A Semantic-based Runtime Weaver for Dynamic Management of the Join Point History

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

January 25, 2007

1 Introduction

Although early research in aspect-oriented programming focussed on aspects that are triggered at a single join point, more recent research has evolved towards aspects that are triggered based on the occurrence of a series of join points in the execution of a program. These types of aspects were dubbed *event-based aspects*, *stateful aspects* [2] and *context-aware aspects* [10], and a number of novel pointcut languages with direct support for these kinds of aspects are currently being developed [9, 5, 1].

One of the most challenging aspects of developing an aspect language that supports these *history-based aspects*, is managing the join point history. In an ideal situation, we would keep all data about join points in memory forever, so that we could write arbitrary pointcuts over this history. However, in reality, this is not feasible. Currently a series of static analysis techniques (e.g. AspectJ [8], Alpha [7]) have been proposed where one analyzes the base code and aspect code to derive *join point shadows*, which are places in the base code that generate a join point that could possibly trigger a pointcut. Henceforth an optimal weaver can be build that omits generating unnecessary join points and hence the recorded join point history is reduced. In this paper we take a look at how the join point history can be managed at runtime so that data about join points can be deleted once it is no longer relevant for resolving pointcuts. More concretely, we discuss the weaver of the HALO language.

The logic-based language HALO provides support for writing history-based aspects that are defined in terms of temporal relations between join points. For this purpose, HALO offers a predefined set of higher-order temporal predicates – derived from temporal logic programming – for connecting pointcuts. The latter is exploited by the HALO weaver: because of the predefined set of temporal relations the weaver can – in some cases – with certainty decide whether data about a specific join point will ever (again) be used in matching a pointcut. Hence we can effectively reduce the join point history as the program runs.

2 The HALO language

HALO is an extension of Common Lisp, allowing one to express history-based aspects over a CLOS program. In addition HALO (similarly to CARMA [4] and Alpha [9]) is based on logic programming and as such pointcuts are expressed as logic queries over the join point history. The built-in pointcut predicates in HALO capture the key events in the execution of a CLOS program. For this discussion, explaining the pointcut predicate for capturing generic function calls suffices. `(gf-call ?gfName ?arguments)` captures generic function call join points and exposes the generic function's name and argument list through the logic variables `?gfName` and `?arguments`. In addition, pointcuts can be composed from other pointcuts by means of the higher-order temporal predicates. In this discussion we consider the temporal predicates: `most-recent`, `all-past` and `since`. For example, the pointcut below matches at a generic function call named `checkout` and also captures the most

recent generic function call named `buy` along with the most recent call to `checkout` before the latter.

```
(at ((gf-call 'checkout ?user1)
      (most-recent (gf-call 'checkout ?user2)
           (most-recent (gf-call 'buy ?user2 ?article2))))
      (format t "~s just bought ~s" ?user2 ?article2))
```

3 HALO weaver

The bulk of the HALO weaver consists of a query engine that matches logic facts generated for each join point against pointcuts; The latter query engine is based on the Rete forward chaining algorithm [3]. Put briefly, logic queries (or pointcuts in HALO) are represented as a network of nodes in Rete. Each such node has a *memory table* that is used to cache partial matches of the query, which are computed by propagating facts through the network. In standard Rete the two main types of nodes are filter nodes and join nodes. Filter nodes store logic facts, whereas join nodes cache conjunctions of the latter. We have extended the Rete forward chaining algorithm with novel types of join nodes to implement the different temporal predicates in HALO. In addition we extended the Rete algorithm to incorporate removing old conclusions when propagating inserts through these nodes.

As an example the Rete network for the pointcut discussed above is depicted in figure 1. A sample program run is depicted in the same figure. In addition, the figure displays tables labelled `LT` (life time): the intervals stored by these tables indicate the begin and end point for the interval during which entries in the memory tables are kept. Note that though the entries in the third filter node are removed as new entries are made, the derived conclusions are not also removed at the same time: at time 7 for example, when the entry made for `(gf-call 'buy <lotte> <dvd>)` is removed, the derived conclusion for time 5 in the first `most-recent` join node is kept. This ensures that at time 8 it can be used to match the pointcut. But this does not mean the derived conclusion is kept forever. The first `most-recent` join node is itself the input of another `most-recent` join node. The input nodes of this second join node share no variables. So the entry for time 5 in the output memory table of the first join node is removed when any other entry is made, which in this example will happen the next time a user checks out if he bought something (e.g. if the user `lotte` does another checkout).

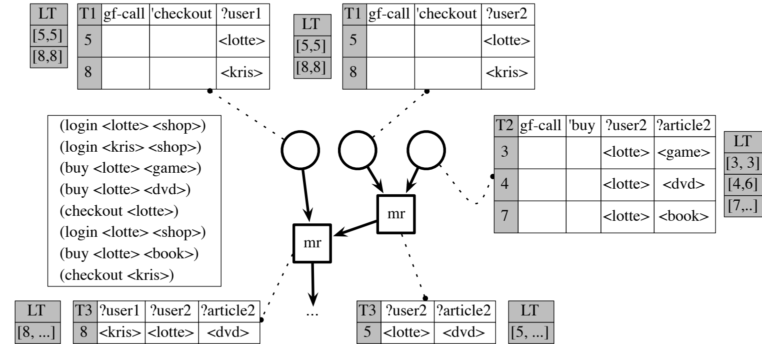

Figure 1: Garbage collection of the join point history.

4 Presentation Outline

The goal of this presentation is to discuss the possible benefits of enabling a dynamic management of the join point history, and to contrast our approach with the static analysis techniques used to avoid generating join points. For this purpose, we are currently benchmarking a web application we extended with two new features using HALO [6]. The results from this experiment will then be presented and used to validate our approach.

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotk, 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] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Con-*