A Proposal for Extensible AspectJ *

Vladimir Oliveira Di Iorio
Universidade Federal de Viçosa - Brazil
vladimir@dpi.ufv.br

Leonardo Vieira dos Santos Reis
Roberto da Silva Bigonha
Mariza Andrade da Silva Bigonha
Universidade Federal de Minas Gerais - Brazil
{leo,bigonha,mariza}@dcc.ufmg.br

ABSTRACT

This article presents the preliminary results achieved while working with a language to define extensions to the concrete syntax of AspectJ. The language uses the concept of *syntax classes*, units that extend classes with syntax definitions, building modular specifications for extensions. A syntax class can define a new construct with cross-cutting features either by translating it into pure AspectJ code or by modifying the behaviour of elements in different parts of the program, acting like an aspect weaver. The definition of new pointcut designators is also possible, with clear separation between run-time and weave-time processing. The language can be used as a tool to create domain-specific extensions to AspectJ, and domain-specific aspect languages embedded into AspectJ.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Extensible languages

General Terms

Languages

1. INTRODUCTION

Domain-specific aspect languages (DSALs) are programming languages with aspect-oriented features specially designed for a particular domain. Aspect-oriented languages must cope with cross-cutting behaviour, meaning that constructs may affect several distinct parts of a program. So the implementation of DSALs frequently requires more sophisticated resources than the ones required by languages without aspect orientation.

Syntax extension is a technique that can be used to embed a domain-specific language within an existing language. Some tools for syntax extension of programming languages work as macro expanders, with new defined constructs replaced by code written on the original (base) language [11, 7]. This technique may also be applied to aspect-oriented languages, when the cross-cutting resources of the base language are powerful enough to express the semantics of the new proposed constructs [10, 8].

This article presents *XAJ*, a language to extend the concrete syntax of AspectJ, with more powerful resources than the ones provided by simple macro expanders. The main concept of XAJ is *syntax classes*, units that encapsulate the specification of extensions, adding syntax definitions to AspectJ classes.

Some of the main features of XAJ are presented by examples, in the next sections. Section 2 explains how the basic mechanism of *syntax classes* works, and shows an example where a single instance of a new proposed construct is translated into several AspectJ elements. Section 3 describes an example that the cross-cutting resources of AspectJ are not powerful enough to define the semantics of the proposed new construct. The solution using XAJ shows how the language can execute typical tasks of an aspect weaver, changing the behaviour of elements located in different parts of the program. Several proposals of extensions for AspectJ deal with extensions of the pointcut language definition. Section 4 describes the XAJ approach for this problem, giving a definition of a new pointcut designator with a clear separation of run-time and weave-time processing.

Although the examples discussed in the following sections do not always represent real use situations or complete languages, they give a clear perspective of how XAJ can be used to easily design and implement extensions for AspectJ.

2. MULTI-INTRODUCTIONS

An extension for AspectJ proposed in [6] eliminates redundancy related to inter-type declarations in cases where the declarations must be inserted in several classes, but following a simple pattern. An example is the *accept* method for the *Visitor* pattern, which must be inserted in all classes on a given class hierarchy. Chiba suggests the following syntax for *multi-introductions*:

```
void Base+.accept(Visitor v) { v.visit(this); }
```

The use of *Base+* on the declaration means that it must be inserted on all subclasses of *Base*.

Syntax classes are a XAJ resource which extends classes with syntax definitions. New language constructs may be defined with the *@Grammar* declaration, with an associated semantic action that builds a node of an abstract syntax tree

*Work supported by *Fapemig* (APQ-00205-08).

DSAL'09, March 3, 2009, Charlottesville, Virginia, USA.

```
public class MultiIntro extends Sugar {
  private TypeD returnType;
  private String className, methodName;
  private Formals params;
  private BlockStm block;
  public MultiIntro(TypeD t, String c,
      String n, Formals p, BlockStm b) {
    returnType = t; className = c;
    methodName = n; params = p; block = b; }
  @Grammar extends IntroducedDec {
    MultiIntro ::=
      t = TypeD
      c = String
      "+." n = String
      "(" p = Formals ")"
      b = BlockStmt
    { return new MultiIntro(t, c, n, p, b); }
  public AST desugar(Context ctx) {
    ClassDec cd = ctx.getClass(className);
    ClassDec sub[] = cd.getSubClasses();
    AspectMembers list = new AspectMembers();
    for(ClassDec x :  sub)
      list.addChild('[#returnType #x.getName()
        .#methodName (#formals) #block ]);
    return list;
  }
}
```

Figure 1: Definition of multi-introductions.

(AST). Figure 1 shows a syntax class for *multi-introduction*, defined as an extension of the nonterminal symbol *IntroducedDec* of an AspectJ grammar. The *@Grammar* declaration adds a new nonterminal symbol *MultiIntro* and two new production rules to the grammar:

```
IntroducedDec -> MultiIntro
MultiIntro -> TypeD String "+" "." String
    "(" Formals ")" BlockStm
```

The production rule uses other existing symbols of the AspectJ grammar, such as *TypeD* (return type for methods), *Formals* (list of parameters) and *BlockStm* (block of statements). In the AspectJ grammar adopted, the symbol *IntroducedDec* is used for the definition of declarations inside an aspect, such as pointcuts and advices. So the new construct is treated as a new kind of declaration which may appear inside an aspect.

Besides defining concrete syntax, syntax classes may also be used for the representation of new constructs as nodes of an AST. In Figure 1, the nonterminal symbols of the new production rule are bound to variables t, c, n, p and b and the constructor of the class is executed, building an AST node for the representation of the new *MultiIntro* construct. Note that the type of the attributes of the class, which also represent nodes of an AST, have the same name as the nonterminal symbols of the AspectJ grammar.

A XAJ compiler translates syntax classes into an extended AspectJ compiler. When applied to programs using the extended syntax, the generated compiler first builds an AST that may include nodes defined by the user. Then a traversal on the AST is performed, and the *desugar* method is executed on the new nodes. This method must return pure AspectJ code, which will replace the processed node on the

```
public class GlobalPointcut extends Sugar {
  static List<GlobalPointcut> globalPcs; ...
  public AST desugar(Context ctx) {
    globalPcs.add(this); return null;
  }
}
```

Figure 2: Compilation of global pointcuts.

AST. Semantic errors can be detected and reported. Figure 1 shows an example of *desugar* for the *MultiIntro* syntax class. The *Context* parameter allows access to context information from the point where the new construct is used on the program, such as local variable declarations, or even the entire AST. The Java code on the method body is executed **during compilation time**, when the traversal of the AST is performed. It calculates the set of subclasses of the given type using operations on the AST. At this point, it is not possible to use reflection on the classes of the system, since no real code has been generated yet. Then the method returns a list of inter-type declarations that will replace the multi-introduction node. To build this list, generative programming is used, using syntax based on Meta-AspectJ [8]. The generated code must be something that may be produced by the *IntroducedDec* symbol of the AspectJ grammar, otherwise a syntax error occurs.

3. GLOBAL POINTCUTS

The example presented in this section shows two features of XAJ: the use of more than one traversal pass on the AST and the redefinition of existing AspectJ elements.

In [1], an extension for AspectJ is proposed, called *global pointcut*, defining a common conjunct shared by many pieces of advice. The general form is:

global : <*ClassPatern*> : <*Pointcut*>;

It has the effect of replacing the pointcut of each advice declaration with the conjunction of the original pointcut and the global *Pointcut*, for the aspects whose name matches *ClassPattern*.

The compilation of global pointcuts requires two passes. The first pass registers the information about the global pointcuts, which may appear anywhere inside aspect definitions. The second pass introduces the conjunctions on all advices matching the given pattern, for each global pointcut.

Figure 2 shows the compilation of global pointcut declarations. The *desugar* method is executed, by default, on the first traversal of the AST. Each global pointcut declaration is registered on a static list and then a null value is returned, meaning that the global declaration will not be present on the generated code.

Figure 3 shows the compilation of advices. The method *definePasses* is overridden, defining that *desugar* is called only on the second traversal of the AST. The keyword *overrides* on the *@Grammar* declaration indicates that this class does not define a new AST node, but changes the behaviour of AST nodes representing advices. The complete code of *desugar* is not shown in Figure 3, but it must implement the following: each registered global pointcut is analyzed, and a conjunction is added to the pointcut of the currently visited advice, if the name of the aspect which contains this advice matches the registered pattern.

```
public class AdviceGP extends Sugar { ...
  public void definePasses() {
    includePass(2); } ...
  @Grammar overrides Advice; ...
  public AST desugar(Context ctx) {
    for (GlobalPointcut gp :
      GlobalPointcut.getGlobalPcs()) { ...  }
  }
}
```

Figure 3: Overriding compilation of advices.

```
public AST onWeaving(Context ctx) {
  Type t = ctx.typeOf(expr);
  Type t1 = ctx.typeOf(typeName);
  if (t.subTypeOf(t1)) return '[if (true)];
  if (!  t1.subTypeOf(t)) return '[if (false)];
  return '[if (#expr instanceof #typeName)];
}
```

Figure 4: Weaving-time for a pointcut designator.

The XAJ approach for traversal passes of the AST makes it easy also to define multiple traversals for a single class. More than one pass may be included by *definePasses*, and in this case an alternative version of *desugar* may be used, with an additional parameter *pass* that indicates the current traversal pass:

```
public AST desugar(Context ctx, int pass) {
  if (pass == 1) { ...
  else if (pass == 2) { ...
```

This method could be used, for example, to report a warning for global pointcuts with no matches, in a third traversal pass.

It is important to note that XAJ offers no support for dealing with problems caused by the interaction of more than one extension definition. The programmer is totally responsible to define the operations performed in each traversal of the AST.

4. POINTCUT DESIGNATORS

In [6], a mechanism is proposed in order to enrich pointcut languages with user-defined pointcut designators, constructs that play the role of "new primitive pointcuts". In this section, we show that XAJ may do something similar.

Suppose a pointcut designator *isType* that tests the type of a given expression, used as follows:

```
... && args(x) && IsType(x,T) ...
```

where T is a valid type. Such a test may be implemented in AspectJ with conditional check pointcuts, but only at run-time. XAJ proposes a clear separation of run-time and weave-time processing.

Suppose that a XAJ syntax class defining *IsType* extends the AspectJ definition of pointcuts, with attributes *expr* and *typeName* representing the parameters of the proposed pointcut designator. Figure 4 shows the code of a method *onWeaving* that is called during weave-time. Since the definition of pointcut is extended, this method is called for every join point visited by the weaver.

The Java code on the method body is executed during weave-time. If it is possible to decide the result using only static information, the method returns a pointcut that is always *true* or *false* – in this case, no run-time test is added to the generated code. If it is **not** possible to decide the result using only static information, a run-time test is generated.

The resources for syntax definition even allow the notation for a new pointcut designator to be different from the pattern "*identifier(parameterList)*". It is possible to generate completely different pieces of code for each join point visited during weave-time, depending on the current context, but the generated code must be a valid pointcut. It is also possible to access directly static information on joinpoints, dispensing the use of variables like x, on the given example.

5. RELATED WORK

The idea and format of *syntax-classes* of XAJ, extending Java classes with syntax definitions, are borrowed from XJ [7], a proposal for a language extension to Java. In both XAJ and XJ, extensions are modularized by being attached to classes, and code is generated in the form of AST instances, represented by expressions using a *quasiquote* mechanism. An important difference is that, in XJ, every new construct must be prefixed with the "@" character, having only a local effect, and the semantics of existing constructs may not be altered. Another obvious difference from XAJ to XJ and other tools like OpenJava [11] is that those systems generate only Java code, with no cross-cutting resources.

Maya [3] is another system that generates Java code, using a pattern matching mechanism to define macro expansion rules on Java ASTs. Unlike XJ, the definitions have a global effect on Java programs. This feature produces extensions with lower modularization than XJ, but is well suited for the implementation of language extensions such as aspect weaving. Syntax-classes of XAJ define modularized extensions, and also allow the generation of code that affects several parts of a program, either by generating AspectJ code (as in Section 2) or by overriding the semantics of existing constructs (as in Section 3).

XAspects [10] defines a plug-in mechanism for developing domain-specific aspect languages. The only point for extension of the concrete syntax is the definition of aspects. The *aspect* keyword may be followed by parentheses containing a valid type name for a class, which describes the plugin itself. All the tokens contained in the {} brackets are scanned and interpreted by the plugin. So, unlike XAJ, XAspects is not tailored to the definition of general extensions for the concrete syntax of AspectJ, and the modification of existing code is limited by the use of AspectJ advices and intertype declarations.

In [8], a methodology for language extension using MetaAspectJ is presented. It is another approach that uses AspectJ as the underlying "assembly" language, with the constructs of AspectJ as the only resource for the modification of existing code. Extensions are defined using annotations. No mechanism for the extension of the concrete syntax of the language is discussed.

Josh [6] allows users to define new pointcut designators, with separation of run-time and weave-time processing. If a run-time check is necessary, it must be explicitly inserted using a bytecode manipulation framework. With XAJ, it is only necessary to generate code following the same mechanism used for all extensions. XAJ also allows pointcut designators using a new notation defined by the user.

A common technique for the implementation of DSLs consists of writing a library in an existing programming language, designing a notation for the DSL and then programming a compiler that translates from the notation into equivalent library calls. DSALs makes this task harder, because crosscutting resources are more difficult to be encapsulated in libraries. In [2], the *Syntax Definition Formalism* (*SDF*) [12] is used for defining language syntax (notation). The authors propose the use of Stratego/XT [4] as the program transformation tool for the definition of libraries with crosscutting features.

The *AspectBench Compiler* (*abc)* is an implementation of AspectJ designed to support easy extensibility. In [1], the authors show how several extensions for AspectJ may be implemented using this workbench. The example presented in Section 3 is one of these extensions.

6. CONCLUSION

This article has presented the preliminary results achieved while working with a language called XAJ, a proposal for extensible AspectJ. The main contributions are: an approach that produces modular specifications for AspectJ extensions, encapsulated in syntax classes; the possibility of defining a cross-cutting behaviour for new constructs, either by generating AspectJ code or by overriding the semantics of existing constructs; easy specification of more than one traversal on abstract syntax trees; the possibility of defining new pointcut designators with clear separation of run-time and weave-time processing.

The design of XAJ tries to follow a criterion of proportionality for the development of extensions: smaller extensions should require a smaller amount of work and code. It is evident on the examples shown in sections 2 and 3. If code must be generated only on the first traversal of the AST, just the definition of the simple version of the *desugar* method is necessary. If the extension requires a single traversal of the AST, different from the first one, the *definePasses* method must be overridden. If two or more traversals are required, they must be defined on the *definePasses* method and the extended version of the *desugar* method must be used.

The plans for the complete development of XAJ are as follows: (1) design a first version for the specification of the language; (2) collect significant examples of extensions for the AspectJ language; (3) describe how the selected extensions may be implemented in XAJ; (4) build a compiler for the language; (5) test the extensions with the compiler. This work has presented results from stages (1), (2) and (3) above. Now we are evaluating tools for the implementation of the compiler for the language.

The implementation of XAJ syntax classes requires tools that allow the definition of formal languages with easy extensibility. Polyglot [9] is a good option, with the specification of AspectJ given on the abc workbench [1]. Another option is the *Syntax Definition Formalism* (*SDF*) [12], with the definition of AspectJ presented in [5].

The code generation in XAJ uses a quasiquote mechanism, as shown in figures 1 and 4. One option for this mechanism is Meta-AspectJ (MAJ) [8], a mature meta-programming tool with resources that imply syntactically correct generated programs.

The example presented in Section 4 shows how new pointcut designators can be defined in XAJ. The proposed mechanism produces calls to the *onWeaving* method for every jointpoint visited by the weaver, when the definition of pointcut is extended by a syntax class. In the future, XAJ may propose notations for other extensions that must interact with the AspectJ weaver, such as: the definition of new kinds of joinpoints, and new pointcuts that match these joinpoints; the definition of new pointcuts that produce variable bindings, such as the AspectJ *args* pointcut; the definition and use of new dynamic and static information for weave-time processing. Examples of such extensions are implemented in the abc workbench and are well documented in [1]. XAJ may evolve to become a DSAL with a notation specially designed for the extensibility mechanisms of abc.

7. REFERENCES

[1] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible aspectj compiler. In *Proceedings of AOSD '05*, pages 87–98, New York, NY, USA, 2005. ACM.

[2] A. H. Bagge and K. T. Kalleberg. DSAL = library+notation: Program Transformation for Domain-Specific Aspect Languages. In *DSAL'06 Workshop*, 2006.

[3] J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in java. In *Proceedings of PLDI '02*, pages 270–281, New York, NY, USA, 2002. ACM.

[4] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16. Components for transformation systems. In *PEPM'06 Workshop*, Charleston, South Carolina, January 2006. ACM SIGPLAN.

[5] M. Bravenboer, E. Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. A case for scannerless generalized-lr parsing. In W. R. Cook, editor, *Proceedings of OOPSLA'06*, pages 209–228, Portland, Oregon, October 2006. ACM Press.

[6] S. Chiba and K. Nakagawa. Josh: an open aspectj-like language. In *Proceedings of AOSD '04*, pages 102–111, New York, NY, USA, 2004. ACM.

[7] T. Clark, P. Sammut, and J. Willans. Beyond Annotations: A Proposal for Extensible Java (XJ), 2008. (http://www.ceteva.com/docs/XJ.pdf).

[8] S. S. Huang and Y. Smaragdakis. Easy language extension with meta-aspectj. In *Proceedings ICSE '06*, pages 865–868, New York, NY, USA, 2006. ACM.

[9] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[10] M. Shonle, K. Lieberherr, and A. Shah. Xaspects: an extensible system for domain-specific aspect languages. In *OOPSLA '03*, pages 28–37, New York, NY, USA, 2003. ACM.

[11] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. Openjava: A class-based macro system for java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 117–133, London, UK, 2000. Springer-Verlag.

[12] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.