Modularizing Crosscuts in an E-commerce Application in Lisp using HALO

Charlotte Herzeel[*], Kris Gybels, Pascal Costanza [†] and Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
{ caherzee | kris.gybels | pascal.costanza | tjdhondt }@vub.ac.be

ABSTRACT

Some program concerns cannot be cleanly modularized, and their implementation leads to code that is both hard to understand and maintain. In this paper we consider extending an e-commerce application, written in CLOS, with two of such crosscutting concerns. Though most of the time Common Lisp's macro facilities and CLOS' method combinations can be used to modularize crosscuts, we discuss the use of a more declarative solution when crosscuts depend on the execution history. For this purpose we give an overview of HALO, a novel pointcut language based on logic meta programming and temporal logic, which allows one to reason about program execution and (past) program state.

1. INTRODUCTION

In this paper we take a look at the implementation of an e-commerce application written in CLOS on top of the Hunchentoot framework, and we extend it with a *discounting* and a *suggestions* feature. Implementing the extensions is done by introducing new classes and we observe that in order to integrate these classes with the base application, multiple methods need to be adapted. The concern of this paper is whether these adaptations can be done in a modularized manner in CLOS, or whether we need to resort to a crosscutting implementation. The latter is important to know when striving for a separation of concerns.

Full separation of concerns through modularization using any object-oriented language is difficult to achieve because a program can only be modularized in one way at a time, possibly matching very well for particular concerns, but other concerns that do not align with this modularization end up scattered over different modules. Concerns that do not align with a particular modularization are called crosscutting concerns.

Crosscutting concerns occur in CLOS programs as well. The two new features we want to add to the e-commerce application are in fact examples of crosscutting concerns. Most of the time crosscutting concerns can be modularized in Common Lisp by means of `before`, `after` and `around` methods. As an example consider a banking application where sensitive operations such as withdrawing and depositing money need to be logged. This can be achieved by adding `before` methods for each sensitive method, rather than scattering the logging code over the different method bodies. However, implementing the *discounting* and *suggestions* features from the e-commerce application in a similar way is not trivial. The reason for this is that they seem to rely on a dynamic interplay of different methods and temporary program state. Our proposed solution applies aspect-oriented programming to the problem.

The goal of aspect-oriented programming [15] (AOP) is to modularize such crosscutting concerns. An aspect language is a language extension for a base language, such as Common Lisp, that introduces new constructs that allow the implementation of crosscutting concerns in distinct modules, called *aspects*. An aspect language consists of *a means of describing join points*, namely a pointcut language, and *a means of affecting behavior at join points*, namely an advice language, where join points are well-defined points in (the execution of) a program. A pointcut can be seen as a predicate over all join points in a program, to pick out join points of interest. In CLOS, a `before`, `after` or `around` defines a piece of advice for a single join point, while a macro could be used to expand a declarative description of several join points – that is, a pointcut – into several such `before`, `after` and `around` methods.

Although early research in AOP focused on aspects that are triggered at single join points at distinct moments in time, recent research has evolved towards aspect that are triggered based on the occurrence of a series of consecutive join points in the execution of a program and their respective program state in the past: They were dubbed event-based aspects, stateful aspects [6] and context-aware aspects [22]. In this paper we refer to them as history-based aspects, and as we will explain, history-based aspects are a perfect means for modularizing the crosscuts in the e-commerce application.

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

[†]Funded by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

In the pursuit of an expressive pointcut language for CLOS for history-based aspects, we have applied logic meta programming to the problem, resulting in HALO. HALO – which stands for "History -based aspects using Logic" – extends our earlier research on logic pointcut languages [9] with support for writing pointcuts that can depend not just on the current join point but also on previous join points and also on past program state.

The rest of this paper is organized as follows. In Section 2 we present an overview of HALO. We present both the advice language and the pointcut language, as well as some basic examples. The follow-up section discusses the implementation of a basic e-commerce application, and how it can be extended with a *discounting* and a *suggestions* feature. A modularized implementation in HALO of the latter two features is subsequently explained. Section 5 presents an overview of related work, including other approaches to aspect-oriented programming in Lisp. Finally, before presenting our conclusions, we elaborate on some future work.

2. THE HALO LANGUAGE

HALO ("History-based Aspects using LOgic") is a novel logic-based pointcut language for Common Lisp [2] that allows pointcuts to be expressed over a history of join points as well as allowing interactions with the base language (so called hybrid pointcuts). In this section, we first give some background information on the concept of advices in aspect-oriented programming and how this works in HALO, as well as the motivation for making HALO a temporal-logic-based pointcut language. The remainder of the section explains the HALO pointcut language itself.

2.1 Aspect-Oriented Programming & Advice

Most aspect languages are founded on the concept of advice. Advice already exist in Common Lisp in the form of *before*, *after* and *around* methods in CLOS. The latter "intercept" when a generic function with a particular set of specializers is executed and execute additional behavior before, after or around its execution. This concept is extended in AOP to pointcuts: rather than being defined on a single generic function with a particular name, an advice is defined using a description of which "join points" the advice should intercept, which can be more complex. A join point is a key event in the execution of the program, such as the execution of a method.

While some aspect languages have explicit "before" and "after" advices, HALO does not distinguish between types of advices. Instead, the join point model includes distinct "return" join points on which the application of an advice has the same effect as an "after" advice in other languages.

An example to illustrate the form in which advices using the HALO pointcut language are defined:

```
(at ((gf-call 'buy ?args)
     (escape ?time (get-universal-time)))
   (print "Buy was invoked with arguments: " ?args)
   (print " at time " ?time))
```

This is an example of a straightforward logging advice, which intercepts invocations of the generic function `buy` and prints information about the invocation to a log. The advice body consists of two calls to `print`. The `gf-call` and `escape` forms are predicates in the HALO logic pointcut language.

For comparison, a *before* method in CLOS that does the same as the simple advice depicted above:

```
(defmethod buy :before (args)
  (let ((time (get-universal-time)))
    (print "Buy was invoked with arguments:" args)
    (print " at time " time)))
```

2.2 Logic Pointcuts over Joinpoint Histories

Motivation for logic-based pointcut languages. Several ways to write pointcuts exist. For example, the AspectS framework for Smalltalk [12] uses Smalltalk's meta-object protocol to write pointcuts as Smalltalk expressions that compute a collection of methods. The use of Smalltalk expressions means a Turing-complete language can be used to express pointcuts, the use of the MOP furthermore allows complex pointcuts that check statements in the method bodies, relationships between classes and so forth. A similar approach could be taken in CLOS. But on the other hand, previous work on CARMA [9] and other logic-based pointcut languages [21, 10, 25] has demonstrated the suitability of logic programming [17] as the basis for a pointcut language. Logic programming allows pointcuts to be written in a declarative style, without control loops and such, which in general is taken to make expressions in these languages easier to read (as Kowalski explained through his well-known equation "algorithm = logic + control" [18]).

Logic pointcut matching. In logic pointcut languages, pointcuts are written as logic queries over logic facts that give information about the join points. Conceptually, whenever a join point (such as a method execution) occurs, facts giving information about the join point (such as its name and its arguments) are made. The pointcuts of all advices are then checked against this fact base. If the pointcut (a logic query) has a solution using this fact base, we say the pointcut *matches* the join point. Such a match means the advice body is then executed. Note that it is possible for logic queries to have multiple solutions, with different values for logic variables. In that case, in HALO, the advice is executed for every solution. The advice body can also make use of the values of the logic variables, as illustrated in the example in the previous section: the `?args` and `?time` variables are used in the advice body, and will respectively contain the arguments and time of execution of calls to the function `buy`.

HALO and temporal logic. In most early pointcut languages, no history of join points was available to pointcuts. Using a logic-based pointcut language as example: in CARMA, whenever a join point occurs, only information about that join point and some additional information about the program (the classes, relationships between classes, their methods and so forth) is available in the fact base. More recent proposals for pointcut languages have included some form of allowing pointcuts to depend on join points that occurred earlier. For logic-based pointcut languages, this means the fact base does not just include information about the "current" join point, but also about earlier join points. This allows for pointcuts that intercept the execution of a method named "buy", but only if there was previously an execution of "login". This means there are also temporal relations between the join points in the fact base. To deal with these, HALO is in particular based on temporal logic programming.

2.3 HALO Pointcut Language

In a logic pointcut language, join points are represented as logic facts and 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 1.

Note that while HALO is a logic-based language, in contrast with other logic-based pointcut languages, it does not use the syntax of the well-known logic language Prolog. Instead, 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)`.

2.3.1 Joinpoint 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 accessing or changing of a slot (instance variable).

Figure 1 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` predicate respectively capture invocations of a generic function and executions of specific methods of a generic function. 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 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 actual instance.

2.3.2 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, 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)`.

The `since` temporal predicate is the more difficult one of the three 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.

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 [19]).

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 and gives 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 `?arti-`

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

Figure 1: Grammar for HALO pointcut language. For conciseness we have depicted the arguments of the different predicates as logic variables preceded by a "?".

cle variables, the inner pointcut then matches on all previous calls to buy with the same article object as argument.

2.3.3 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. But if the return value of the Lisp code is nil, the condition has no solution. 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; The only restriction is that applies is that all of the variables used in its condition are bound by the rest of the pointcut. The semantics is such that it appears as if the Lisp code of the escape conditions inside the most-recent is evaluated when user2 buys an article. The following example advice prints the price for which a user previously bought an article when the same article is bought by another user:

```
(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.

2.4 Defining Rules

Programmers can define rules for new predicates using the defrule construct. As in other logic-based pointcut languages [9, 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.

3. HALO WEAVER

Though the focus of this paper is to illustrate the HALO language, the reader might benefit from a little inside information on how HALO is implemented. We won't delve into all the details, but we do cover how HALO code and Lisp code are combined. The latter integration process is called *weaving* in AOP terminology and – in HALO's case – involves a runtime process for intercepting join points and a query engine for matching pointcuts to a join point. A schema of the dynamic weaving process, responsible for combining HALO code and base code, is depicted in Figure 2.

The weaver is responsible for mapping the key events in the execution of a CLOS program (or join points) 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[14] to attach code for generating the facts. Hence in order to generate join points for classes or generic functions, it is necessary to introduce a correct meta class at the class or generic function definition.

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 interpreter for our logic language HALO. A first implementation of the HALO query engine was based on backward chaining.This approach however relies on storing the entire execution history forever and making deep copies of object state at each point in time. This appeared to be necessary to evaluate hybrid pointcuts correctly using backward chaining [11]. However, currently, a more practical implementation of HALO relies on an implementation strategy based on forward chaining. Because of the way pointcut queries are then pre-computed as join point facts are asserted, it diminishes copying object state for evaluating `escape` conditions correctly. In addition HALO's predefined set of temporal relations allows us to build a weaver where memory management can be optimized. More concretely, we have implemented an extended version of the RETE [8] forward chaining algorithm to evaluate HALO pointcuts, which allows for a dynamic management of the join point history. In fact, the e-commerce application as presented in this paper runs on top of this implementation.

Switching HALO from backward chaining to forward chaining is however not trivial. More precisely, because of HALO's hybrid pointcut mechanism, allowing full variable sharing and recursive rule definitions appears problematic. However a full discussion on these problems is out of the scope of this paper, and will be discussed elsewhere.

4. IMPLEMENTING AN E-COMMERCE APPLICATION

The application we discuss in this paper is an e-commerce application for selling clothing. This application was first implemented on top of the Hunchentoot framework [24] and CLOS. Hunchentoot is a toolkit for building dynamic websites and is currently being used for commercial web sites such as ERGO [24]. Hence Hunchentoot is an excellent starting point for developing a *realistic* web application. The latter is important because we want to show that something as exotic as a temporal logic-based pointcut language is indeed useful and plausible to use in a real-life application.

In this paper we extend the e-commerce application with a *discounting* and *suggestions* feature using HALO. To get a better grasp of how the application works, we discuss the basic workings of an e-shop in the next section. Thereafter follows an overview of the basic program structure, referenced in the follow-up sections which discuss how aspects in HALO adapt the base program to implement the *discounting* and *suggestions* features.

4.1 Going shopping

Online Shopping at Boutique. The *Boutique* store exclusively sells clothing by mail order: the *Boutique* catalogue itself can be viewed on a website and customers can place their orders online. Though browsing the catalogue requires nothing more than surfing the website, placing an order requires one to identify oneself by filling in a login form. The latter can only be done by customers that have an account at *Boutique*. When logged in, the customer can add different products from the catalogue to his virtual shopping basket. The contents of a shopping basket can be viewed at a dedicated web page, displaying each added article, a number indicating the total price of all purchased articles and a checkout form asking for a desired payment method and packaging details. When the customer fills in the checkout form, the products are retrieved from the *Boutique* stock and mailed to the customer, along with the bill. Two screenshots of the *Boutique* website are depicted in Figure 3

Program structure. The application implementing the above e-shop is modelled as depicted in Figure 4. The figure displays a UML diagram of the classes implementing the different roles outlined in the above description of the e-shop. The class `shop` represents the *Boutique* store and the slots `articles`, `customers` and `accounts` are there to track the catalogue content, the customers visiting the *Boutique* website and the customer accounts respectively. When a customer accesses the *Boutique* homepage, he is represented as an instance of the class `user` and added to the shop's `customers` list. When the customer logs in, the `user` object is mapped to one of the accounts stored at the shop.

Accounts can be added to or removed from the shop using the methods `create-account` and `remove-account`. As described by the class `account` in Figure 4, an account itself consists of a unique `user-name` and `password`, as well as a `credits` number. The latter stores the total of all bills paid by the customer owning the account. As hinted before, customers are represented as instances of the class `user`.

The class `user` in the same UML diagram prescribes that all customers have their own shopping `basket`, a reference `account-id` to the account a customer logs in on and a session identifier. Logging in a user is done through the method *login* and results in setting the user's `account-id` – given that the provided user name and password match an existing account. The methods `buy` and `undo-buy` implement adding and removing a product from the user's basket. The `checkout` method is used to finalize a shopping session and results in removing the products cached in the `basket` (by means of the methods `add-article` and `remove-article` defined in the class `Basket`), decreasing the stock

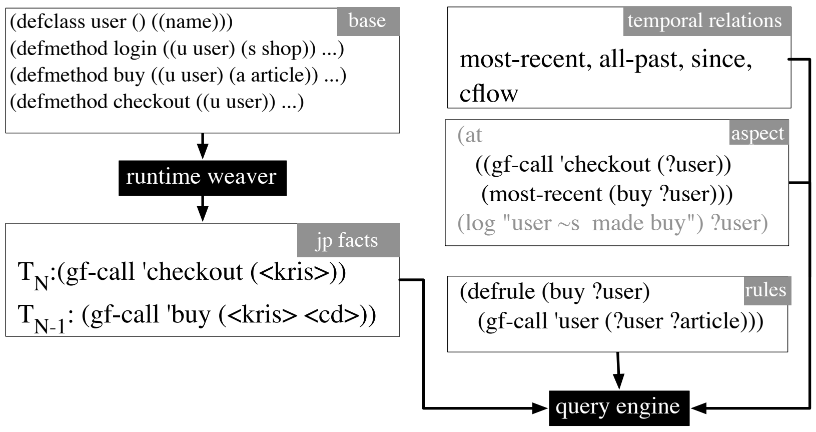
base
(defclass user () ((name)))
(defmethod login ((u user) (s shop)) ...)
(defmethod buy ((u user) (a article)) ...)
(defmethod checkout ((u user)) ...)
runtime weaver
jp facts
T_N:(gf-call 'checkout (<kris>))
T_{N-1}: (gf-call 'buy (<kris> <cd>))
temporal relations
most-recent, all-past, since, cflow
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

Figure 2: HALO weaver schema.

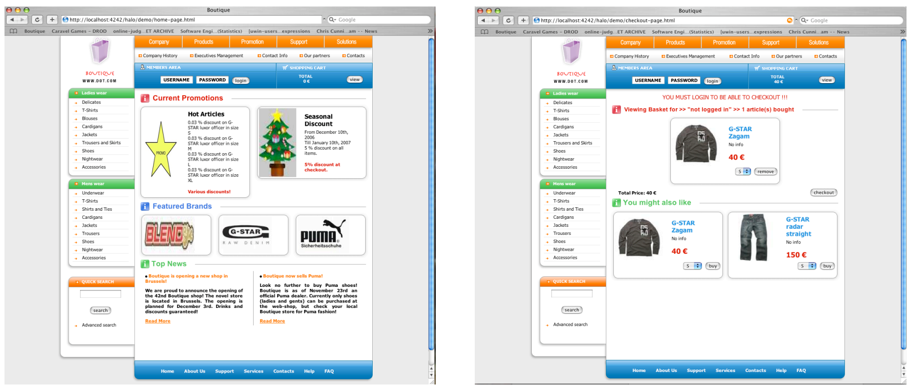

Figure 3: Screenshots of the e-commerce application.

number for each of these products in the shop (see method `decrease-nr-of-articles-for-size` in the class `Article`), and increasing the total amount purchased by the user. Finally, the class `Article` models a product from the *Boutique* catalogue. Each such article has a `name`, a `description`, a `price` and a `picture` and belongs to a `collection`: all these slots hold some information to be displayed when a customer browses the catalogue. The slot `quantity-in-stock-per-size` in the same class denotes the amount of each product stocked at `Boutique` per available size ("Small", "Medium", "Large" or "Extra Large").

Note that for conciseness we have omitted the classes and methods responsible for generating HTML code and that explaining the methods and slots left undiscussed so far, is done on as necessary in the following sections.

4.2 Adding Discounts and Suggestions

The e-commerce application discussed above implements the primary features of an e-shop. In this section we investigate how HALO can be used to add a *discounting* and a *suggestion* feature. Both features are discussed separately in the next section; For each we first discuss the feature's purpose, we continue by explaining what extra classes we need to implement the feature, and finally we show how the base application and the extra classes are linked using HALO. The latter discussion illustrates HALO's special properties such as temporal pointcut language, the ability to access Common Lisp from within the base language, rule abstraction, referencing past program state, etc. as defined in the previous sections.

4.2.1 Promotions and Discounts

Occasionally, *Boutique* launches a promotional campaign. For example, when it's Christmas, all customers get a discount on checkout or when stock of a particular product is piling up, *Boutique* offers a discount to customers willing to buy it. These kinds of promotions are referred to as *dynamic pricing strategies* [13].

Implementing the *discounting* feature requires us to adapt the e-commerce application to both make customers aware of promotions, as well as to make sure that discounts are processed when a bill is created. More concretely, when a customer visits the *Boutique* homepage, a list of currently active promotions pops up. When the customer logs in dur-

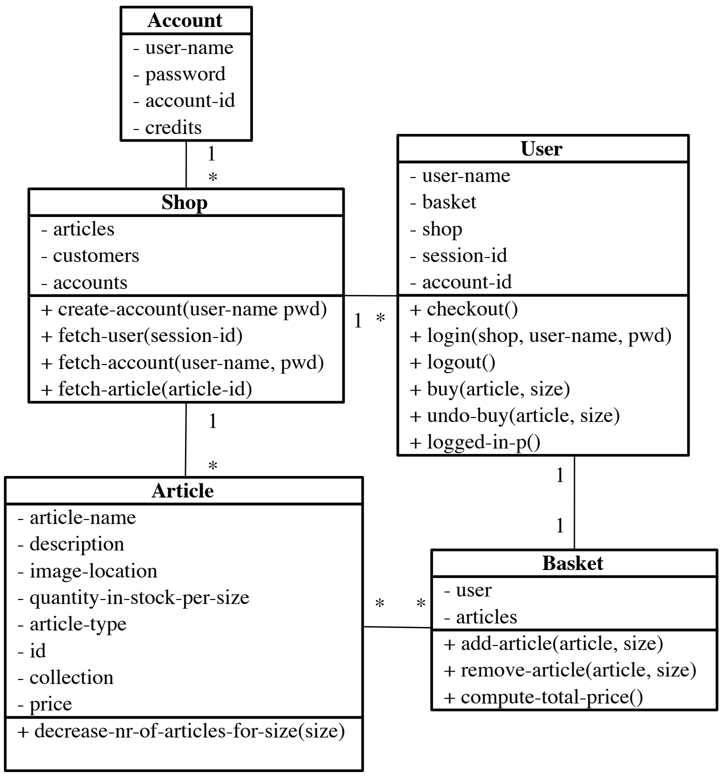

Figure 4: Sketch of a UML diagram for the e-commerce application.

ing a time at which a certain promotion is active, he should get the associated discount when he checks out his basket and the bill is computed – no matter whether the promotion is still active then. If the latter weren't true, the customer could login and be promised a promotion, and finally not get it when he checks out his basket.

The Promotion class. Promotions are modelled using the classes depicted in Figure 5. The class `Promotion` is an abstract class that outlines the interface that each promotion needs to implement: a method `promotion-rate` that computes the promotional rate for an article, a method `promotion-active-p` that can be used to see if a particular promotion is currently active or not, a method `promotion-info` that generates an informative text explaining a promotion and a method `promotion-banner` that returns a picture. The latter two methods are used to render a representation of the promotion for a customer visiting the `Boutique` website. The `Promotion` class stores extra information by means of a slot `title`, referring to the name of a promotion, as well as a slot `catch-phrase`. In Figure 5 we have depicted two concrete subclasses as well, namely `christmas-promotion` and `overflow-promotion`.

The class `christmas-promotion` implements a promotion where each customer is offered a (constant) seasonal discount between December 15th and January 15th. The class itself simply overrides the methods defined by its superclass `promotion`. Verifying whether a christmas promotion is active is done by checking whether the current time and date is between December 15th and January 15th. More concretely, this is done by the method `promotion-active-p`, which accesses the time through the system function `get-decoded-time`. As in our example a seasonal discount is constant for each *Boutique* product, the method `promotion-rate` simply returns a constant.

Another promotion is implemented by the class `overflow-promotion`. This class extends the class `promotion` with an extra slot `articles+treshold+rate`. The latter maps articles to a threshold and a discount rate. The threshold denotes a maximum of articles that is desired to keep in stock, whereas the discount rate is a rate specific to each kind of article. For example, yellow jackets of the brand "Winter" are highly in fashion at the moment, whereas a basic raincoat is an evergreen that always sells well. So for the "Winter" jacket a low stock is kept, which is refilled regularly. When that stock starts piling up, indicating that yellow "Winter" jackets are no longer in fashion, large discounts are given to get rid of them quickly. In the case of the classic raincoat however, it is okay for *Boutique* to stock up on lots of basic raincoats at once, and only give low discounts to boost sales. The methods `find-particular-rate` and `find-particular-threshold` defined in the class `overflow-promotion`, are there to retrieve discount rates and thresholds for a particular article. The method `promotion-active-p` verifies whether there currently is a stock overflow for a particular article by comparing the current stock number of the article to the threshold defined

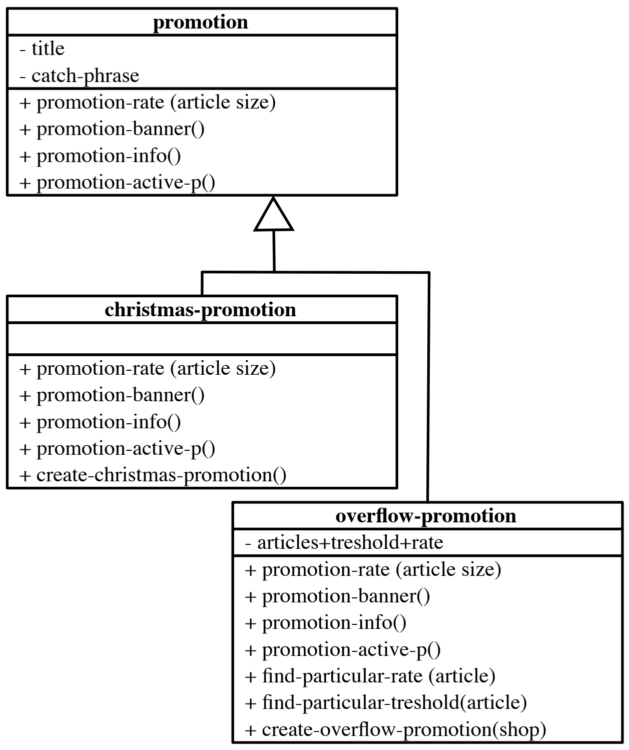

Figure 5: Sketch of a UML diagram for the Promotion class and subclasses.

by the promotion. In the overflow example, the discount rate for an article depends on the type of the article. Henceforth the method `promotion-rate` looks up this discount rate.

Linking Promotions using CLOS. In order to link the base application described in Section 4.1 and the promotion classes from the previous section, we need to adapt the behaviour of quite a few methods defined in the base application. We first discuss an implementation using only CLOS, and in the next paragraph we explain how linking the promotions and the base application can be done more concisely by means of HALO.

Clearly we need to extend the behaviour of the method `checkout` to incorporate calculating discounts for the products bought during a shopping session. Furthermore we must make sure that the discount rate for each article bought conveys with the discount rate that was active when the `login` happened. So we need to include computing and caching the discount rates for each article in the method `login`.

More concretely, in order to store the different discount rates we add a new slot `rates-per-size`. This slot maps articles to a discount rate and a size. Of course some extra methods need to be implemented to manipulate the slot `rates-per-size`, as well as a new class to create content for the mapping:

```
(defclass article+size+rate ()
  ((rate :initarg :rate :accessor rate)
   (size :initarg :size :accessor size)
   (article :initarg :article :accessor article)))

(defmethod add-rate-per-size
  ((user user) (article article) rate size)
  (setf (rates-per-size user)
        (cons
         (make-instance
           'article+size+rate
           :rate rate
           :size size
           :article article)
        (rates-per-size user))))

(defmethod reset-rate-per-size ((u user))
  (setf (rates-per-size u) '()))
```

In addition we extend the class `shop` with a slot `promotions` to keep track of the promotions at *Boutique*; Of course we also need to define an `:around` method for both the methods `create-christmas-promotion` and `create-overflow-promotion` that updates the `promotions` list of a `shop`.

Subsequently the method `login` is extended to compute the discount rate for each article stored in the shop, for each possible promotion and size, and saving the the rates for the user logging in. The latter is implemented using an `:around` method for the method `login`.

```
(defmethod login :after ((u user) (s shop) name pwd)
  (dolist (article (articles s))
    (dolist (promo (promotions s))
      (dolist (size (sizes s))
        (let ((rate
                (promotion-rate promo article size)))
          (when rate
            (add-rate-per-size u article rate size)
            ))))))
```

Next we include giving the discounts at `checkout`. For this purpose an `:around` method `checkout` is defined as depicted below. It retrieves the articles bought by a `user`, and gives a discount for each article for which a rate was cached at `login` time. The method `recompute-price` is a method we introduce to charge a customer the discounted price of an article.

```
(defmethod checkout :around ((u user))
  (let ((articles (articles (basket u))))
    (call-next-method)
    (dolist (a (rates-per-size u))
      (let ((size (size a))
            (article (article a))
            (rate (rate a)))
        (when (find-if
                (lambda (p)
                  (and (equal (cdr p) size)
                       (equal (car p) article)))
                  articles)
        (recompute-price u article rate)))))
```

Though the code in this section works perfectly for combining the promotions and the base application, it still requires the introduction and extension of quite a few methods. Due CLOS built-in `:after`, `:before` and `:around` capabilities, there is no need to produce tangled code. One could argue however that the implementation is scattered because the code to make sure a customer gets his discounts covers multiple different methods and classes. More importantly, we also observe that a lot of code is concerned with recording/restoring program state (such as the discount rates). In the next section we discuss how HALO allows a more declarative implementation, where the storing and retrieving of discount rates is automated. In fact, in HALO, this code can be modularized by means of a single piece of advice and adding a single new method.

Linking Promotions using HALO. From the latter chain of adaptations, we can identify the join points of interest to be calls to the methods `login`, `buy` and `checkout`. The idea is to group the scattered code by means of a history-based pointcut, wherein temporal relations are used to interconnect join points and in addition to make use of the `escape` mechanism in HALO to refer to the past discount rates [1]:

```
(defrule
  (forall-articles-in-basket ?user ?article ?size)
    (all-past
      (create article ?article)
      (member ?size ("S" "M" "L" "XL")))
    (escape
      ?member (in-basket-p (basket ?user)
                           ?article
                           ?size)))

(at
   ;; pointcut
   ((gf-call checkout ?user)
    (forall-articles-in-basket ?user ?article ?size)
    (since
      (most-recent
        (gf-call login ?user _ _ _)
        (current-discount-rate
          ?user ?article ?size ?rate))
       (all-past
         (gf-call buy ?user ?article ?size))))

   ;; advice code
  (recompute-price ?user ?article ?rate))

(defmethod
  recompute-price ((u user) (a article) rate)
  ...)
```

Roughly said, the piece of advice can be read as: "*when* a `checkout` call happens, execute the method `recompute-price` on a user object `?user`, for all articles `?article` on which `buy` *was* called, given a discount rate `?rate`. In addition, the `buy` calls need to have happened since the most recent call to `login` for the user `?user`. And also, the discount rate `?rate` is the one active for the different articles `?article` when the `login` happens."

More concretely, the pointcut consists of three pointcuts interconnected by means of the `since` temporal operator. The outer pointcut of the `since` operator, namely `((end-gf-call checkout ?user ?res) (forall-articles-in-basket ?user ?article ?size))`, simply employs the built-in predicate `gf-call` to capture a call to the method `checkout` and exposes all articles in the shopping basket at checkout time through `(forall-articles-in-basket ?user ?article ?size)` [2]. The two arguments of the `since` operator each capture a series of join points in the past of the latter outer pointcut. For example, the pointcut put as the first argument, captures the *most recent* join point that matches `(gf-call login ?user)` and computes all past discount rates (bound to `?rate`) by means of `(current-discount-rate ?user ?article ?size ?rate)`. The latter is possible as the logic variable `?rate` is left unbound by any of the pointcuts capturing execution join points. Note that `current-discount-rate` is not a predefined predicate.

The predicate `current-discount-rate` is defined as a separate rule:

```
(defrule
  (current-discount-rate ?user ?article ?size ?rate)
  (all-past (promotion-create ?promo))
  (all-past (article-create ?article))
  (article-rate ?promo ?article ?rate ?size))
```

As explained in Section 3, verifying whether a user defined predicate holds at a join point, means the body of the rule, basically also a pointcut, must hold at the join point. So (current-discount-rate ?user ?article ?size ?rate) holds at a `login` join point, if

1. *all* join points matching `(promotion-create ?promo)` *past* the `login` are captured
2. *all* join points matching `(promotion-create ?promo)` *past* the `login` are captured
3. (article-rate ?promo ?article ?rate ?size) is computed at the `login`

Note that these three conditions themselves are all defined as separate rules. `promotion-create` is a predicate that captures join points representing instance creations of classes that are subclasses of the class `promotion`. `article-create` simply captures instantiations of the class `article`:

```
(defrule (promotion-create ?promo)
```

[1]For brevity we have omitted the name of certain logic variables, and replaced them by a _ as they are not relevant in matching the pointcut.

[2]Note that the predicate `forall-articles-in-basket` is defined a separate rule. Matching rule definitions is explained further down the text.

```
1 (create overflow-promotion <promo>)
2 (create article <jacket>)
3 (create article <t-shirtt>)
4 (create article <trousers>)
5 (create article <socks>)
6 (gf-call 'login <kris> <boutique> "kris" "kg")
where the promotion rates given <promo> are:
0.20 for <t-shirt> in size "M"
0.25 for <trousers> in size "M"
7 (gf-call 'buy <kris> <trousers> "M")
9 (end-gf-call 'checkout <kris> <res>)
10 (gf-call 'logout <kris>)
11 (gf-call 'login <kris> <boutique> "kris" "kg")
where the promotion rates given <promo> are:
0.05 for <jacket> in size "M"
0.10 for <socks> in size "M"
12 (gf-call 'buy <kris> <t-shirt> "S")
13 (gf-call 'buy <kris> <jacket> "M")
14 (gf-call 'buy <kris> <socks> "M")
15 (end-gf-call 'checkout <kris> <res>)
```

Figure 6: A sample history of join points (to simplify the example, only generic function calls are considered; Note also that there is always a default promotional rate of 0.0 for any article, but for conciseness we've also omitted this from the trace).

```
  (create ?class ?promo)
  (escape ?sub (sub-class-p ?class promotion)))

(defrule (article-create ?article)
  (create article ?article))
```

The predicate `article-rate` is defined in terms of the `escape` predicate, so that `?rate` matches the result of calling the method `promotion-rate` on `?promo`, `?article` and `?size`. `member` simply enumerates the elements of the list `("S" "M" "L" "XL")`.

```
(defrule (article-rate ?promo ?article ?rate ?size)
  (member ?size ("S" "M" "L" "XL"))
  (escape ?rate (promotion-rate ?promo ?article ?size)))
```

Matching pointcuts. To further clarify the way the pointcuts are matched, we explain which advices are executed given the sample execution trace shown in Figure 6. Note that we use the notation `<name>` to denote object identifiers (e.g. `<t-shirt>` represents an object, obviously intent to be an instance of `article`).

Recall the first pointcut we discussed first in this section:

```
(at
  ;; pointcut
  ((end-gf-call checkout ?user _)
   (forall-articles-in-basket ?user ?article ?size)
   (since
     (most-recent
       (gf-call login ?user _ _ _)
       (current-discount-rate
         ?user ?article ?size ?rate))
     (all-past
       (gf-call buy ?user ?article ?size))))
```

It has multiple solutions for join point 15, one for each article the user checking out ever bought and for which there was a promotion active when the user last logged in (the articles `<jacket>` and `<socks>` bought by user `<kris>`).

In more detail, this pointcut is matched at join point nr. 15, because it matches the outer pointcut `(end-gf-call checkout ?user _)`, and exposes the discount rate of all `buy` join points (namely nr. 13 and 14) that match the second argument of the since predicate, since the last `login` join point that matched the first argument of the since predicate (join point nr. 11). Due `(forall-articles-in-basket ?user ?article ?size)` only `buy` events are captured of articles that are still in the user's basket (so no articles that were "unbought" (cf. `undo-buy` in Figure 4)). In addition the `buy` join points and the `login` join point are matched in the past of the `checkout` join point. Because of the (forall-articles-in-basket ?user ?article ?size)

Note that the exposed discount rates are the ones active at the latter `login` join point. This discount rate is exposed by means of the predicate `current-discount-rate`, for which we repeat the definition below:

```
(defrule
  (current-discount-rate ?user ?article ?size ?rate)
  (all-past (promotion-create ?promo))
  (all-past (article-create ?article))
  (article-rate ?promo ?article ?rate ?size))
```

At join point nr. 11 this rule has two matches (exposes rates 0.05 and 0.10 for the articles `<jacket>` and `<socks>`). At join point nr. 11 the first pointcut `(all-past (promotion-create ?promo))` in the body of the rule matches four join points, namely all instance creation join points of the class `article` (so nrs. 2 through 5) predating the login join point. The second pointcut in the body of the rule, namely `(all-past (article-create ?article))` matches a single join point: the creation of the object `<promo>` (nr. 1). Finally, given those join points, the last pointcut in the body of the rule, namely `(article-rate ?promo ?article ?rate ?size)`, filters out the join points nrs. 2, 5 and 1 because there is only a promotion active for the articles `<jacket>` and `<socks>`.

4.2.2 Suggestions

In order to try and trick customers into buying more, *Boutique* suggests customers products they might like to buy. It has been observed by the *Boutique* owner that customers mostly buy clothing from the same collection: so if they buy a shirt from one collection, they are likely to buy trousers from the same collection. In addition, customers that like certain products, are perceptive to products bought by customers that buy similar articles. For example, if one customer buys a jacket from the "Winter" collection and a

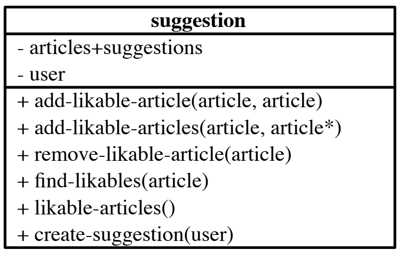
suggestion
- articles+suggestions
- user
+ add-likable-article(article, article)
+ add-likable-articles(article, article*)
+ remove-likable-article(article)
+ find-likables(article)
+ likable-articles()
+ create-suggestion(user)

Figure 7: Sketch of a UML diagram for the Suggestion class.

pair of trousers from the "Autumn" collection, then another customer buying the same jacket, probably also desires the trousers. So when customers purchase an article, the *Boutique* website pops up a list of suggestions, which all somehow "match" the article just bought.

The Suggestion class. Suggestion lists are modelled by means of the class `suggestion` in Figure 7. The class defines a slot `articles+suggestions` that maps each article to a list of articles that "match" the article (e.g. articles of the same collection). The slot `user` simply keeps a reference to the customer for whom the suggestion list applies. The method `add-likable-article` adds a suggested article for an article. Conversely, the method `remove-likable-article` deletes an article and its derived suggestions. The method `find-likables` is used to retrieve the list of suggested articles associated with an article, whereas the method `likable-articles` returns a concatenation of simply all suggested articles. Finally, the method `create-suggestion` can be used to initialize a new `suggestion`, given a user.

HALO code. Integrating the *suggestion* feature can lead to scattered and tangled code. For example, making sure that articles bought by one user are added to the suggestion list of another, requires one to adapt the behaviour of the `checkout` method to record per user the list of articles bought. Then we need to adapt the `buy` method to query this record of bought articles per user to compute a new list of suggested articles and to update another user's suggestion list. In this section, we consider a modularized implementation in HALO.

In order to make sure that each user even has a suggestion list, each instantiation of the class `user` triggers creating a fresh suggestion list. The piece of advice below consists of a pointcut that captures each instantiation of the class `user` and a piece of advice code that triggers a call to the method `create-suggestion` (see Figure 7):

```
(at
 ((create  user ?user))
  (create-suggestion ?user))
```

Though the latter makes sure that each user has a unique suggestion list, this suggestion list still needs to be updated regularly. The suggestion list needs to be updated each time the user adds a new article to his basket. For example, as suggested articles, we can add each article from the same collection to the user's suggestion list. The pointcut below consists of two pointcuts interconnected by means of the temporal operator `most-recent`. The outer pointcut captures all calls to the method `buy` through use of the built-in predicate `gf-call`. The inner pointcut then captures the *most recent* join point that matches `(end-gf-call create-suggestion ?user ?suggestions)` – so basically given a `buy` call on a user object, the most recently created suggestion list for that user. The latter is acceptable, as a suggestion list is unique for each user. Finally, the advice code simply computes a list of suggested articles and adds them to the captured suggestion list (logical variable `?suggestion`). Note that the variable `*boutique*` simply refers to an instance of the class `shop`.

```
(at
 ((gf-call buy ?user ?article ?size)
  (most-recent
    (end-gf-call
      create-suggestion
      ?user
      ?suggestions)))
 (add-likable-articles
   ?suggestions
   ?article
   (compute-articles-same-collection
     *boutique*
     ?article)))
```

Furthermore each suggestion list needs to be updated as soon as the user for whom it exists, removes an item from his basket; If he isn't interested anymore in an article, then he probably isn't interested in the articles that "match" it either. The pointcut below is similar to the one we perviously described and simply wraps calls to the `undo-buy` method to include updating a suggestion list.

```
 (at
  ((gf-call  undo-buy ?user ?article ?size)
   (most-recent
     (end-gf-call  create-suggestion ?user ?likables)))
  (remove-likable-article ?likables ?article))

;; at checkout reset likables

(at
  ((gf-call  checkout ?user)
   (most-recent
     (end-gf-call  create-suggestion ?user ?likables)))
  (setf ( likables ?user) ( create-suggestion ?user)))
```

As a final example, we consider a variation on the *suggestion* feature – which as explained below exploits HALO's hybrid pointcut mechanism. We recall that there are people that think alike and tend to enjoy similar things. The piece of advice below makes sure that this knowledge is used to suggest products to one customer, based on what another one bought. Relative to the pointcut (end-gf-call buy ?user ?article ?size ?res), there are two other pointcuts that need to be true, might the entire pointcut match a join point. Of particular interest is the second form that starts with most-recent, as this makes use of the escape predicate. That condition computes the content of a user basket at the time the method checkout is computed. This makes sure that when the entire pointcut is actually matched by a buy method call, ?articles is bound to that past content of a user basket.Though of course, the content of the basket by then will have changed.

```
(at
 ((end-gf-call  buy ?user ?article ?size ?res)
  (most-recent
    (end-gf-call create-suggestion ?user ?suggestions))
  (most-recent
    (gf-call  checkout ?user2)
    (escape ?articles (fetch-articles (basket ?user2)))
    (all-past (gf-call  buy ?user2 ?article ?size2))))
  (add-likable-articles ?suggestions
                        ?article
                        (quote ?articles)))
```

4.2.3 Discounts revisited

Below we have depicted an implementation of the *discounting* feature from Section 4.2.1 that makes use of the possibility to write escape conditions in a pointcut that make use of logic variables bound in a "future" pointcut. The latter feature allows us to implement the *discounting* by means of a single piece of advice:

```
(at
  ;; pointcut
  ((end-gf-call checkout ?user _)
   (since
      (most-recent
        (gf-call login ?user _ _ _)
        (all-past (promotion-create ?promo))
        (escape ?rate
          (promotion-rate ?promo ?article ?size)))
        (all-past (gf-call buy ?user ?article ?size))))
  ;; advice code
  (recompute-price ?user ?article ?rate))

  ;; definition of promotion-create as before
```

More concretely, the escape condition to calculate the discount rates at login time, makes use of the variables ?article and ?size, which are bound by the pointcut capturing calls to the method buy. The latter buy calls happen *after* the login call, and as discussed in Section 3 in order to make sure that the escape condition is evaluated against the correct program state, the weaver must make a copy of all objects at the time the login join point occurs. For this purpose, the weaver makes use of the generic function copy. Making deep copies of objects is costly in general; The idea is that the HALO programmer can specialize the generic copy to define what parts of which objects are relevant to copy. In the example at hand, we define a version of copy specialized on the class article that backs up the stock number per size of an article (see the slot quantity-in-stock-per-size in Figure 4):

```
(defmethod copy ((a article))
  (let ((article-copy (make-instance 'article))
        (stock
          (copy-alist (available-stock-per-size a))))
    (setf (available-stock-per-size article-copy)
          stock)
    (setf (article-name article-copy)
          (article-name a))
    (setf (description article-copy)
          (description a))
    ;; set other slots as well
    ))
```

The HALO code from this section is more concise than the code explained in Section 4.2.1 because it uses variables in escape conditions, bound by "future" pointcuts. For example, it is not necessary to define the predicates current-discount-rate, article-rate or article-create, nor to refer to them within the pointcuts. Furthermore, control over the weaver is possible to dictate what object state needs to copied through the generic copy.

5. RELATED WORK

5.1 AOP and Lisp

HALO is the only implementation of an aspect-oriented programming language for a Lisp dialect that allows writing pointcuts based on the execution history of the application. To the best of our knowledge it is also the first logic-based approach to aspect-oriented programming in a Lisp dialect. A framework for aspect-oriented programming in Scheme was proposed by Tucker et. al. [23] , where the notion of AspectJ-like pointcuts and advices are introduced in Scheme. The focus of the research by Tucker et. al. is to define the particularities of combining higher-order functional languages and aspect-oriented programming, such as the treatment of scope, the lack of "names", etc. The latter is not the focus of HALO; Furthermore Tucker et. al. unify aspect-oriented programming with AspectJ [16] in their research and do not consider expressing history-based aspects. AspectL [4] is a library that provides aspect-oriented extensions for Common Lisp and CLOS. These extensions include support for writing *generic pointcuts*, *destructive mixins*, *special classes* and *special functions*. However, there is no explicit means in AspectL for expressing history-based aspects.

5.2 History-based Aspects in Other Languages

A closely related approach to our work is Alpha [21], a logic-based pointcut language for a Java-like language. 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 the use of standard Prolog only allows interaction with the base program at the current join point. So (as discussed in Section 4.2.1), this means the "past rate discounting" aspect must be expressed as two pointcuts and advices. 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 in allowing hybrid pointcuts to interact with the base program.

Tracematches [1] and J-LO [3] are extensions of AspectJ and hence use Java as the base language. History-based pointcuts are expressed in Tracematches as regular expressions over AspectJ pointcuts, and in J-LO as temporal logic formulae over AspectJ pointcuts. A different temporal logic is used in J-LO than the one in HALO. Rather than relying on AspectJ or a similar, existing non-logic pointcut language for Lisp and then using temporal logic to turn it into a history-based pointcut language, HALO uses the same logic formalism throughout.

5.3 Hybrid Aspects

OReA [5] is a family of logic-based pointcut languages for Smalltalk, in which the concept of "hybrid aspects" was originally introduced. The prime objective in this work was for "hybrid advices" 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 changed 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 can be translated to an `escape` condition. OReA also supports interaction from the base and advices 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 forward-chaining-based variant. But this 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 joinpoints.

6. FUTURE WORK

There still remains work to be done on behalf of HALO. Future research on HALO's language design and library support is still necessary, to adhere to some known (open) issues in aspect-oriented programming. In addition, HALO suffers from some issues that relate to its symbiosis with Common Lisp.

Language design. HALO does not currently feature predicates that offer a static model of the base application, as in other logic-based pointcut languages [9, 21]. Predicates for such a model could be easily added, and is a question of setting up a standard library in HALO. In addition, HALO does not fully support *obliviousness* [7], which is to be thought one of the main requirements for aspect-oriented programming. *Obliviousness* dictates that the base program should not be adapted nor be prepared so that it can be combined with an aspect program. Due the explicit meta classes HALO programmers employ to alert the weaver of possible join points of interest, HALO breaks *obliviousness.* One option could be to redefine the `defclass`, `defgeneric`, `defmethod` macro's to simply include setting the appropriate meta classes. However this would create a tremendous overhead – think of all the irrelevant join points being generated. Another option would be to try and statically derive from the pointcuts which class definitions and generic function definitions need to be tagged with a meta class. In order to make a full aspect-oriented programming language out of HALO we need to introduce an *aspect composition language* [20]. Such a language is necessary to resolve conflicts between aspects that apply at *shared join points.* Current composition languages rely on constructs for expressing an ordering between aspects. We are thinking along the same line for HALO, though for uniformity we plan to devise a composition language based on temporal logic.

Hybrid pointcuts. Furthermore some open issues remain to be explored concerning HALO's hybrid pointcut mechanism. The latter allows interaction with Common Lisp from within the pointcuts and is in fact a language symbiosis mechanism. Up till now we have implicitly considered `escape` conditions to be side effect-free expressions. However behaviour is currently undefined when expressions with side effects are being used. Should the side effect occur when the entire pointcut is matched, or as soon as the `escape` condition can be evaluated? In addition what does it mean when `escape` is used in recursive rules?

7. CONCLUSIONS

In this paper we validated the suitability of the logic-based pointcut language HALO to modularize crosscuts as history-based aspects. For this purpose, an overview of the temporal logic-based aspect language was given, discussing both advice and pointcut language. A large part of this paper consists of illustrating the use of HALO for extending an e-commerce application with a *discounting* and a *suggestions* feature. Both features require the implementation of some additional classes and linking these with the base program can be expressed in terms of the execution history of the e-commerce program. Therefore they can be expressed as history-based aspects in HALO. From these experiments it has become clear that HALO's ability to define pointcuts that refer to past program state result in more concise code, because it automates storing and referencing program state at past join points.

8. REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. 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] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common lisp object system specification. *Lisp and Symbolic Computation*, 1(3-4):245–394, January 1989.

[3] E. Bodden. J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen

university, 2005.

[4] P. Costanza. A short overview of aspectl. In *European Interactive Workshop on Aspects in Software (EIWAS '04)*, Berlin, Germany, September 23 –24 2004.

[5] M. D'Hondt and V. Jonckers. Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development*, 2004.

[6] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. *Lecture Notes in Computer Science*, 2192:170–184, 2001.

[7] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, Minneapolis, 2000.

[8] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, September 1982.

[9] 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.

[10] 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.

[11] C. Herzeel, K. Gybels, and P. Costanza. A temporal logic language for context awareness in pointcuts. In "Workshop on Revival of Dynamic Languages", 2006.

[12] R. Hirschfeld. Aspect-Oriented Programming with Aspects. In *Lecture Notes in Computer Science: Objects, Components, Architectures, Services, and Applications for a NetworkedWorld: International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7–10, 2002. Revised Papers*, 2003.

[13] A. Kambil and V. Agrawal. E-commerce: The new realities of dynamic pricing. In *Outlook journal*, July 2001.

[14] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[16] I. Kiselev. *Aspect-Oriented Programming with AspectJ.* Sams, 2002.

[17] R. 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.

[18] R. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, 1979.

[19] C. Lopes, E. Hilsdale, J. Hugunin, M. Kersten, and G. Kiczales. Illustrations of crosscutting. In P. Tarr, M. D'Hondt, C. Lopes, and L. Bergmans, editors, *International Workshop on Aspects and Dimensional Computing at ECOOP*, 2000.

[20] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution.* Phd thesis, IPA, May 2006.

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

[22] É. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-aware aspects. *Lecture Notes in Computer Science, Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, 4089:227–242, 2006.

[23] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 158–167, New York, NY, USA, 2003. ACM Press.

[24] E. Weitz. Hunchentoot - the common lisp web server formerly known as tbnl. http://weitz.de/hunchentoot/.

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