Aspect-Oriented Programming and Component Weaving: Using XML Representations of Abstract Syntax Trees

Stefan Schonger[1], Elke Pulvermüller[2], and Stefan Sarstedt[1]

[1]Fakultät für Informatik, Universität Ulm, D-89069 Ulm, Germany

[2]Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe, D-76128 Karlsruhe, Germany

{schonger, pulvermueller, sarstedt}@acm.org

Abstract

Aspect-Oriented Programming (AOP) and related techniques propose solutions to the problem of crosscutting requirements, usually by providing a *weaver* that reimplements major parts of a compiler.

This paper proposes XML based "operators" as an extensible aspect language. We work on XML representations of abstract syntax trees (AST) for the base language. These can be generated by modifying an existing compiler and allow us to use XML tools for tree query and manipulation. A prototype that encompasses constructs from several aspect languages, in particular AspectJ and Composition Filters, has been implemented.

1 Introduction

The need for modularization and separation of concerns has been the topic of several by now classical publications (like [1]). Although not necessarily straightforward, this decomposition along the axis of functional requirements is well understood and has been subject of extensive research.

On the other hand, requirements that have complex semantics and a crosscutting impact on software—such as distribution, persistence, or robustness—still present well-documented difficulties for software development [2]. Various techniques have been proposed to manage such concerns: *Aspect-Oriented Programming* [2] (AOP) and related techniques like *Composition Filters* [3] (CF) and *Subject Oriented Programming* [4] (SOP). Increasingly, and furthermore in this paper, the term *aspect-oriented software development* (AOSD) is used to subsume all these concepts.

For the three approaches mentioned above, implementations exist: *AspectJ* is a system for AOP and version 1.0 was released at the time of this writing [2]. *Hyper/J* is a prototype for MDSOC [4] and *ComposeJ* is a prototype implementing a limited subset of CF [5]. All three prototypes use Java as the base language. Since source code is only available for AspectJ and ComposeJ and their approach is better documented, we will mostly refer to these for comparison.

Although AOP seems to become useful in solving real-world problems, the situation is not perfect from a research point of view as well as for some practical applications: there is no strong theoretical basis yet, the existing prototypes are always bound to a particular base language and the user usually has no support to extend the language. Although there is some research on AOSD semantics [6, 7, 8, 9] the semantics descriptions provided by the available AOSD tools are neither formal nor give enough hints at the implementation strategy. Finally, the examined implementations are large.

The main goal of the operator approach is to provide a platform for experimentation in AOP techniques with more flexibility in terms of the aspect language and the base language and to heavily reuse existing technologies like compiler front-ends and XML tools to make the implementation significantly shorter and the transformation parts succinct and readable. In the context of this work, a prototype that implements the new approach has been built and it has been shown how examples from the AspectJ and ComposeJ literature can be realized using this technique. In the following we present our XML-based operator approach.

AspectJ version:

```
aspect PointBoundsChecking {

   static final int MIN_X = 0;
   static final int MAX_X = 100;

   ...

   void checkX(int x) {
     check("New x value illegal.",
          (MIN_X < x) && (x < MAX_X));
   }

   void check(String description, boolean test) {
      if ( !test ) throw new Error(description);
   }


   // AspectJ part that does the binding
   // corresponding to the operator on the right

   before(int x):
     execution(void Point.setX(int)) &&
     args(x) {
       checkX(x);
   }
}
```

Java part:

```
class PointBoundsChecking extends Aspect {

   static final int MIN_X = 0;
   static final int MAX_X = 100;

   ...

   void checkX(int x) {
      check("New x value illegal.",
           (MIN_X < x) && (x < MAX_X));
   }

   void check(String description, boolean test) {
      if ( !test ) throw new Error(description);
   }
}
```

Operator part:

```
<transform>
  <operator name="method-execution" modifier="before">
    <join-point xpath="//class[@name="Point"]" \
                       "//method[@name="setX"]"/>
    <call-method aspect="PointBoundsChecking"
                 name="checkX"/>
    <parameter-forward>
      <forward type="argument" number="1">
    </parameter-forward>
  </operator>
</transform>
```

Figure 1: An example for preconditions. AspectJ code on the left, the operator approach on the right.

2 The Operator Approach

We demonstrate the principles of the approach with an example. Figure 1 shows code for precondition checking in AspectJ and using the proposed operator approach side by side. The AspectJ version is a slightly modified version of an example from [2].

The code that performs the actual precondition checking at the top of Figure 1 is identical for both approaches with one exception: for the operator approach it is Java code separate from the binding instructions whereas in AspectJ it is part of the "aspect". The Java class containing this part of the aspect is generated by the weaver. Apart from this separation of code and advice in the operator approach, the binding instructions at the bottom of Figure 1 use a different syntax: In AspectJ the syntax is close to Java and is seen as a language extension, whereas for the operator approach the syntax is defined using XML. This XML code does not necessarily have to be directly written by the user, it can be generated by an IDE offering a user-friendly interface. The actual code produced by both versions is almost identical.

2.1 Ingredients for an AO language

To motivate the structure of the operators used, we briefly take a look at existing AO languages and contrast them with our approach.

Although no agreement on the requirements for an AO language has yet been reached, according to Kiczales et al. [2], there is a distinction between *static* and *dynamic crosscutting*. For the latter, three language elements are deemed necessary: a *join point model*, a means of identifying join points and a means of specifying the additional behavior at these join points.

AspectJ's dynamic join point model characterizes join points as "certain well-defined points in the execution flow of the program" [2]. Join points are identified and combined using pointcut designators such as `execution` for method executions, `call` for method calls or `this` for "all join points at which the executing object (value of `this`) is an instance of `Point` or a subclass of `Point`" [10, Section 3.4].

The *advice* construct associates code in the *advice body* with a particular pointcut and executes it `before`, `after` or instead of (called "`around`") the

```
public class Test {
   String toString( int i ) {
   return ""+i;
   }
   public static void main( String argv[] ) {
     System.out.println(
       new Test().toString( 42 ) );
   }
}

aspect Negation {
   before(): !execution(* *.toString( * )) {
     System.out.print(".");
   }
}
```

Figure 2: Code causing `ajc`[1] to generate a non-terminating recursion for the `toString` method

original code.

We argue that not all pointcut keywords are equal, but rather fall into classes. Firstly, there are *primitive join points* like `call` and `execution` that cannot reasonably be combined with other pointcut designators using "and" or "not". An example to illustrate this point is a single negated primitive join point like `!execution(* *.toString(*))`. The semantics definition of AspectJ [11] does not explicitly forbid this construct, but also does not precisely define its meaning. Without the negation, all method executions of methods with the name `toString`, irrespective of the return type, arguments, and class membership are advised. With the negation, probably all join points except the `toString` method execution should be affected. Not only is the semantics definition unclear on this, but the current version of `ajc`[1] causes a non-terminating recursion on entry to the `toString` method—clearly not what was intended (complete example in Figure 2). The only reasonable combination—"or", or set union—can be replaced by two separate advice declarations. Thus we argue that each pointcut should contain *at most one* primitive join point and as we will later see actually *exactly one*. In the operator approach, this is the name of the operator. Using a primitive join point thus specifies the type of pointcuts currently considered. As such, a primitive join point can be seen as the superset of the join points that should be selected by providing a pattern as a parameter.

Secondly, there are *syntactical constraints* expressed by the keyword `within` or the signature of the selected method calls. They narrow the set of join points selected by the primitive join point. In the operator approach, syntactical constraints are specified using XPath[2] [12] expressions over the AST nodes.

[1] Version 1.0

```
public class Test // same as in Figure 2

aspect Constraint {
   before(): this(Test){
     System.out.print(".");
   }
}
```

Figure 3: Aspect with a syntactical constraint only

Thirdly, there are *dynamic constraints* that make the execution of the advice body dependant on the state of the program. They do not alter the set of join points but generally require a check at run-time. Examples for AspectJ constructs falling in this category are `this`, `cflow` or `if`. In the first prototype of the operator approach, these are not modelled, but they could be added at a later stage. AspectJ resolves these dynamic constraints as program instructions statically woven into the base code. Therefore, the current lack of dynamic constraints is no principal limitation of our approach.

The AspectJ semantics definition does not explicitly specify that the constructs just categorized as constraints can never stand alone. Again, `ajc` causes counterintuitive results: An example using only a constraint can be seen in Figure 3. Without being quite clear why, the code translated with `ajc`[1] causes four invokations of the aspect code when `main` is executed. When the constraint `this(Test)` in the example is exchanged with `within(Test)`, the aspect code is executed ten times. We thus argue that constraints should not stand alone and each pointcut should contain *at least one* primitive join point, resulting in *exactly one* primitive join point with the above result.

For AspectJ's static crosscutting, the `introduction` operator is used. It takes a location and what should be introduced as parameters.

Constructs from other AO languages seem to fit as well. For example, a `forward` operator that introduces unconditional forwarding methods to selected inner objects has been implemented (cf. "Nutshell Classes" [13]). Forwarding and delegation are used for object composition.

Composition filters [3] allow messages sent and received by objects to be intercepted and manipulated. The ComposeJ prototype [5] implements the input fil-

[2] XPath seems expressive enough, but upcoming XML query languages like XQuery could also be used once they are standardized

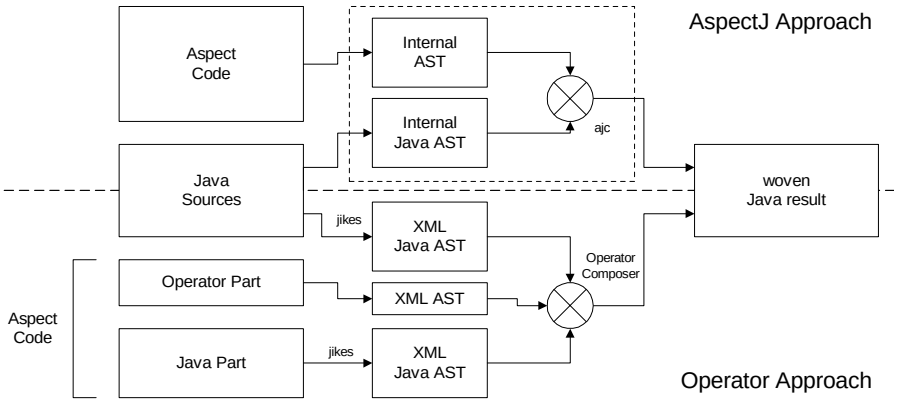

Figure 4: AspectJ vs. the Operator Approach

ters `Error` and `Dispatch`. Both input filters that were modelled in the ComposeJ prototype [5] have successfully been implemented in our approach and the "`USVMail`" example provided with ComposeJ works as intended. As our focus was to show the general feasibility to integrate other approaches, our prototype at this point shares the limitations of the ComposeJ prototype. Again this is no principal limitation of our approach.

2.2 Operator structure

The operator structure will be briefly presented by revisiting the example in Figure 1.

All operators are enclosed by the `transform` tag and in general ordering is important. The type of operator is given as the `name` parameter and for this example specifies the primitive join point type. Syntactic restrictions are specified in the `join-point` tag using XPath. The following tags describe which aspect method to call and which parameters should be passed to the aspect method.

The general operator structure can be described using XML Schema [12], which allows for automatic schema validation.

2.3 Implementation

All examined prototypes have to parse regular Java code and perform semantic analysis at least to a certain point. They all use ASTs internally to represent and manipulate Java programs. Since open-source implementations of Java compilers—such as `jikes`—exist that do semantic analysis and build an AST as part of the compiler front-end, this is an obvious source for reuse. There even has been work [14] where an annotated AST in XML format is generated and externalized during a run of a slightly modified version of `jikes` (about 1650 lines of code were added [14, p. 7]). This XML representation can directly be parsed using an XML API that exists for most languages including Java. Our prototype has been realized using Badros's JavaML work [14].

Figure 4 shows the translation process using the operator approach in comparison to AspectJ. The input in both cases consists of Java source code and aspect code. The Java input is identical and the aspect constructs are specified in a different format as detailed above. Parsing, semantic analysis, transformation and output to Java source code or compilation to class files are all implemented internally in AspectJ. The operator approach takes an annotated AST from the modified `jikes` compiler, using it for parsing and semantic analysis. The prototype then transforms the input as specified using XML tools to operate on the tree. The output from the tree is either performed by using XSLT stylesheets to generate Java source code or alternatively the XML interface to `jikes` could be made bi-directional and the back-end directly used for compilation of the transformed AST.

Basing the transformation process on abstract syntax trees has the benefit of being independent of the concrete syntax. Additionally, the implementations of existing operators are easier to adapt to a different, but similar base language. Although other existing prototypes work on ASTs, they seem to have a tighter coupling between the language dependant parsing and semantic analysis part and later stages. Separating the

base language part from the operator part and using an independent syntax for the latter further helps to prevent coupling. Some other prototypes like ComposeJ also keep the base language part separate and decoupling on a similar level has been proposed in [15]. Admittedly, a syntactically close language extension like AspectJ has some appeal for developers used to the base language and keeping advice and binding instruction together is intuitive. Flexibility is lost, however, and we think that IDE support will get more important, for example to automatically list all advice at join points in the source. This IDE support could also show the operator next to both the corresponding Java part and the points where it crosscuts.

XML is used to model the AST as well as for tree query and manipulation. This helps to keep the implementation of the prototype short and concise. Extensible stylesheet language transformations (XSLT) have been used to unparse the AST as proposed in [14], with features, such as syntax-highlighting or hyperlinks from variable references to the corresponding variable definitions, being possible. XPath is a tree query language and was successfully used to select primitive join points and model static restrictions. Using a standardized query language also helps to realize performance gains from newer implementations.

We took some measurements to compare the code sizes of the various approaches using the non commentary source statements (NCSS) metric[3]. AspectJ[4] has about 34.6 kNCSS in the `org.aspectj.compiler` package, the ComposeJ prototype has 16.8 kNCSS and our prototype currently totals 1.4 kNCSS including a graphical user interface. Even with adding the 1.7 kLOC of JavaML and a rough estimate of a maximum of about 0.5 kLOC of XSLT code, our prototype is still an order of magnitude smaller. However, we have not researched if and how much of the other approaches' code is generated.

3 Related Work

There have been a few papers suggesting to use transformational techniques—like the ones used in the context of compiler optimizations—for AOP implementations [16, 17]. However, in contrast to its early days, the focus of AOSD has developed from lower-level to design level code manipulation. This has actually made implementations easier since the join points must be specified explicitly and transformations are just applied once. So although the transformations used by newer AOP approaches are different and avoid the problems of confluence and termination, they still use source-to-source transformations.

The fact that data structures used internally by compilers such as ASTs are more useful than source code as input for many tools is widely acknowledged. JavaML [14] is a tool that externalizes an AST in XML format and has been used as a basis for our work.

The concept of an operator as a construct for composition has been used in various contexts, such as for mixin-based inheritance [18] or design patterns [19].

Criticism of the AspectJ pointcut constructs has been issued in [20], where the authors come to similar conclusions. However, their classification is slightly different and they do not present concrete examples.

Furthermore, it has been suggested [21] to use compact representations of abstract syntax trees as intermediate languages for mobile code. In the context of this work, this would enable the techniques used here to transform source code at compile time to work on the intermediate representation at class-load time.

Intentional Programming (IP) is another AOSD approach that is implemented using AST transformations [22]. Its focus is on domain specific languages and not on transformations for crosscutting concerns. Ten years after its inception in 1991 neither a prototype was released to the general public nor is the source code available and the project's future at Microsoft seems unclear.

4 Further work

The next logical steps will be to study how the approach extends to further AOP techniques like Hyper/J, how easily the base language can be changed and to gather empirical evidence on how well the different AOP approaches can be used in combination.

Refactorings strive to improve system architectures by applying semantics preserving transformations. Having access to the annotated AST, it should be relatively straightforward to implement some of them as operators.

It has been suggested to view the introduction of design patterns as code transformations [19]. In fact, examples for both AspectJ and composition filters show the aid of AOSD for the implementation of certain

[3] The NCSS metric measures code size similar to the lines of code (LOC) metric but is independent of the coding style. We used the JavaNCSS tool for our measurements

[4] Version 1.0

design patterns. Thus it may be worthwhile to investigate the suitability of higher level operators for pattern introduction.

In general, more research in the area of operators is planned. Besides the extension with additional AO and composition operators, other categories of operators will be explored and integrated.

5 Conclusion

Empirical studies [23] suggest that aspect-oriented programming is a useful new programming technique. Most existing AOSD languages and prototypes seem to suffer from several problems: there is little guidance on how to combine different approaches, the AO languages are hard to extend, the base language is fixed, and the prototype sources are relatively large.

This work contributes by proposing an experimental platform, the operator approach, that shows steps towards possible solutions for these problems. The issues we found with language design in AspectJ underline this need for experimentation. We base our transformations on abstract syntax and work on AST nodes expressed in XML and generated in a separate step. This make the migration to other base languages possible. Basing the aspect language on XML makes it easy to extend the language and to combine different approaches. The implementation of our individual operators is relatively short and succinct since we can use various standardized XML libraries for tree query and manipulation. Reusing a compiler front-end through JavaML [14] is feasible and further helps towards a lean implementation.

The work with a prototype for this approach, so far, has been encouraging, but the results must still be considered preliminary.

References

[1] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, pages 1052–1058, December 1972.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Grisworld. Getting started with AspectJ. *Communications of the ACM*, October 2001. Also see http://www.aspectj.org.

[3] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters. *Communications of the ACM*, October 2001.

[4] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using hyperspaces. Technical Report RC 21452(96717)16APR99, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1999.

[5] J. C. Wichman. ComposeJ: The development of a preprocessor to facilitate composition filters in the java language. Master's thesis, Department of Computer Science, University of Twente, 1999.

[6] R. Lämmel. A Semantical Approach to Method-call Interception. In *Proc. 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*. ACM Press, April 2002. To appear.

[7] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming, January 2002. To appear in FOOL 9.

[8] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Reflection*, LNCS 2192, pages 187–206, September 2001.

[9] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Reflection*, LNCS 2192, pages 170–186, September 2001.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Grisworld. An overview of AspectJ. In *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pages 327–353, Heidelberg, Germany, 2001. Springer-Verlag.

[11] G. Kiczales et al. The AspectJ programming guide, AspectJ version 1.0, November 2001. Also see http://aspectj.org/doc/dist/progguide/.

[12] W3C XML standards. See http://www.w3.org/XML.

[13] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998.

[14] G. J. Badros. JavaML: A markup language for java source code. In *WWW9 – Ninth International World Wide Web Conference*, volume 33, pages 159–177, 2000. Also see http://www.cs.washington.edu/homes/gjb/.

[15] A. Speck, E. Pulvermüller, and M. Mezini. Reusability of concerns. In *Proceedings of the Aspects and Dimensions of Concerns Workshop, ECOOP2000*, Sophia Antipolis and Cannes, France, June 2000.

[16] U. Aßmann and A. Ludwig. Aspect Weaving by Graph Rewriting. In *Generative Component-based Software Engineering (GCSE)*, Erfurt, October 1999.

[17] P. Fradet and M. Südholt. AOP: towards a generic framework for aspect-oriented programming. In *Third AOP Workshop, ECOOP'98 Workshop Reader*, volume 1543, pages 394–397, 1998.

[18] G. Bracha and W. Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of Fourth ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 303–311, 1990.

[19] T. Gensslers and B. Schulz. Design patterns as operators implemented with refactorings. In *Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, 1998.

[20] B. De Win, B. Vanhaute, and B. De Decker. Towards an open weaving process. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, August 2001.

[21] M. Franz. *Mobile Object Systems*, chapter Adaptive Compression of Syntax Trees and Iterative Dynamic Code Optimization. LNCS 1222, pages 263–276, February 1997.

[22] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. Transformation in intentional programming. pages 114–123. IEEE Computer Society Press, 1998.

[23] M. A. Kersten and G. C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. OOPSLA, 1999.