A Temporal Logic Language for Context Awareness in Pointcuts

Charlotte Herzeel, Kris Gybels, Pascal Costanza
{charlotte.herzeel, kris.gybels, pascal.costanza}@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel

Abstract

Aspects based on join points that occurred in the execution history of a program provide a powerful way to make applications aware of their context. We present HALO, a logic metaprogramming approach based on temporal logic, that is designed with context-awareness in mind. A number of illustrative examples demonstrate HALO's expressivity, including expressions about past events which contain variables only bound in their future.

1 Introduction

In a good modular design, program concerns are decomposed into packages, classes, methods, etc., so that each module implements a different concern. Some concerns, however, are crosscutting, which means that their implementation is typically scattered over different modules. Aspect-oriented Programming (AOP) focuses on the modularisation of such crosscutting concerns [7]. An AOP language provides a notion of join points that are events in the execution of a program, a pointcut language that allows declarative description of sets of join points, and advices that effect the program behavior before, after or around the join points captured by a pointcut.

The notion of context-aware aspects has been introduced in earlier work by Tanter et al. [11]: Related to the idea of context-aware applications, context-aware aspects are aspects whose behavior is context- dependent, which includes the possibility to restrict aspects to certain contexts. In that paper, the authors address the problem that current AOP languages incorporate a too limited notion of context when considering context-aware applications (e.g. context is only information on the join point and not on the state of the whole program) and that therefore the definition of context needs to be extended, for example to include the ability to refer to *past* contexts. The use of a specialized pointcut language is proposed, but only back-end technology based on Reflex is discussed.

In this paper, we introduce HALO, a new pointcut language designed with context-awareness in mind. In the next two sections, we introduce the notion of context-aware aspects using an example and give a brief overview of the extensions that were made to the Reflex framework to support them. Section 4 introduces HALO, discusses how it can be used to implement the example context-aware aspects and outlines some issues with weaving. We end the paper by discussing the related work and stating our conclusions.

2 An E-shop Application

Consider a simple e-commerce application. A shop has customers and sells articles. Customers have an account and have to login to put shop articles in their basket and to checkout their basket. The UML diagram is shown in Figure 1.

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

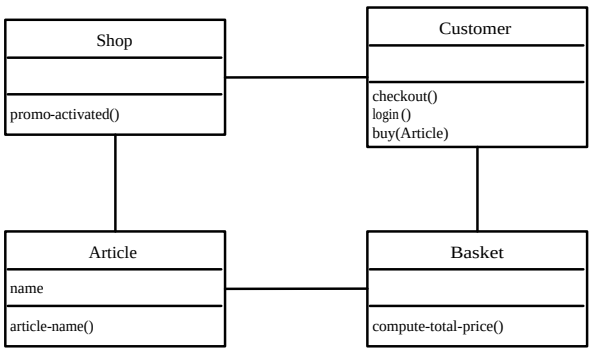

Figure 1: E-commerce application

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

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

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

To illustrate the program's desired behavior, consider a sample program run depicted in Figure 2. There is a timeline depicted for two users: we see that there is temporarily one promotional context active, namely `seasonal-promo`. On the timeline, we see that user 1 logs in when the context `seasonal-promo` is active: At that time, the website is popping up banners to lure customers to login (e.g. "Login now and get a discount on checkout!"). The idea is that when user 1 checks out at a time the `seasonal-promo` context is no longer active, user 1 still gets her discount related to the `seasonal-promo` context. User 2 however does not get this discount, as she logged in when the promotional context was not active, but no harm done: The banners promising a discount were no longer being displayed when she logged in anyway.

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

In the next section we discuss the extensions that were made to Reflex to implement aspects such as these.

3 Context-Aware Aspects in Reflex

Tanter et al. have previously presented support for context-aware aspects in an extension of the Reflex framework [11]. Reflex is an open reflective extension of Java that supports both structural and behavioral modifications of programs. The core concept of Reflex is the link; although there are both structural and behavioral links in Reflex, we only need to explain behavioral links here. A behavioral link invokes messages on a metaobject at occurrences of operations specified by a hookset. A hookset, like a pointcut, is a composable entity that specifies a set of operations based on selection conditions. Hooksets express lexical crosscutting only (pointcut shadows), but activation conditions can be added to a link which express additional dynamic conditions.

The Reflex framework was extended with support for defining contexts, and for defining new context-specific link activation conditions. New contexts can be defined by subclassing the class `Context` and overriding the `getState` method. The method must return `null` which indicates the application is not currently in that context, or a `ContextState` object which indicates the context is active with certain parameters. An example of a parameter would be the rate of overflow for a stock overflow context. New link activation conditions that depend on contexts can be defined by subclassing `CtxActive`. A straightforward example is the

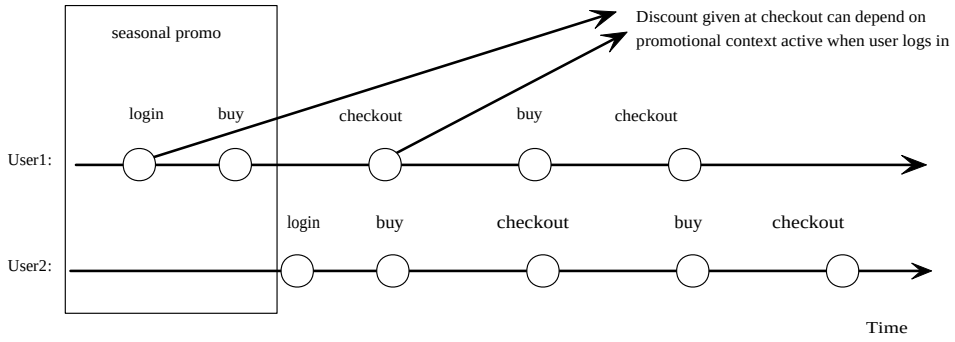

Figure 2: E-commerce application program run

subclass `CurrentlyInCtx`, it can be used for link activation conditions that simply check whether the application is currently in a particular context. The abstract subclass `SnapshotCtxActive` provides support for defining activation conditions that check if the application has been in a particular context in the past. It provides support for taking "snapshots" of the state of the context at particular points so that these can be checked later. Examples are instances of the `CreatedInCtx` class: These activation conditions on a link are true if the object in which the link intercepted an operation was previously created when a specific context was active.

While the extensions to Reflex introduce support for context-aware aspects and in particular the snapshotting of contexts necessary to refer to past contexts, the extension is in the back-end only and no dedicated pointcut language has yet been introduced. In the next section we introduce such a pointcut language, dubbed HALO.

4 HALO: A Temporal Logic Pointcut Language

We present HALO, a novel pointcut language based on temporal logic programming. This makes HALO similar to logic-programming-based pointcut languages like CARMA [5], but the use of temporal logic gives HALO additional operators to express temporal relations between conditions in a pointcut. A pointcut can thus refer to *past* join points and the context in which these occurred. In this section, we first introduce some basic notions of temporal logic and we then show how the aspects from Section 2 can be implemented using HALO. To conclude the section, we cover the details of a prototype implementation of HALO based on Common Lisp.

4.1 Temporal Logic

The term temporal logic [9] refers to all logics that allow the representation of temporal information, meaning the truth value of a formula changes over time. A temporal logic introduces so-called temporal operators in a logic (e.g. "always" and "sometime") next to the usual truth-functional operators (e.g. "and", implication, etc.) which can be used in rules to specify – on a high level – at what time a rule applies. HALO's design is based on *metric temporal logic* (MTL) [3] [2] because MTL introduces temporal operators in first order predicate logic (e.g. previous, sometime-interval, etc.) that allow one to connect formulae indicating that the second formula becomes true within a certain time-interval of time points from when the first formula becomes true: This idea can be used to describe an execution history as a sequence of events. We next explain how aspects, advices, pointcuts etc. are written in HALO.

4.2 The HALO Language

HALO defines predicates about join points, higher-order temporal operator predicates and a predicate `escape` which allows pure Lisp to be used as

logic conditions. These predicates are used to write pointcuts as logic queries. Advices are written simply as logic rules where the head of the rule specifies a Lisp function that will be invoked whenever a join point occurs that matches the pointcut and the body of the rule specifies the pointcut. Figure 3 gives an example in which the second rule defined using `defrule` is an advice, because its head specifies the predicate `advice` with the following signature:

1. A string that refers to a name for the advice.
2. A function symbol that refers to the method or function implemented at the base level (Lisp) to implement the advice.
3. A list of variables that denote the arguments for the function that implements the advice. This list can contain logic variables and ordinary Lisp values.

The join point model of HALO defines join points for two events in the execution of a program: a call to a function and the creation of an instance of a class. Two join point predicates can be used in a pointcut to capture these join points. The `call` predicate takes two arguments, being and meaning in order:

1. A string that names the method being called during program execution.
2. A list of logic variables and values that represent the arguments of the method being called.

The `create` predicate also takes two arguments, being in order:

1. A class symbol that refers to the class being instantiated during program execution.
2. A logic variable or value that refers to the created instance.

A temporal operator is a higher-order predicate that puts a temporal relation between the conditions used within the operator and the conditions used outside the operator. The temporal operators available in HALO are based on the operators in MTL and are `previous`, `sometime-past` and `sometime-interval`. Only the operators of MTL that refer to the past are adopted, while there are examples of pointcuts that refer to events in the future [6, 5], this creates some obvious conceptual and technical problems that we currently did not wish to focus on. One of the conditions inside the temporal operator should use a join point predicate, thus connecting the join point caught by the conditions outside the operator with one caught by the conditions inside the operator. The other conditions inside the operator can put conditions on the context in which that past join point should have occurred.

The meaning of the different temporal operators is as follows. When the operator `previous` is used to connect two pointcuts, this means the second pointcut – ergo the join points described by the pointcut – occurs exactly before the first pointcut, that is without any intervening join points. Using the operator `sometime-past` to connect two pointcuts implies that the events captured by the second pointcut occur sometime before the events described by the first pointcut. Note that only the most recent events matching that second pointcut are captured here. The operator `sometime-interval` allows one to write down a pointcut specifying the time-stamps for which the described join-points actually occur.

The `escape` pattern is a pattern with predicate name "escape" and two arguments:

1. A logic variable.
2. A Lisp-form (ergo a Lisp program).

It allows the programmer to bind a logic variable (first argument) to a value retrieved from evaluating a Lisp-form at the base level (second argument). This Lisp-form can contain logic variables. For example, `(escape ?Nr (nr-of-gifts ?Shop))` means that a method `nr-of-gifts` is called on a logic variable `?Shop`, which has to be bound to an object that understands the message. The result of this method call (a Lisp value) is bound to the logic variable `?Nr`.

4.3 The E-commerce Application in HALO

Let us take another look at the e-commerce application from Section 2 and see how we can implement some sample aspects based on the possible scenarios for assigning and processing a promotion

by using the HALO language defined in the previous section.

A first example is Figure 3. It states that a user gets a 10% discount when she checks out, and at that time there is seasonal promotion active.

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

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

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

Figure 3: Aspect 1

The implementation of the advice, the discount method, is a plain method and can of course be reused in another aspect definition. Note also that the promotion context is defined as a separate rule (`seasonal-promo`) and can also be used to implement a second aspect depending on that promotion context (Figure 4).

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

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

Figure 4: Banner aspect.

A second aspect is illustrated in Figure 5. This example illustrates that not only past events are captured, but also past values: The activation state of the shop is the state it had when the user logged in. The example is made more interesting by giving customers a gift, but only when there are still gifts left. This refers to the *current* state of the shop. The combination of reasoning about past and current values does not pose any problems.

We present another example in Figure 6. The third aspect is more complex than the previous examples because inside the sometime-past operator we refer to a variable `?Article` that is bound in the call form for the buy event. The buy event takes place after the login event but the pointcut for the login event puts a condition on a variable bound by

```
;; Give a customer a free gift when she checks out,
;; as longs as she logged in when the promotions
;; were activated, and the gifts are not depleted.

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

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

Figure 5: Aspect 2

the buy event. This particular example shows off the expressivity of HALO but imposes some difficulties on the weaver (Section 4.4).

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

;; CONTEXT definitions
(defrule (stock-promo-active (?Shop ?Article))
  (stock-overflow (?Shop ?Article)))

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

Figure 6: Aspect 3

4.4 Implementation Details

HALO is a language extension for Common Lisp. The implementation uses the metaobject protocol (MOP) of the Common Lisp Object System (CLOS) and the Prolog interpreter by Peter Norvig [8]. We next discuss the problems for making the system work and present the solutions. Note, however, that we do not consider the technical details but rather provide a general idea of the implementation.

4.4.1 Weaving

On each method call or instance creation, a fact has to be recorded in the logic repository so that pointcuts can reason about it in the future. This is achieved by use of the CLOS MOP, which allows extending the default CLOS semantics. Facts about such events are stored in the form `(call T ...)` or `(create T ...)` where T stands for a timestamp which is generated by a global event counter.

The weaver for HALO is by necessity a dynamic weaver because pointcuts can depend on information about events that is only available at runtime, e.g. the values of the arguments of specific method calls. Again, the default CLOS semantics are extended to include queries of the logic repository for applicable advices and their possible execution.

4.4.2 Implementing temporal operators

Before a rule is added to the logic repository, the rule is compiled to Prolog analogous to the translation from MTL to first order logic defined in [2]. The basic idea is as follows. Each predicate gets an extra argument, namely a temporal variable, that keeps a spot to store the timestamp to which a formula is evaluated. The temporal operators are compiled away by putting constraints on this argument. As an example, consider the following formulae compiled to Prolog.

```
MTL

1. (A ?a) :- (sometime-past B ?b)
2. (A ?a) :- (prev B ?b)
3. (A ?a) :- (sometime-interval a b B ?b)

Prolog

1. (A ?t ?a) :- (B ?k ?b),(< ?k ?t)
2. (A ?t ?a) :- (B ?k ?b), ?k is ?t - 1
3. (A ?t ?a) :- (B ?k ?b),(< ?k, ?t),(< ?k b),(> ?k a)
```

Of course, there are subtleties in compiling nested operators such as generating new temporal variables, maintaining order for temporal variable constraints, etc., but these details are outside the scope of this paper.

4.4.3 Saving object State

Rules that put constraints on variables used inside a temporal operator are not always bound when the event happens. For example, take aspect 3 (Figure 6) of the previous section. It states that a user gets a discount on an article if there was a stock overflow for that particular article at the time the user logged in. The problem is that when the login event happens, we cannot possibly know at that time what article the user will buy. So we cannot compute the result of the promo-activated condition. This variable is bound when the buy event takes place. However at that time we must evaluate the promo-activated condition in respect to the state of the shop when the user logged in.

Therefore, whenever a method call is recorded with such advices, as can be derived from the rule definitions, a deep-copy is taken from the arguments and this copy is used to evaluate the promo-activated condition later on. This is similar to the notion of taking "snapshots" of the system state in the Reflex-based extension described in Section 3.

4.4.4 Garbage Collection

It is not very economic to store each method call and instance creation forever because it is very likely that the system runs out of space as many methods are called during program execution. Luckily, we observe that certain facts become obsolete after a new fact is added. For example, say we have only one logic rule.

```
(A ?a) :- (sometime-past (call B ?b))
```

Recall that the semantics of the sometime-past operator is such that *only* the most recent `(call B ?b)` event will be matched to the `(sometime-past call B ?b)` goal. Even if a system generates many such events only the last one will ever be used to resolve a query `(?- A ?a)`. So we can throw away all `(call B ?b)` facts except the last one.

It becomes a little more complicated if we put conditions on the arguments of the facts:

```
(A ?a) :- (sometime-past (call B ?b) (> ?b 5))
```

The goal `(sometime-past (call B ?b) (> ?b 5))` will match the last `(call B ?b)` event that meets the requirement `(> ?b 5)`. So we must keep the last `(call B ?b)` event that meets the constraint, and not simply the last unconstrained event.

There is a problem when dynamic constraints are defined.

```
(A ?a) :- (sometime-past (call B ?b) (> ?b ?a))
```

We cannot determine which `(call B ?b)` events fulfil the `(> ?b ?a)` constraint because the value of `?a` is not fixed. Therefore, we cannot decide which `(call B ?b)` facts become obsolete. In simple cases, we can of course see if the arguments of successive facts are equivalent: If the `?b` arguments are simply bound to numbers, we know only one `(call B 5)` has to be kept.[1] However in general, if the `?b` variable is bound to complex values, such as objects, this is not obvious.

We have implemented a program to analyze the logic rules, checking the above cases. The rule-analysis also takes nesting of temporal operators into account etc. The static analysis of rules provides the conditions the garbage collector has to check to decide whether a fact can be garbage-collected or not.

5 Related Work

Alpha [10] is also a logic-programming-based approach to AOP and it is possible to reason about the execution history: Events such as method calls are represented by relations that have explicit time stamps. A number of predicates is defined to express the order between time stamps and these are used to express the order of events. In our approach however, temporal operators are used to connect *entire* pointcuts, creating a temporal context for the entire pointcut expression: This allows us to easily express that certain conditions must hold at the time a specific events occurs.

J-LO [1] is a tool for checking temporal assertions in Java programs. These temporal assertions are written down as LTL formulae in the form of Java annotations in the source code. LTL is a logic that extends propositional logic with temporal operators to arrange propositions on a timeline. In J-LO, AspectJ pointcuts are the propositions, so it is in fact possible to describe sequences of events; J-LO also supports a free-variables mechanism, which can be used to refer to variables bound earlier in the execution history, but not to variables bound at a later stage in the execution history, as is possible in HALO.

Douence et. al. [4] formally defined event-based aspect oriented programming (EAOP) as a general framework for AOP in which they define aspects in terms of sequences of events emitted during program execution (e.g. a method call) and crosscuts are defined in terms of sequences of such events. Currently, they have implemented an EAOP tool which is basically an OO framework for writing EAOP applications in Java: with this approach they do not offer a pointcut language, but rely on the programmer to manually generate and match events using built-in methods.

[1] Static typing of logic variables might come in handy here.

6 Summary and Future Work

In this paper, we introduced HALO, a new pointcut language based on temporal logic, that is explicitly designed to tackle context awareness in aspect definitions. We illustrated our approach with a number of examples that show some interesting properties of this pointcut language. We presented a straightforward implementation and outlined some optimizations that we have incorporated. Nevertheless, efficiency is lacking and there is room for improvement. However, we think the language is expressive enough to pursue work in this area. In fact, we are currently experimenting with a new implementation based on the Rete algorithm, and future work will concentrate on exploring the feasibility of efficient implementations.

References

[1] Eric Bodden. J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen university, 11 2005.

[2] Christoph Brzoska. Temporal logic programming with bounded universal modality goals. In *ICLP'93: Proceedings of the tenth international conference on logic programming on Logic programming*, pages 239–256, Cambridge, MA, USA, 1993. MIT Press.

[3] Christoph Brzoska. Temporal logic programming with metric and past operators. In *IJCAI '93: Proceedings of the Workshop on Executable Modal and Temporal Logics*, pages 21–39, London, UK, 1995. Springer-Verlag.

[4] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. *Lec-*

ture Notes in Computer Science, 2192:170–184, 2001.

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

[6] Gregor Kiczales. The fun has just begun. Keynote at AOSD2003.

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European conference on Object-Oriented Programming*. Springer-Verlag, jun 1997.

[8] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[9] Mehmet Ali Orgun and Wanli Ma. An overview of temporal and modal logic programming. In D M Gabbay and H J Ohlbach, editors, *Proceedings of ICTL'94: The 1st International Conference on Temporal Logic*, pages 445–479, Berlin Heidelberg, 1994. Springer-Verlag.

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

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