODUCTION

Examples of aspect languages which do not use a specialized pointcut language exist, but the use of specialized languages is clearly motivated. One example of the former is the AspectS extension for Smalltalk [11]: both advice and pointcuts are written in Smalltalk. But even the archetypical example AspectJ [13] uses a specialized pointcut language that is different from the underlying Java. This is motivated by the desire to make pointcuts easier to read by using a declarative pointcut language. A declarative pointcut language specifies *which* join points a pointcut should match, but not *how* to compute this set of join points: compare using an imperative language such as Java to using SQL to query over a database. Many pointcut languages therefore take the form of a declarative query language.

Besides readability, a pointcut language is of course also defined by its expressiveness. This depends on a number of factors, such as how many types of join points are available, how much information is available about them to express queries and the computational power of the language it is based on. In earlier work we specifically proposed the use of pointcut languages based on logic programming such as Prolog, as this doesn't put an a-priori limit on the latter factor by not being Turing-complete (Prolog is), while offering other features such as unification and logic rules which are useful in writing pointcuts [6]. The other factors depend largely on the logic predicates made available to express pointcuts. Several logic-based pointcut languages now exist [17, 8, 19, 3, 1, 2, 15].

Despite the advantages of having a specialized pointcut language, it also introduces a problem of language integration. There is a need to be able to interact, from the pointcut language, with the language used in the rest of the program. For example, an advice which should be executed whenever an instance of a class `Article` is accessed, but only if the article has a high price requires a way for the pointcut language to determine this price. As, following good object-oriented practice, the article object will encapsulate the price, this requires a symbiosis between the pointcut and object-oriented language to send a message to the object from the pointcut. While a number of logic-based pointcut languages actually support such a mechanism [6, 17], the effect of including such a mechanism in a logic pointcut language hasn't yet been studied in detail.

A further study is also required of symbiosis in history-based pointcut languages. Since the inception of event-based AOP [4], several pointcut languages have been developed that allow a pointcut to describe a set of join points based on their relationships to past join points. An approach to including this in a logic-based pointcut language was demonstrated in the work on Alpha [17], but this did not include a consideration of the impact of the history feature on the language integration and only allows to make use of it at the join point that is currently being matched.

In our own work on HALO [9], we have included a symbiosis feature in the language that allows sending base level

*Author funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

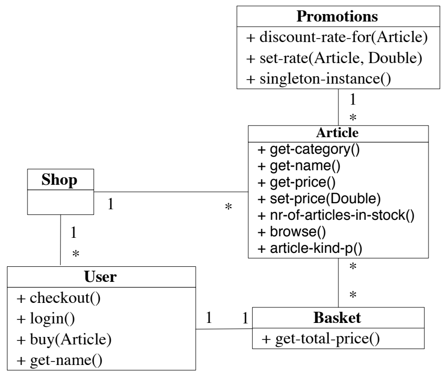


Figure 1: Overview of the e-shop running example.

messages at different times in the pointcut. We have so far mostly studied its applications [9] and implementation using a forward chaining logic [10]. The aim of this paper is to focus on the relationship between HALO and other efforts in integrating symbiosis in logic-based and particularly history-based pointcut languages. We approach this by constructing two unimplementable models of logic-based pointcut languages, one without history and one with, and consider the impact of different limitations and techniques that allow linguistic symbiosis to be used to approximate these models.

In the next section we introduce a simple application which is used for the example pointcuts in the remainder of the paper. The following section first discusses the relationship between linguistic symbiosis and other features of a pointcut language in the context of a non-history based pointcut language, through a simple model of how such logic pointcut languages work. In Section 4 we extend this to history-based logic pointcut languages. Section 5 discusses related work and the final section presents our conclusions and future work.

2. RUNNING EXAMPLE

As a running example, we use an e-commerce application, as shown in Figure 1. The classes `Shop`, `User` and `Article` model the e-shop, its customers and the sold articles respectively. A class `Promotions` simply maps articles to a discount rate, which can be changed using the method `set-rate`, and accessed with `discount-rate-for`. The method `singleton-instance` is used to retrieve the `Promotions` class' only instance.

The shop also needs to have a "discounting feature", which cuts across the methods in Figure 1. The idea is that when a customer visits the homepage of the e-shop, promotional campaigns are advertised by pop-up banners. One such banner could for example state "*Happy Hour!* Login in now, and get a 5% discount!". So if the customer responds to the banner by logging in, he will get a 5% discount on the total amount purchased when he checks out his basket. In order to implement the discounting functionality we need to extend the `login` method in Figure 1 to record whether the user logs in while a promotional campaign is active. Next we need to extend the `checkout` method to compute the appropriate discount. This shows that the "discount" functionality gets scattered over different methods, breaking the modularity of the e-commerce application. We can solve this by implementing the feature as an aspect.

3. LOGIC-BASED POINTCUT LANGUAGES

3.1 SLAL: A Model

From the basic notion of Aspect-Oriented Programming follows a simple mental model for how aspect languages with a logic-based pointcut language work: while executing a base program, the execution is interrupted at every join point. At each such interruption the weaver generates a logic fact to represent this join point and checks every pointcut to see whether the join point fact matches its conditions. When a pointcut matches, its associated advice code is executed.

As a concrete instantiation of this mental model, consider a very simple aspect language with a logic-based pointcut language (SLAL). We use the typical logic language Prolog as the basis for this pointcut language[1]. The base programs it works on are written in Lisp using the Common Lisp Object System.

Figure 2 illustrates the mental model for SLAL: the black boxes represent the different components of the weaver, whereas the labeled boxes denote both the base program and the aspects. The flow between the different weaver components shows that the execution of the base program is mapped onto logic facts and deposited in a fact repository, and that this fact base is queried to verify whether a pointcut is matched by a join point fact; If so, a piece of advice is executed and inserted in the program flow.

To keep SLAL simple it only intercepts a single kind of join point: method calls. Each method call join point is represented as a logic fact using the logic predicate `call` that has an argument `?Receiver`, `?Name` and `?Arguments` referring to the receiver object, name and argument list of a concrete method call join point[2].

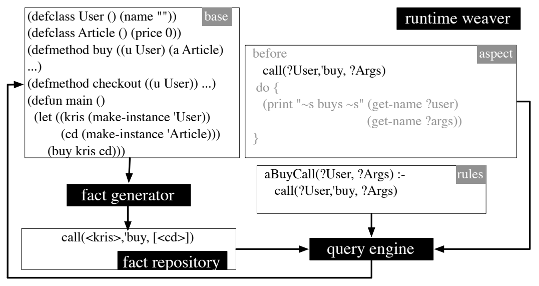
3.2 SLAL: Need for Symbiosis

The model of SLAL is sufficient to explain how logic pointcut languages work, but it has of course also a very limited pointcut language. One way to increase the expressiveness of the pointcut language is to add new types of join points. Most pointcut languages for example expose message sends, variable accesses and changes of variables as additional join points. A more important way of increasing the expressiveness of the language is to expose additional join point context.

The need for symbiosis arises because it is also interesting to consider information derived from the base program's

[1]Note that in our examples, we use a slight deviation from standard Prolog syntax: variables are written with a question mark, as in `?var`. Words beginning with a capital, like `Article`, are atoms (strings) whereas they would be variables in standard Prolog. This change makes clearer when logic variables are used in pieces of advice.

[2]Because Lisp uses multi-dispatch "generic functions" instead of single-dispatch "methods", the concept of a single designated receiver object does not exist there. But we adopt it here because of the familiarity of this concept to programmers in other OO and aspect languages. Hence the "receiver" is always the first argument of the generic function.



```
(defclass User () (name ""))
(defclass Article () (price 0))
(defmethod buy ((u User) (a Article)
...)
(defmethod checkout ((u User)) ...)
(defun main ()
  (let ((kris (make-instance 'User))
        (cd (make-instance 'Article)))
    (buy kris cd)))
```

```
before
  call(?User,'buy, ?Args)
do {
  (print "~s buys ~s" (get-name ?user)
                      (get-name ?args))
}
```

```
aBuyCall(?User, ?Args) :-
  call(?User,'buy, ?Args)
```

```
call(<kris>,'buy, [<cd>])
```

Figure 2: Mental model of how a simple logic aspect language works.

methods as join point context data. In early aspect languages, and still some current ones, it is only possible to do this via the advice language. For example, selecting a method call join point where the bought article is of type "cd" can then be done as follows:

```
before {
  call (?User, buy, [?Article])
}
do {
  (if (article-kind-p ?Article 'cd)
   (print "~s buys a CD" (get-name ?User)))
}
```

The solution of including the type test in the advice however has two problems. The first is conceptual, a pointcut describes when to do some additional behavior and the advice contains the additional behavior. In the above example, the purpose of the aspect is to do logging when someone buys a CD. This is however not readily reflected in the pointcut, which only expresses that the advice is executed whenever someone buys an article, regardless of its kind.

The second problem with having to put the test in the advice rather than the pointcut is a practical one. It impedes using the reusable pointcuts mechanism of the pointcut language. As was shown in CARMA [6] and other logic pointcut languages [17], reusable pointcuts are written using the logic rules mechanism. This should allow us to write a rule expressing when an expensive article is bought, so that this rule can be used in multiple pointcuts. This definitely requires a mechanism to express the test in the pointcut language rather than the advice language.

We can model this mechanism in SLAL by including `result` facts in the fact base. The goal of these `result` facts is to make available all of the join point context that is hidden away in the computations performed by the base methods. The `result` predicate has four arguments: a receiver object, the name of a message, a message argument list and a result object. Conceptually, the fact generator generates facts for this predicate by executing every method in the base program with every possible argument list and recording a fact for each returned result (we ask the reader to keep in mind we're explaining a conceptual model of logic pointcut languages here, and to suspend disbelief of this last statement for a moment).

We can now use this to write the pointcut to select method call join points where the bought article is of type "cd" as depicted below. The `result` condition matches when the "article-kind-p" test evaluates to "true" for a bought article.

```
before{
  call(?User, buy, [?Article]),
  result(?Article, article-kind-p, ['cd], true)
}
do{
  (print "~s buys a CD" (get-name ?User))
}
```

3.3 Symbiosis Implementation

In the previous section, we've explained the conceptual model of SLAL in which `result` facts are generated by the fact generator by executing every method with every possible argument list. It's clear that to implement this would in fact require the fact generator to rely on an oracle (similar to a Turing oracle [18]) to do this, not to mention an infinite amount of space to store the infinite number of facts. But an approximation of this conceptual model can be made, in which the facts are produced on demand, with certain limitations.

If a Prolog engine is suitably integrated with the base language, the `result` predicate can be defined as a primitive predicate. This has been demonstrated in one form or another in for example SOUL [20] (on which the logic pointcut languages CARMA [6] and OReA [3] are based), Alpha [17] and other works, a more extensive survey is provided by D'Hondt [3]. In the primitive implementation of the `result(?rcvr, ?m, ?args, ?result)` predicate, the message `?m` is simply sent to the object `?rcvr` with the argument list `?args`. Depending on whether a value is already given for the `result` condition's `?result` argument, the returned value is compared with the given value which makes the logic proof either succeed or fail, or the `?result` variable is simply assigned the return value.

This implementation of `result` conditions requires all arguments except for the `?result` to be already given, thus putting some limitations on SLAL in comparison with the conceptual model. This does not mean that no variables can be used as arguments for such conditions, but they must be given a value by "earlier" parts of the pointcut. Understanding what the "earlier" parts are and how this works exactly requires some understanding of how the Prolog logic evaluation procedure works[3]. But simplified, a logic query is proven to be true or false by starting with the first condition and so on, using facts or rules to prove the condition, and as a side-effect constraining variables to have the value given by a fact. I.e. if a condition `call(?Rcvr, ?Name, [35])` is proven by using a fact `call(<cd1>, 'set-price, [35])`, the variables `?Rcvr` and `?Name` get the value `<cd1>` and `'set-price` respectively.

As an example of the limitations this puts on the language, take the following examples:

```
result(?Article, get-price, [], ?Price),
call(?User, buy, [?Article])

call(?User, buy, [?Article]),
result(?Article, get-price, [], ?Price)

call(?User, buy, [?Article]),
result(?Article, ?Name, ?Args, 35)
```

The first pointcut doesn't work, because when proving the first condition, `?Article` will not yet have a value. The

[3]The Prolog proof procedure is known as SLD resolution.

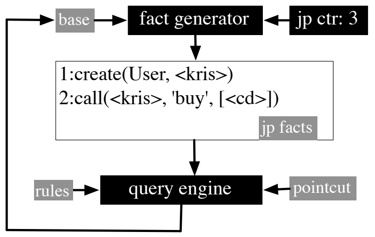


Figure 3: The mental model for X-HALO.

second pointcut is declaratively entirely the same as the first one, but it will work because the conditions have been swapped so that `?Article` will have a value before the `result` condition is evaluated. The last pointcut won't work, consider what this pointcut expresses: "the pointcut should match a call of `buy` with a certain article `?Article` if for some random method and random arguments list, invoking the method on the article gives the result 35".

4. HISTORY-BASED LOGIC POINTCUT LANGUAGES

In the previous section, we considered some typical symbiosis implementation issues for the integration of logic and object-oriented programming that also arise in aspect-oriented pointcut languages. In this section we discuss the issues that arise when this is extended to pointcut languages that use a history of join points. To this end, we extend the SLAL model to X-HALO[4], a straightforward extension of the model in which the fact base is never cleared. From the conceptual model follows equally straightforwardly that the `result` facts are also never cleared, but this brings up new symbiosis implementation issues.

4.1 X-HALO: A Model

The mental model of how X-HALO works is almost the same as SLAL's, the only difference is that information about past join points is kept. In SLAL, the fact base is cleared at every join point, thus only facts about the current join point are available. To model history-based aspect languages, in X-HALO the fact base is not cleared. To be able to differentiate between the facts generated for different join points, they are tagged with a timestamp, which is simply given by a counter increased at every join point [5]. This is illustrated in Figure 3. The facts in the fact base are all notated in the notation *`number:fact`*[6]. We also include instance creations as join points in this model, for which facts of the form `create(`*`class, instance`*`)` are generated. So for example, if we execute a base program:

```
(defun main ()
  (let ((kris (make-instance 'User))
        (cd (make-instance 'Article)))
     (buy kris cd)))
```

The following logic facts will be present in the fact base after its execution:

```
1:create(User, <kris>).
2:create(Article, <cd>).
3:call(<kris>, buy, [<cd>]).
```

The timestamp notation is similarly used to pick out join point facts in the conditions of a join point. Either a concrete value can be used, or a logic variable. For example, a pointcut `1:call(?User, checkout, ?Args)` will be matched against the first join point ever recorded. The pointcut `?T:call(?User, checkout, ?Args)` will match any method call join points where the method name is 'checkout'.

Similarly, each user-defined predicate gets a time stamp. As an example, consider the rule below. One should read the rule as follows: " a join point matches the condition `buy-article ([?Article])` at a time `?T` when the join point matches `call (?User, buy, [?Article])` at that time `?T`".

```
?T:buy-article (?User, [?Article]) :-
  ?T: call (?User, buy , [?Article]).
```

A more interesting example where multiple join points are referred:

```
?T: logBoughtArticles (?Article) :-
  ?T: checkout (?User),
  ?T2: buy-article (?User, [?Article]),
  ?T2 < ?T.

?T: checkout (?User) :-
  ?T: call (?User, checkout).
```

The pointcut matches a join point time-stamped `?T` if this join point matches `checkout (?User)` and a join point time-stamped `?T2` matches `buy-article (?User, [?Article])`, so that the latter join point was recorded before the other (since `?T2 < ?T` must hold).

A slight change is made to how pieces of advice are written. Recall that at the point where the weaver processes a pointcut-advice, the query engine will be used to resolve the pointcut and the resulting bindings for free variables in the pointcut are used to ground and then execute the piece of advice code. Hence the semantics of X-HALO is such that each pointcut-advice is processed in relation to the current join point. The advice notation is extended so that one timestamp variable is explicitly designated to be about the current join point. This variable will be given the value of the current join point's timestamp. For example, if we have a base program and pointcut-advice as in the following code listing (`logBoughtArticles` defined as in previous example):

```
(defun main ()
  (buy kris cd)
  (checkout kris)
  (buy kris dvd))

before ?T {
  ?T:logBoughtArticles(?User, ?Article)
}
do{
  (print "~s bought a ~s" (get-name ?User)
                          (get-name ?Article))
}
```

[4]HALO stands for "History-based Aspects using LOgic". The name is taken from the pointcut language we discussed in earlier work [9], but the prefix "X-" for "eXtreme" refers to the unimplementability of the full conceptual model.

[5]We assume sequential program execution. A model of X-HALO for parallel programming is left for future work.

[6]This can be done in standard Prolog, the expression `foo:bar` is syntactic sugar for a fact `:(foo,bar)` for the predicate ":"

At join point 2, ?T will be given the value 2 (when the second statement in the main is executed) and the advice code will be executed because the pointcut matches, resulting in logging a message "Kris bought a cd". At join point 3, ?T is given the value 3 and the pointcut does not match. The extension of explicitly designating one timestamp variable to be about the current join point is necessary because of the following: if ?T had not been given a value, the condition ?T:checkout(?User) in the rule logBoughtArticles would be matched against the checkout fact with timestamp 2, meaning the pointcut as a whole would match at join point 3, which is clearly not the desired semantics.

4.2 X-HALO: Need for Temporal Symbiosis

As in SLAL, it would be beneficial to include result facts in the fact base. We can start again with a conceptual model in which these facts are simply magically generated and recorded at every join point. Thus, like all other join point facts they get a timestamp. This makes it possible to write pointcut queries that refer to the result of sending a message to an object in the *past*.

To elaborate on this, consider an implementation of the "discounting" feature of the e-shop application. The discounting feature consists of two aspects: popping up banners to advertise discounts for articles, and giving the actual discount when a user makes a purchase. One strategy the e-shop owner could follow, would be to promise customers discounts if they login now, e.g. by displaying a banner "Discount guaranteed if you *login now!*". Hence if the customer responds to the promotion by logging in, he's guaranteed to get a discount when he purchases an article, no matter whether the promotion is switched off in between the login and the purchase. In X-HALO, this can be implemented by means of a pointcut that captures the "login" and the "buy" method calls and exposes the discount rate at the time of the "login". This is for example done in the pointcut definition below, where ?T1 is used to quantify both the join point condition matching the "login" and the result condition for exposing the discount rate.

```
?T2:logDiscountRate(?User, ?Article, ?Rate) :-
  ?T1:call(?User, login, []),
  ?T1:result(Promotions, singelton-instance, [], ?Singleton),
  ?T1:result(?Singleton, discount-rate-for, [?Article], ?Rate),
  ?T2:call(?User, buy, [?Article]),
  ?T2 < ?T1.

before ?T2 {
  logDiscountRate(?User, ?Article, ?Rate)
}
do{
  (print "~s gets a ~ % discount on ~s" ?User ?Rate ?Article)
}
```

For the program below, where the customer logs in at a time there is a 5 % discount rate guaranteed for CD's, this means that the customer gets a 5 % discount on *all* the CD's he subsequently buys, even though in the meantime, the discount rate for CD's is switched to 0 %. Otherwise, the customer might have felt cheated for not getting the discount he was promised at login.

```
(defun main ()
  (let ((kris (make-instance 'User))
         (cd (make-instance 'Article :amount 10)))
       (set-rate (singleton-instance 'Promotions) 'cd 0.05)
       (login kris)
       (buy kris cd)
       /* change the discount rate for cd's*/
       (set-rate (singleton-instance 'Promotions) 'cd 0.00)
       (buy kris cd)))
}
```

However, the e-shop owner might follow another strategy for giving discounts. He could decide to only guarantee a discount if the customer buys the article *right now*, e.g. by advertising "Guaranteed discount if you *buy now!*". Again, due to the fact that result conditions are timestamped, this is easy to express in X-HALO. In order to implement this discount strategy, one can reuse the pointcut definition above, but one would have to change the timestamp of the result condition, exposing the discount rate, to ?T2, which coincides with the timestamp of the "buy" condition.

4.3 Implementing Temporal Symbiosis

As with SLAL, an implementation for the result predicate in X-HALO should be provided that approximates the conceptual model. This again implies that certain limitations are put on the language in comparison to what is possible given the conceptual model. Several variations of X-HALO are actually possible, and we discuss the way these can be implemented and their limitations in this section.

4.3.1 *Current* results *only*

The mechanism of implementing result in SLAL, described in Section 3.3, can also be used in X-HALO. But this can only work for a severely limited version of X-HALO: the limitation with respect to the conceptual model is that the result facts are cleared from the fact base after every join point, so that only the facts for the current join point are available. This means that the example of Section 4.2 is not a valid pointcut in this variation of X-HALO.

The limitation is necessary because the implementation of Section 3.3 always gives result facts as if they are produced at the current join point. The implementation executes the method on-demand when the Prolog engine needs to prove the existence of a result fact. Thus having the effect that this will always give the result at the time of the current join point. Suppose this implementation was adopted for timestamped result facts in X-HALO, then consider the effect on the example from Section 4.2. When the pointcut matches the first 'buy' call (line 6 in the base program), the advice will print a message: "Kris gets a 5% discount on cd". Using the semantics of result from the conceptual model, we expect that when the second buy call (line 9) happens, the same message is printed. But because the rate of CDs was changed (line 8), the SLAL implementation of result would result in a different message being printed: "Kris gets a 0% discount on cd".

4.3.2 *Stratifying the use of* result

One way of implementing temporal symbiosis involves splitting up the pointcut definitions. We split up the pointcut definitions so that we get pointcuts where result conditions are only quantified with the "current time stamp". Hence all of the latter "split up" conditions can be resolved using the on-demand strategy, discussed in Section 3.3. If we cache these results, we can implement temporal symbiosis in terms of matching the result conditions against the cached results. We next elaborate on how pointcuts are split up and we discuss the limitations of this approach when rule abstraction, "future variables" and recursive rules are combined with result conditions.

Splitting up pointcuts. The timestamp of a `result` condition must coincide with a condition matching a join point. E.g. for the pointcut below, the `result` conditions have the time stamp `?T2`, which matches the "login" join point condition's time stamp. The latter expresses that the `result` condition needs to hold at the same time a join point matching the "login" join point condition occurs. Hence in the example, the discount rate of the article must be the rate active when the user logged in. When we split up a pointcut, we must make sure that the `result` conditions end up in the same pointcut as the join point condition with the same time stamp.

```
?T2:logDiscountRate(?User, ?Article, ?Rate) :-
  ?T1:create(Article, ?Article),
  ?T2:call(?User, login, []),
  ?T2:result(Promotions, singelton, [], ?Singleton),
  ?T2:result(?Singleton, discount-rate-for, [?Article], ?Rate),
  ?T1 < ?T2,
  ?T3:call(?User, buy, [?Article]),
  ?T2 < ?T3.
```

More specifically, whenever we have a pointcut in which a `result` condition occurs, we split it up, so that we have one part consisting of the `result` condition and all conditions where the time stamp is *smaller than or equal to* the `result` condition's time stamp, and another part which is simply the rest of the pointcut. Determining whether a condition's time stamp is smaller or equal than the `result` condition's can be done by analyzing the time stamp constraints in the pointcut definition.

Following this strategy, the pointcut above is split up in the two pointcuts listed below. The first pointcut groups the `result` conditions, the "login" condition, because it has the same time stamp (namely `?T2`), and the "create" condition because its time stamp is smaller than the `result` conditions' (note `?T1 < ?T2`). The other pointcut, named `logDiscountRate`, consists of the rest of the pointcut above, namely the "buy" condition and one extra condition. This condition refers to the result to be cached when the first pointcut is resolved.

```
?T2:callLogin(?User, ?Article):-
  ?T2:call(?User, login, []),
  ?T1:create(Article, ?Article),
  ?T1 < ?T2,
  ?T2:result(Promotions, singleton-instance, [], ?Singleton),
  ?T2:result(?Singleton, discount-rate-for, [?Article], ?Rate).

?T2:logDiscountRate(?User, ?Article, ?Rate) :-
  ?T1:rate-at-login(?User, ?Article, ?Rate),
  ?T2:call(?User, buy, [?Article]),
  ?T1 < ?T2.
```

The above splitting is done automatically. This is similar to writing this manually using a version of X-HALO with current `result` only. Below are two advices which do this. The first one is triggered when a user logs into the shop and asserts a fact for each article, mapping it to the current discount rate. The second piece of advice is triggered when a user buys an article and prints the discount rate, which was computed at the time the customer logged in. Of course, the advantage of doing this automatically is that the programmer does not need to write such boilerplate code himself.

```
before ?T {
  ?T:callLogin(?User, ?Article, ?Rate)
}
do{
  (assert *fact-repository*
    (make-instance 'Fact :time :T :predicate 'rate-at-login
                         :arguments '`(,?User ,?Article ,?Rate)))
}

before ?T {
  ?T:logDiscountRate(?User, ?Article, ?Rate)
}
do{
  (print "~s gets a ~ % discount on ~s" (get-name ?User)
                                         ?Rate
                                         (get-name ?Article))
}
```

The implementation strategy we just described can only work if we put some restrictions on the use of `result`. A first restriction is that all variables used in a `result` condition must be bound by conditions with a time stamp that is "smaller" than the `result` condition's own time stamp. Another restriction has to do with how to deal with (recursive) rule definitions.

Future Variables. The `result` conditions must not reference variables that are only bound by a "future" join point condition. By this we mean that all variables used in a `result` condition should be bound by conditions matching join points that happen before or at the same time the `result` condition needs to hold. This is because the on-demand implementation of linguistic symbiosis requires that *all* variables are bound at the time a `result` condition is evaluated (see Section 3.3). In Section 4.3.3, we discuss alternative implementation strategies for linguistic symbiosis predicates to allow "future variables" in `result` conditions.

Flattening rules & Recursion. Before we split up the pointcut definitions, we need to *flatten* them. *Flattening* a pointcut means that all conditions based on a user-defined predicate are replaced by the defining rule's right-hand side. If there are multiple definitions for a particular predicate, then flattened versions of the pointcuts will be created for each of these definitions. Below we have depicted a rule, a pointcut and its flattened version. Note that without the flattening, the pointcut cannot be split up correctly because there is no guarantee a `result` condition will be associated with a join point condition. E.g. the rule below is not defined in terms of a join point condition, and without the flattening, the condition would never be evaluated, as described in the previous paragraph. However an apparent drawback of the flattening approach is that recursive definitions containing a `result` are no longer allowed.

```
is-cd(?Article) :- /* rule */
  result(?Article, is-article-kind-p, [cd], true).

logCdPurchase(?User):- /* pointcut */
  call(?User, buy, [?Article]),
  is-cd(?Article).

logCdPurchase(?User):- /* flattened pointcut */
  call(?User, buy, [?Article]),
  result(?Article, is-article-kind-p, [cd], true).
}
```

4.3.3 *Stratified* `result` *with Future Variables*

"Future variables" can be supported in X-HALO if we record the program state at specific points in the execution of a program so that we can evaluate the `result` condition in terms of this recorded state at a later time, e.g. when we *do* have a binding for the "future variables". One way of implementing this, is again by splitting up the pointcuts so

```
?T2:logDiscountRate(?User, ?Article, ?Rate) :-
  ?T2:call(?User, login, []),
  ?T2:result(Promotions, singleton-instance, [], ?Singleton),
  ?T2:result(?Singleton, discount-rate-for, [?Article], ?Rate),
  ?T1 < ?T2,
  ?T3:call(?User, buy, [?Article]),
  ?T2 < ?T3.
```

Figure 4: "Future variable" `?Article` is only bound *after* the `result` conditions must hold.

that we end up with pointcuts where `result` conditions are only quantified with the "current" time stamp. `result` conditions containing "future" variables will however be moved to pointcuts that provide bindings for all the "future" variables. Note that this means the `result` condition's time stamp will be changed to a *later* time. These will therefore be evaluated in terms of past program state, copied at the "right" time (the original time the `result` condition was quantified with). We next discuss how we need to extend some of the split-up pointcuts to generate copies of the join point context they expose.

Saving the past state. A `result` condition contains a "future variable" when it uses a variable that is only bound by a condition to be matched at a *later* time than the `result` condition needs to hold. In Figure 4, the variable `?Article` referenced in the second `result` condition is a "future" variable, because it is only bound by the "buy" condition, whose time stamp `?T3` is "larger" than the result condition's time stamp `?T2` (note `?T2 < ?T3`). This means we cannot simply split up the pointcut following the strategy described in Section 4.3. That would result in one pointcut consisting of the first four conditions, and another with the rest of the conditions: there would be no binding for `?Article` in the first pointcut. Rather, we have to split up the pointcut so that the `result` condition with the future variable ends up in a pointcut where the other conditions provide a binding for the future variable. In other words, we defer the `result` condition to a later time. Below we have split up the pointcut from Figure 4 accordingly.

```
?T:callLogin(?Singleton):-
  ?T:call(?User, login, []),
  ?T:result(Promotions, singleton-instance, [], ?Singleton).

?T1:logDiscountRate(?User, ?Article, ?Rate) :-
  ?T1:call(?User, buy, [?Article]),
  ?T2:saved-input([?PromoSingleton]),
  ?T1:result(?PromoSingleton, discount-rate-for, [?Article], ?Rate)
  ?T2 < ?T1.
```

The first pointcut in the listing above is matched when a "login" join point occurs and exposes the `Promotions`' object. The second pointcut contains the condition for matching the "buy" join point, a `saved-input` condition and a `result` condition. In comparison to the original pointcut in Figure 4, the `result` condition computing the discount rate of an article, is deferred from the "login" time to the "buy" time. Hence the latter `result` condition will be resolved (on-demand) when "buy" join points occurs. According to the pointcut in Figure 4 however, the discount rate should coincide with the rate active when the "login" happens. Guaranteeing that the discount rate is indeed the rate active at "login", is done by means of the condition `saved-input`. The `saved-input` condition binds all variables – other than the "future" variable – needed to resolve the `result` condition. These bindings refer to copies of join point context exposed at the "login". Thus when the `result` condition is evaluated to compute the discount rate, it is done so in respect to the program state at the "login" time, yielding the desired semantics. `saved-input` facts are generated whenever a "login" join point occurs. Again, this saving of the fact, as well as splitting the pointcut, could be done by the programmer manually, by simulating this as shown in the piece of advice below. But by doing it automatically in an implementation of `result`, (s)he does not have to write this sort of plumbing code.

```
before ?T {
  ?T:callLogin(?Singleton)
}
do{
  (assert *fact-repository*
    (make-instance 'Fact :time :?T
                         :predicate 'saved-input
                         :arguments `(,(copy ?Singleton))))
}
```

5. RELATED WORK

We are certainly not the first to consider the general problem of integrating a logic language with an object-oriented one. A survey of languages and systems that offer a level of integration of a logic language (and other forms of rule-based languages) with an object-oriented language is offered by D'Hondt [3]. While numerous such integrations exist, the effects of such integrations on the two languages are not always considered, which can also result in more ad-hoc forms of integration. The goal of the research on linguistic symbiosis is to explicitly take such integrations as a field of study in its own right.

The term linguistic symbiosis was first coined in work on integrating RBCL with C++ as a way of implementing reflection [12]. Researchers that are, or were once, at our lab have taken up the term, with a primary focus on integration of logic and object-oriented languages as exemplified in the logic language SOUL [20, 3, 7]. These can be divided into studies on the core symbiosis mechanisms [7], its applications in logic meta programming [20, 7], and applications in implementing business rules [3]. Outside of this RBCL-influenced branch of research exist industrial approaches such as CORBA and the .NET inter-operability platform, as well as other research studying language integration between for example Java and Scheme [5], or formal foundations for integration [16].

In our own application of SOUL to AspectJ-like aspect-oriented programming, CARMA [6], the symbiosis mechanism of SOUL was simply adopted without further study. It is sometimes considered that such a mechanism is not necessary, because in aspect languages, the advice body which is written in the base language provides a straightforward way of interacting with the base language. In this paper we have made explicit our rejection of this notion on the basis of making a clear conceptual split between the "when" and "what" parts of a piece of advice, and the use of the mechanism in the logic rules of the logic pointcut language.

We can use our conceptual models and the derived variations to place existing logic-based pointcut languages. Most are not history-based, so they fall in the category of SLAL or variations that further limit the integration with the base language (all are more expressive than SLAL when it comes to offering additional types of join points etc.). Alpha [17] is history-based and offers a kind of symbiosis mechanism in the form of its "@" construct, but little is said about it [17]. No history is taken over this mechanism, thus placing Alpha in our X-HALO variation discussed in Section 4.3.1 ("Current results only").

GAMMA [14] is an initial proposal to support pointcuts that refer to the future by performing a fix point computation. Rather than by saving program state to "transport" it to the future in pointcuts that have been split, as we've considered here, the fix point computation proposal involves executing the program multiple times. While facts that give access to the instance variables of all objects are considered, the impact of including a `result`-like predicate to access methods was not considered.

As mentioned earlier, our own work on HALO [10, 9] falls in the variation discussed in Section 4.3.3 but with a further limitation to a fixed set of higher-order timestamp comparison operators which allows dynamic optimization of the fact base.

6. CONCLUSIONS

In this paper, we've discussed the differences between a number of pointcut languages with regards to integration with the base language. We approached this by positing two simple models for respectively non-history based and history-based logic pointcut languages. In these models, integration with the base language is modelled straightforwardly by `result` facts that are "simply" generated by the fact generator. While this is as such unimplementable, it allows a discussion of how different actual implementations approximate the more ideal language given by the model, and the influence this approximation has on what pointcuts can actually be expressed. In the section on related work, we briefly discussed how existing logic pointcut approaches fit in the different variations of the models.

7. REFERENCES

[1] B. Adams and T. Tourwé. Aspect orientation for C: Express yourself. In *AOSD 2004 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, Chicago, US, 2005.

[2] J. Brichau, A. Kellens, K. Gybels, K. Mens, R. Hirschfeld, and T. D'Hondt. Application-specific models and pointcuts using a logic meta language. In W. D. Meuter, editor, *14th International Smalltalk Conference*, Lecture Notes in Computer Science (LNCS). Springer Verlag, 2006.

[3] M. D'Hondt. *Hybrid Aspects for Integrating Rule-Based Knowledge and Object-Oriented Functionality.* PhD thesis, Vrije Universiteit Brussel, 2004.

[4] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. Technical Report 01/3/INFO, Ecole des Mines de Nantes, 2001.

[5] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 231–245, New York, NY, USA, 2005. ACM Press.

[6] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the Second International Conference on Aspect-Oriented Software Development*, 2003.

[7] K. Gybels, R. Wuyts, S. Ducasse, and M. D'Hondt. Inter-language reflection: A conceptual model and its implementation. *Elsevier Journal on Computer Languages, Systems & Structures*, 32:109 – 124, 2006.

[8] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *Proceedings of 5th International Conference on Aspect-Oriented Software Development, AOSD2006*, 2006.

[9] C. Herzeel, K. Gybels, and P. Costanza. Modularizing crosscuts in an e-commerce application in Lisp using HALO. In *Proceedings of the International Lisp Conference 2007*, 2007.

[10] C. Herzeel, K. Gybels, P. Costanza, C. D. Roover, and T. D'Hondt. Forward chaining as an implementation strategy for the history-based logic pointcut language HALO. In *International Conference on Dynamic Languages*, 2007.

[11] R. Hirschfeld. Aspect-Oriented Programming with AspectS. In *Revised Papers from the 2002 International Conference on Objects, Components, Architectures, Services and Applications for a Networked World*, 2003.

[12] Y. Ichisugi, S. Matsuoka, and A. Yonezawa. RbCl, a reflective object-oriented concurrent language without a runtime kernel. In *IMSA'92 International Workshop on Reflection and Meta-Level Architectures*, 1992.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001.

[14] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *Workshop on Foundations of Aspect-Oriented Languages*, 2005.

[15] R. Lämmel and K. D. Schutter. What does aspect-oriented programming mean to COBOL? In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 99–110, New York, NY, USA, 2005. ACM Press.

[16] G. A. Ospina and B. L. Charlier. Towards precise descriptions for programming language interoperability: a general approach based on operational semantics. In *Proceedings of the Third International Conference Interoperability for Enterprise Software and Applications*, 2007.

[17] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *European Conference on Object-Oriented Programming*, 2005.

[18] A. Turing. Systems of logic based on ordinals. In *Proceedings of the London Mathematical Society*, 1939.

[19] T. Windeln. LogicAJ - eine erweiterung von AspectJ um logische meta-programmierung. Diploma thesis, CS Dept. III, University of Bonn, Germany, Aug 2003.

[20] R. Wuyts. *A Logic Meta Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, 2001.