A Case for Explicit Join Point Models for Aspect-Oriented Intermediate Languages

Hridesh Rajan
Iowa State University
hridesh@iastate.edu

Abstract

Aspect-oriented languages mostly employ implicit language-defined join point models, where *well-defined* points in the program are called join points and declarative predicates are used to quantify them. The primary motivation for using an implicit join point model is obliviousness and ease of quantification. A design choice for aspect-oriented intermediate languages is to mirror the source language model. In this position paper, I argue that an explicit join point model is better suited at the intermediate language level and sketch a preliminary solution.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features - Control structures; D.3.4 [*Programming Languages*]: Processors-runtime environments

General Terms Languages

Keywords Aspect-oriented intermediate languages, explicit join point models, implicit join point models, Nu AO intermediate language

1. Introduction

Aspect-oriented (AO) languages based on static compilation models such as AspectJ [17], Eos [23], etc have shown the potential to provide significant modularity benefits. Support for aspect-orientation in virtual machines and intermediate languages open new avenues. A key benefit is to preserve separation of concerns beyond compilation [21], which in turn promises to simplify development processes such as incremental compilation, debugging, etc. More optimization opportunities also open up as shown by Bockisch *et al* in their recent work [4].

The design of AO intermediate languages has also received some attention recently. Abstractions provided by enhanced intermediate language designs promise to preserve the separation of concerns achieved by AO source languages at the object code level. For example, one such language design discussed in our previous work [21, 22] on *Nu* extends existing intermediate languages to include two new AO invocation primitives. We demonstrated that the *Nu* AO intermediate language allowed design modularity to maintained even at the object code level. All common advising structures in prevalent aspect-oriented (AO) mechanisms [9, 15] could be modeled as simple combinations of these two invocation primitives.

The purpose of this position paper is to direct attention at another important aspect of the intermediate language design: the join point model. A point in the execution of a program is called a join point in the popular terminology of aspect-oriented languages. A method for selecting a subset of these join points is called quantification. The notion of quantifiable join points [10, 12] is central to the notion of aspect-orientation [9, 16]. The AspectJ programming guide [3], for example, defines a join point as a new concept and explains that it is a *well-defined* point in the execution of the program. Others often define a join point as an implicit *language-defined* point in the execution of the program. I argue that an explicit join point model is more suitable for an AO intermediate language design, instead of an implicit language-defined model.

2. Rationale for a Language-Defined Implicit Join Point Model

The primary rationale for a language-defined join point model is *obliviousness* [10, 11]. Obliviousness is a widely accepted tenet for aspect-oriented software development. In an oblivious AOSD process, the designers and developers of base code need not be aware of, anticipate or design code to be advised by aspects. This criterion, although attractive, has been questioned by others for various reasons and there is at least some consensus among researchers that complete obliviousness between base and aspect designers and developers may be a mirage [2, 5, 6, 8, 13, 18, 25]. Tools such as AspectJ Development Tools (AJDT) alleviate the problem [1] but do not completely solve it. Nevertheless, the notion of obliviousness appears to have significant influence on the design of aspect-oriented languages. A language-defined implicit join point model promotes obliviousness in that it allows aspect developers to quantify join points without requiring the base code developers to declare them.

The implicit language-defined join point model wins hands down with respect to the ease of the first time implementation. It is definitely much easier compared to manual join point selection (e.g. by placing annotation on join points) for the programmer to select join points for advising. By just writing simple declarative expression, they can select join points throughout the code base. In the next section, I discuss the design rationale for employing an explicit join point model in the intermediate language design.

3. Rationale for Explicit Join Point Models

In this section, I discuss the rationale for an explicit join point model in AO intermediate languages. In particular, I describe two important goals: extensibility, and the need for a mechanism to make reflective information available at a join point.

Workshop VMIL '07 March 12-13, 2007 Vancouver, British Columbia, Canada.
Copyright © 2007 ACM 1-59593-661-5/07/03. . . $5.00.

3.1 Extensibility

Virtual machines and intermediate language designs are often subject to standardization, which makes it extremely hard to change them. On the other hand, language designs often evolve to incorporate experimental ideas and constructs. For example, new join points are proposed to address use cases that the traditional join point model was not able to address [14, 24]. Other use cases are also documented, where desirable join points were not quantifiable in the current models: for example, in the context of the AO design of the Hypercast system, Sullivan *et al* [25] observed that *many join points that have to be advised in the same way cannot be captured by a quantified PCD, e.g., using wild-card notations. A separate PCD is required for each join point. There were about 180 places in the base code where logging was required. Most of the join points do not follow a common pattern. Not only is there a lack of meaningful naming conventions across the set of join points, but also variation in syntax: method calls, field setting, etc.* [25, pp. 170] These use cases will serve to fuel the evolution of the join point model at the source language level.

Adopting an implicit language model at the intermediate language level will restrict the design space of aspect-oriented source languages. Either the language designer will have to wait for the intermediate language designs to evolve to support new join points or they will emulate them using existing models thereby sacrificing the benefits of deeper support at the virtual machine and intermediate language level. Therefore, a key goal for an intermediate language design is extensibility. An explicit join point model, where join points in the intermediate code are precisely marked by the compiler, is likely to be more extensible compared to an implicit model. Such an implicit model will also need a precisely defined semantics and enforcement mechanism to rule out locations in the object code that may not be marked as join points.

3.2 Uniform Reflective Information

Current aspect languages provide an interface for accessing contextual (or reflective) information about a join point. An aspect can access the contextual information at the join point using pointcuts such as *this* to access the executing object (*this*), *target* to access the target object (such as the *target* of a call), *args* to access the arguments at a join point, etc. Alternatively, one can explicitly marshal this information from an *implicit* argument, often called *thisJoinPoint*, available to the advice, where other miscellaneous information such as source code location, name, etc, is also available.

This interface between the join point and the aspects is fixed in current AspectJ-like languages. There are rational reasons for such design decisions. This interface introduces coupling between the classes and the aspects. The thinner this interface is the lower the coupling will be, resulting in perhaps easier and independent evolution of classes and aspects. Extending the set of language constructs to include access to more primitives also takes away regularity from the language design [20]. As it is, current language constructs for retrieving contextual information are not completely regular, e.g. this, target, and arguments are not available at all join points [3].

In addition, others have shown that this rather limited interface does not satisfy all usage scenarios. For example, in some cases access to a local variable is needed [25, pp. 170] in others access to other information such as join point specific messages for logging is needed at the join points. Therefore, the source language design may evolve to include additional reflective information. It is therefore imperative that a more flexible method to access contextual information at the join point is provided at the intermediate language level that can support these evolutions.

```
ExplicitJP
   : .joinpoint modifier type
     identifier([arguments])block
block
   : {[instruction_list]}
```

Figure 1. Abstract Syntax of Explicit Join Points

```
class Point: FigureElement {
 ...
 public void SetX(int x) {
      if(this.x != x){
           this.x = x;
        }
     }
 }
```

Figure 2. An Example Code Snippet

```
1  .method public hidebysig instance
2      void SetX(int32 x) cil managed
3  {
4    // Code size       17 (0x11)
5    .maxstack  2
6    .joinpoint public void ExecutionSetX(int32 x)
7     {
8     IL_0000:  ldarg.0
9     IL_0001:  ldfld      int32 Point::x
10    IL_0006:  ldarg.1
11    IL_0007:  beq.s      IL_0010
12    IL_0009:  ldarg.0
13    IL_000a:  ldarg.1
14    IL_000b:  stfld      int32 Point::x
15    IL_0010:  ret
16    } // end of join point execution(public void Point.SetX(int x))
17 } // end of method Point::SetX
```

Figure 3. An Explicitly Declared Execution Join Point

4. An Explicit Join Point Model for Intermediate Languages

The proposed intermediate language design has two key characteristics that serve to satisfy the goals set in the previous section. First, it explicitly labels sections in the intermediate code that correspond to the join point shadows, and second, it explicitly defines the types of reflective information exposed at the join point. The view is similar to that of Ligatti *et al* [19] and Clifton and Leavens [7] in their semantics but has not appeared in language designs. Figure 1 shows the abstract syntax.

These labels will be generated by the compilers. To model language-defined implicit join points, the compiler would generate appropriate labels at all necessary locations (join point shadows) defined by the language semantics. For example, to model an `execution` join point shadow in AspectJ-like languages, the matched method code will be labeled as shown in Figure 3. The figure shows the intermediate code in Common Intermediate Language (CIL) for the source code shown in Figure 2. Here the intermediate code for the method `SetX` of class `Point` is labeled as the join point `ExecutionSetX` on line 6. Based on the reflective information being used in the advice, join point only exposes the value of the argument. It may also choose to expose reflective information as in AspectJ-like languages. The scope of the join point is identified by the block that encompasses instructions from line 8 to line 15.

Note that these labels are not visible to the programmer; therefore the source language-level obliviousness is still maintained. Moreover, explicit join point shadow makes the AO intermediate

```
.method public hidebysig instance
     void SetX(int32 x) cil managed
{
  // Code size       17 (0x11)
  .maxstack  2
   IL_0000:  ldarg.0
   IL_0001:  ldfld      int32 Point::x
   IL_0006:  ldarg.1
   IL_0007:  beq.s      IL_0010
  .joinpoint public void IfBlockInsideSetX(int32 x)
   {
     IL_0009:  ldarg.0
     IL_000a:  ldarg.1
     IL_000b:  stfld      int32 Point::x
   } // end of join point IfBlockInsideSetX
   IL_0010:  ret
} // end of method Point::SetX
```

Figure 4. Supporting Finer-grained Join Points

language extensible in that it may now support source languages with different join point models.

To demonstrate the extensibility of this AO intermediate language model, let us now consider an evolutionary scenario, where the join point model of the source language is enhanced to include conditional constructs (if, switch) as join points. Using this enhanced model, the aspect developer chooses to select the execution of the true block (line 5 in Figure 2) of the *if* statement inside the method `SetX`. This statement truly represents the state change of the Point class. The compiler for this enhanced language model may now generate the intermediate code as shown in Figure 4. In this modified version, only the intermediate code corresponding to line 5 in Figure 2 is within the scope of the new join point `IfBlockInsideSetX`.

5. Conclusion

In this position paper, I argued that explicitly declared join points are better suited for intermediate languages to support extensibility in source languages in two dimensions. First key dimension is evolution of join point models of source languages. Second dimension is extension of the reflective information that is available at the join point. A preliminary solution was proposed with the expectation that it will serve to generate exciting discussion during the workshop.

Acknowledgements

The author is supported in part by National Science Foundation grant CT-ISG 0627354 and by a generous startup grant provided by Iowa State University. This paper draws upon the discussions with my colleague Gary T. Leavens and my students Robert Dyer and Juri Memmert. Many thanks to the anonymous reviewers of the AOSD VMIL 2007 reviewers for helpful comments.

References

[1] AJDT:AspectJ development tools. http://www.eclipse.org/ajdt/.

[2] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *Proc. 2005 European Conf. Object-Oriented Programming (ECOOP 05)*, pages 144–168, July 2005.

[3] AspectJ programming guide. http://www.eclipse.org/aspectj/.

[4] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient control flow quantification. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, 2006.

[5] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report TR03-01a, Iowa State University, January 2003.

[6] Curtis Clifton and Gary T. Leavens. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-23, Department of Computer Science, Iowa State University, December 2005.

[7] Curtis Clifton and Gary T. Leavens. MiniMAO$_1$: Investigating the semantics of proceed. *Science of Computer Programming*, 2006.

[8] C. Constantinides and T. Skotiniotis. Reasoning about a classification of cross-cutting concerns in object-oriented systems. In Pascal Costanza, Günter Kniesel, Katharina Mehner, Elke Pulvermüller, and Andreas Speck, editors, *Second Workshop on Aspect-Oriented Software Development of the German Information Society*. Institut für Informatik III, Universität Bonn, February 2002. Technical report IAI-TR-2002-1.

[9] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[10] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October 2000.

[11] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-oriented Software Development*, pages 21–35. Addison-Wesley Professional, 2004.

[12] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. pages 21–35. Addison-Wesley, Boston, 2005.

[13] William G. Griswold, Kevin J. Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software, Special Issue on Aspect-Oriented Programming*, Jan/Feb 2006.

[14] Bruno Harbulot and John R. Gurd. A join point for loops in aspectj. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2006. ACM Press.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Budapest, Hungary, June 2001.

[16] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, pages 481–90, Boston, Massachusetts, 17–23 May 1997. IEEE.

[17] 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 (ECOOP)*, Finland, June 1997. Springer-Verlag.

[18] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[19] Jay Ligatti, David Walker, and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, Winter 2005/2006.

[20] Bruce J. MacLennan. *Principles of programming languages: design, evaluation, and implementation (2nd ed.)*. Holt, Rinehart & Winston, Austin, TX, USA, 1986.

[21] Hridesh Rajan, Robert Dyer, Youssef Hanna, and Harish Narayanappa. Preserving separation of concerns through compilation. In Lodewijk Bergmans, Johan Brichau, and Erik Ernst, editors, *In Software Engineering Properties of Languages and Aspect Technologies (SPLAT 06), A workshop affiliated with AOSD 2006*, March 2006.

[22] Hridesh Rajan, Robert Dyer, Harish Narayanappa, and Youssef Hanna. Nu: Towards an aspect-oriented invocation mechanism. Technical Report 414, Iowa State University, Department of Computer Science, Mar 2006.

[23] Hridesh Rajan and Kevin J. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, September 2003. ACM Press.

[24] Hridesh Rajan and Kevin J. Sullivan. Aspect language features for concern coverage profiling. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 181–191, New York, NY, USA, 2005. ACM Press.

[25] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *The Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 166–175, Sept 2005.