A Unified Model of Aspect-Instantiation Policies [Extended Abstract]

Andre Loker
University of Twente
the Netherlands
a.loker@student.utwente.nl

Steven te Brinke
University of Twente
the Netherlands
s.tebrinke@utwente.nl

Christoph Bockisch
University of Twente
the Netherlands
c.m.bockisch@cs.utwente.nl

Categories and Subject Descriptors

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

Keywords

instantiation policy, aspect-oriented programming, unified model

1. INTRODUCTION

One important concept in aspect-oriented execution environments is implicit invocation, typically supported through the pointcut-advice approach [13]. In this approach, functionality is given by an *advice* which is implicitly executed at execution points determined by *pointcuts*. In general, advices preserve state across multiple invocations and even multiple advices may share state. Aspect-oriented languages prevalently are extensions to class-based object-oriented languages, in which *aspects* extend the class concept and can be instantiated to objects at runtime. Aspect instances are used to store the state required by advice jointly defined in the same aspect, and advices are executed—similarly to usual methods—in the context of an aspect instance.

Because advices are called implicitly, such aspect-oriented languages support the specification of so-called *instantiation policies* to define how to retrieve the aspect instance for the implicit invocation of advice. Dufour et al. [3] have found that aspect-instance look-up took more than 26% of the total execution time in selected use cases. Thus, it is important to research such policies and optimization opportunities. To this end, we suggest a unified model that conceptualizes instantiation policies to simplify and optimize the implementation of current and future instantiation policies in aspect-oriented execution environments.

2. DESIGN SPACE

We have investigated the instantiation policies of a wide range of existing aspect-oriented languages which follow very different approaches. For instance, in AspectJ with its statically deployed aspects, instantiation policies can only refer to statically expressible context. In CaesarJ and JAsCo, aspects can be deployed dynamically and, thus, policies can be defined that refer to runtime values. In the Compose* language, aspects are not instantiated; data fields declare their scope individually. We also studied work on the design space of aspect-oriented (e.g., [9]) and other languages with advanced modularization mechanisms [1].

A very popular instantiation strategy is to create only one *singleton* aspect instance which is used for every advice invocation. *Per-object* instantiation policies (such as `perthis` and `pertarget` in AspectJ) use a different aspect instance for each distinct value of a specific role (e.g., the caller or callee at a join point) in the execution context. Sakurai et al. [11] present the concept of *Association Aspects* where the user explicitly associates a specific aspect instance with a unique combination of n values.

AspectJ also supports the `percflow` and `percflowbelow` strategies to share aspect instances while a specified join point is active on the call stack. In JAsCo or CaesarJ, the instantiation policy can be per thread. JAsCo also supports the policies `permethod`, `perclass` and custom instantiation policies. The first two associate one aspect instance with each distinct method (or class declaring the method) accessed at a join point. Custom policies receive a reification of the current join point, but can create and share aspect instances freely.

3. UNIFIED MODEL

Policies have significant semantic differences: For Association Aspects, aspect instances must be created explicitly by the application program, but AspectJ aspects are implicitly instantiated by the runtime environment. Also, Association Aspects support the use of wildcards in the specification of an instantiation policy. A unified model for instantiation policies must be able to support all these configurations.

All instantiation policies we encountered in our study establish a relation between n-tuples of context values (potentially logic values as in the case of, e.g., `percflow`, or $n = 0$ as for the singleton policy) and aspect instances. An instantiation policy can be described by the rule how these n-tuples are built from the execution context (the `bind` function) and whether or not aspects may be instantiated implicitly (the `implicit` flag).

If the binding function creates a key for which no aspect instance exists *and* the implicit flag forbids implicit instantiation, no aspect instance can be retrieved. In such a case,

kind of query:	exact	full-range	partial-range
array	$O(n)$	$O(n)$	$O(n)$
hash map	$O(1)$	$O(n)$	
binary search tree	$O(\log n)$	$O(n)$	$O(\log n)$
trie			$O(k)$

Table 1: Computational complexity of queries

the execution environment typically decides to not execute the advice. In our study, we did not encounter alternative behavior such as raising an error when no applicable aspect instance can be retrieved.

Our model defines aspect instance retrieval for an aspect instantiation policy by the `implicit` flag together with the functions `bind`, `find`, and `store`.

To retrieve aspect instances, first `bind` creates a *query-key tuple* consisting of values from the execution context and wildcards. `Find` then uses this query key to look up associated aspect instances. If no matching instances can be found, the `implicit` flag is true, and the query-key tuple does not contain wildcards, a new aspect instance is created. In such a case, `store` is called to remember the association between the query key and the aspect instance.

The function `find` takes a query-key tuple k and returns all existing aspect instances with a matching key tuple. The tuples match if they are point-wise equal at all non-wildcard positions. All positions with a wildcard $*$ in the query-key tuple are ignored. The function `store` remembers a new association between a key tuple (i.e., a tuple without wildcards) and an aspect instance.

Example: The `pertarget` policy defines `bind` to return a query-key tuple containing the target of the current method invocation. However, some instantiation policies do not depend on a direct runtime value, such as per-thread and per-cflow. In languages with a managed execution environment, such as Java, a first-class representation of the current thread can be retrieved from the runtime. For the current control flow (cflow), e.g., a shadow stack can be maintained [12] to create such a representation of the current control flow. These first-class representations are then used in the query-key tuple. ▪

4. GENERIC STORAGE FUNCTION

The implementation of the storage function is independent of the instantiation policy's semantics. Nevertheless, different data structures can be employed which have different performance characteristics. We differentiate three uses of `find`: In *exact queries* the key tuple does not contain wildcards; the tuple acts as a key in a dictionary. In *full-range queries* the tuple contains only wildcards and all existing aspect instances are returned. In *partial-range queries* the tuple contains both wildcards and non-wildcard values.

Table 1 lists the asymptotic computational complexity of search algorithms for different data structures depending on the kind of query performed. A *trie* [4]—or *prefix tree*—is specifically suitable for partial-range queries; for other queries it has the same complexity as other trees. If wildcards only appear in the last query-key-tuple positions of a partial range query, the complexity of a partial-range query depends on the number, k, of non-wildcard values in the tuple.

Binary search trees have a comparatively low complexity, but they require a natural order of key tuples. Since key tuples consist of arbitrary application objects, such ordering may be difficult to establish. Besides the complexity of the find operation, the complexity of the store operation also has to be considered to choose the most appropriate data structure. Also, for small numbers of key-tuple associations, the time complexity of implementations can be very different from the theoretical complexities, thus a direction for future research is to perform actual benchmarks to choose the optimal implementation.

5. RELATED WORK

Lorenz and Trakhtenberg present a pluggable framework for Aspect Instantiation Models (AIM) in AspectJ [6]. Their approach resembles our model in that it allows instantiation policies to be treated as separate modules. The Event Composition Model (ECM) [8, 7] defines an interface which can be implemented to configure the instantiation of so-called *event modules*. Thus, instantiation policies can also be flexibly customized. In contrast to our *declarative* model which concentrates on abstracting the behavior of instantiation policies, both above approaches focus on providing the infrastructure to make *imperative* definitions of instantiation policies replaceable. Different policies cannot be represented uniformly and generic optimizations are not possible.

Sakurai et al. [11] present an optimization for association aspects associating exactly two objects, which is transparently applied in suitable situations. Haupt and Mezini [5] propose an extension to the virtual machine's object layout to optimize aspect-instance lookup for `pertarget` and `perthis` uniformly. Based on a less complete and less formal description of the unified model, generic low-level optimizations have been implemented in the just-in-time compiler of the Steamloom$^{\text{ALIA}}$ virtual machine [2]. These optimizations are applicable to all instantiation policies with specific characteristics, such as having explicit instantiation and exact queries.

6. DISCUSSION

We have based our study mainly on aspect-oriented languages that extend object-oriented languages and that need to retrieve an object to act as receiver of an advice invocation. Nevertheless, our results are not limited to finding a single receiver. Our prototype is implemented in the ALIA4J [1] framework for execution environments for aspect-oriented languages, based on the Java language. In ALIA4J advice are Java methods. and values passed to these methods as arguments can all be retrieved through the unified instantiation model presented in this work. Using it for the receiver of a method call is just one special case for our prototype. Thus, together with the unified instantiation model presented here, ALIA4J can act as a workbench for advanced modularization mechanisms that go beyond the traditional object-oriented encapsulation of state, such as the *Sea of Fragments* proposed by Ossher [10].

7. REFERENCES

[1] C. Bockisch, A. Sewe, H. Yin, M. Mezini, and M. Aksit. An in-depth look at ALIA4J. *Journal of Object Technology*, 11(1):1–28, Apr. 2012.

[2] C. Bockisch, A. Sewe, and M. Zandberg. ALIA4J's (just-in-time) compile-time MOP for advanced

dispatching. In ACM, editor, *Proc. compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11 & VMIL'11*, SPLASH '11 Workshops, pages 309–316, New York, NY, USA, Okt 2011. ACM Press.

[3] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of AspectJ programs. *SIGPLAN Not.*, 39(10):150–169, 2004.

[4] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, Sept. 1960.

[5] M. Haupt and M. Mezini. Virtual machine support for aspects with advice instance tables. *L'OBJET*, 11(3):9–30, 2005.

[6] D. Lorenz and V. Trakhtenberg. Pluggable aspect instantiation models. In S. Apel and E. Jackson, editors, *Software Composition*, volume 6708 of *LNCS*, pages 84–99. Springer, 2011.

[7] S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. PhD thesis, University of Twente, Enschede, Sept. 2011.

[8] S. Malakuti, M. Akşit, and C. M. Bockisch. In *Ninth IEEE Int. Sympos. Parallel and Distrib. Process. with Appl. Workshops, ISPAW 2011, Busan, Korea*, pages 328–335, USA, May 2011. IEEE.

[9] I. Nagy. *On the design of aspect-oriented composition models for software evolution*. PhD thesis, Enschede, June 2006.

[10] H. Ossher. A direction for research on virtual machine support for concern composition. In *Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, VMIL '07, New York, NY, USA, 2007. ACM.

[11] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD '04: Proc. 3rd Int. Conf. Aspect-oriented software development*, pages 16–25, New York, NY, USA, 2004. ACM.

[12] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Schoeberl, and M. Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the java virtual machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 11–20, New York, NY, USA, 2011. ACM.

[13] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.