XAJ: An Extensible Aspect-Oriented Language

Leonardo V. S. Reis, Roberto S. Bigonha, Mariza A. S. Bigonha
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
leo@dcc.ufmg.br, bigonha@dcc.ufmg.br, mariza@dcc.ufmg.br

Vladimir O. Di Iorio, Rodolfo C. Ladeira
Departamento de Informática
Universidade Federal de Viçosa
Viçosa, Brazil
vladimir@dpi.ufv.br, rodolfo53821@hotmail.com

***Abstract*—This work presents the language XAJ (*eXtensible AspectJ*), giving a formal syntax definition and implementation details. The main purpose of the language is to increase the modularity and portability of extension definitions for the aspect-oriented language AspectJ. XAJ is itself an extension of AspectJ, introducing the concept of syntax classes, units that extend classes with syntax definitions. Syntax classes also define the semantics of extensions and serve as a representation for nodes of abstract syntax trees. The language can be used as a tool to create domain-specific extensions to AspectJ, and domain-specific aspect-oriented languages embedded into AspectJ.**

***Keywords*-aspect oriented programming; domain specific languages; language extensions;**

I. Introduction

Aspect-oriented languages give the programmer a powerful code transformation tool, but for many applications, programmers need only a subset of aspect languages functionality. *Domain-specific aspect-oriented languages* (*DSALs*) are domain-specific languages (*DSLs*) with aspect-oriented features. They may be a solution for several problems involving cross-cutting concerns, offering enough power to a specific application together with an intuitive and productive syntax. Some of the benefits of using DSALs are higher expressiveness, easiness to use, gain in productivity and reduction of maintenance costs [15].

The same benefits described above can be achieved by using *domain-specific extensions* [4] on existing aspect-oriented languages. So, the research on DSALs is tightly connected to the development of tools and techniques for building extensions for aspect-oriented languages, specially for AspectJ [1], [3], [6], [17].

This paper presents XAJ (*eXtensible AspectJ*), a new language which allows the extension of the concrete syntax of the AspectJ programming language and can be used for the definition of DSALs embedded into AspectJ. *Syntax classes* are the main concept of XAJ. They are units that encapsulate the specification of extensions, adding syntax definition to classes. The AspectJ grammar can be modified at virtually any point, and the semantics of the new proposed elements are defined by manipulation of the abstract syntax tree (AST) inside special methods of syntax classes.

XAJ represents an attempt to increase the modularity and portability of extensions defined for AspectJ. We say that XAJ extensions are modular because the syntactic definition of an extension, the attributes for representing it as an AST node and the translation that defines its semantics are all encapsulated in a single program unit. XAJ is portable because extensions are completely defined with the language itself, not depending on additional development tools. The proposal of the language was first published in [11], showing some possible examples of usage. This paper presents the formal syntax definition of XAJ, discusses informally the semantics of some of its most important features and gives implementation details. Interesting modifications of the original definition are also proposed.

This paper is structured as follows. Section II proposes a new construct for AspectJ, as a motivation for a language extension. In Section III, a formal definition of the syntax of XAJ is presented, and the semantics of some of its most important elements is informally discussed. Section IV gives implementation details of the first compiler for the language. Related work is presented and compared with XAJ in Section V. The conclusions and future work are discussed in Section VI.

II. Motivation

In the *Visitor* design pattern [12], an *accept* method must be inserted in all classes which may be "visited" by a visitor object. Using Java, the code for this method may be:

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

An AspectJ implementation for this design pattern may use intertype declarations to define the *accept* method for the classes to be visited. The hole implementation can be encapsulated in a single code unit (an aspect), but a programmer must still repeatedly write an intertype declaration of *accept* for every class.

Chiba and Nakagawa [8] propose a new construct for AspectJ that could avoid or minimize the code repetition discussed above. If the classes to be visited are exactly the subclasses of class named *Base*, the syntax of the proposed construct could be:

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

The use of *Base+* in the declaration means that the intertype declaration will be inserted on all subclasses of *Base*. This new command saves code repetition and has another important advantage: if the structure of the hierarchy is altered, with new classes being included or excluded, the code needs no modification, provided that *Base* remains the superclass of all classes to be visited.

The extension described above is a simple example of a problem solved with the definition of a new construct for the AspectJ language. It will be referred in the next sections as *multi-introduction*, meaning that it produces multiple intertype declarations (also called *member introductions* in AspectJ).

III. The XAJ Language

The XAJ language is an extension of AspectJ which uses the concept of *syntax classes* to modify its own concrete syntax. Syntax classes encapsulate the specification of AspectJ extensions, adding syntax and semantics definitions to classes, and also serve as a representation for AST nodes. The syntax of language extensions is defined with syntax grammar declarations, allowing the modification of virtually any element of the AspectJ grammar. The semantics can be given by a translation to pure AspectJ code, implemented in a special method named *desugar*. The information for AST representation is automatically generated from the grammar definition.

Figure 1 shows how XAJ extends the AspectJ grammar, using a BNF representation. The non-terminal symbol `class_member_declaration` belongs to the original AspectJ grammar, defining the set of elements that may appear inside a class, such as attribute or method declarations. The first production in Figure 1 adds two new elements to this set. Syntax grammars are defined using the keyword `@Grammar` followed by `extends` (adds new productions to the Aspectj grammar) or `overrides` (changes the semantics of productions). The keyword `@numberOfPasses` defines the number of parser passes required to translate the new construct. A class with a syntax grammar is called a *syntax class*.

We explain some of the elements of a syntax class giving a definition for the *multi-introduction* extension discussed in Section II. The syntax class presented in Figure 2 is defined using the keyword *extends*, meaning that the following new productions are inserted into the original AspectJ grammar:

```
intertype_member_declaraction  ⟶  MultiIntro
MultiIntro  ⟶
    modifiers_opt type
    IDENTIFIER "+" "." IDENTIFIER
    "(" formal_parameter_list_opt ")" block
```

All non-terminal grammar symbols, except `MultiIntro`, are defined on the original AspectJ grammar. The symbol `intertype_member_declaration` is used for the definition of declarations inside an aspect, such as pointcuts and advices.

```
class_member_declaration ⟶
    syntax_grammar | number_passes.
syntax_grammar ⟶
    "@Grammar" "extends"
      non_terminal "{" {production} "}"
    | "@Grammar" "overrides" non_terminal.
production ⟶
    non_terminal "::=" expression ";".
expression ⟶
    term [semantic_action]
      {"|" term [semantic_action] }.
term ⟶
    factor {factor}.
factor ⟶
    | STRING_LITERAL
    [ identifier "=" ] non_terminal
    | [ identifier "=" ] "(" exp ")"
    | [ identifier "=" ] "[" exp "]"
    | [ identifier "=" ] "{" exp "}".
exp ⟶
    term {"|" term }.
non_terminal ⟶
    IDENTIFIER.
semantic_action ⟶
    "{" {statement} "}".
number_passes ⟶
    "@numberOfPasses" "=" INTEGER_LITERAL.
```

Figure 1. Main elements of the XAJ grammar.

So, the new construct is treated as a new kind of declaration which may appear inside an aspect.

A syntax class acts also as a representation for AST nodes. New attributes are automatically declared inside class *MultiIntro*, associated with each nonterminal symbol of the production given in Figure 2. The identifiers used together with nonterminal symbols define the names of the new attributes – for example, *modifiers* and *className*. The types of these attributes are automatically selected from a predefined set of classes that represent AST nodes for the AspectJ grammar. Other automatically generated code includes *get* and *set* methods for the access to the new attributes and also a constructor that creates an AST node representing a *multi-introduction*, called when such construct is found by the parser. If a semantic action is attached to a production, the action must create an AST node representing the new construct, overriding the default constructor call automatically generated by the XAJ compiler. All the automatic code generation explained in this paragraph are improvements proposed by this work to the original definition of XAJ.

Figure 1 shows that a *@Grammar* block may contain more than one production. One of them is the *main* production, initiating with a new nonterminal symbol with the

```
public class MultiIntro {
  @Grammar extends
      intertype_member_declaration {
    MultiIntro ::=
      modifiers = modifiers_opt
      returnType = type
      className = IDENTIFIER
      "+."  methodName = IDENTIFIER
      "("  params =
        formal_parameter_list_opt  ")"
      introducedCode = block ; }

  public AST desugar(NodeFactory nf,
      Context ctx) {
    ClassDecl cd = ctx.getClass(className);
    ClassDecl sub[] = cd.getSubClasses();
    List list =
      nf.TypedList(ClassMember.class);
    for(ClassDecl x : sub)
      list.add(
        nf.IntertypeMethodDecl(modifiers,
        returnType, x, methodName,
        params, introducedCode));
    return list; }
}
```

Figure 2. Definition of multi-introductions.

same name of the syntax class (*MultiIntro*, in Figure 2). *Auxiliary* productions may be used to define a more sophisticated syntax, allowing also recursiveness. Only the main nonterminal symbol is visible outside of the syntax class. Internal classes are automatically generated to represent AST nodes associated with the auxiliary productions, and these internal classes are not visible outside the syntax class where they are defined.

The semantics of extensions is given with the special method *desugar*. This method is executed at compilation time, after the parser builds an AST for the input program. It must build and return a new AST node containing only pure AspectJ code to replace the extension node in the original AST. Figure 2 shows *desugar* for multi-introductions, slightly simplified. The code searches for all classes which are subclasses of *className*. For every subclass, a new intertype declaration is inserted into a list of class members. Finally, this list is returned. This means that a *MultiIntro* node is replaced by a list of intertype declarations, written with pure AspectJ.

In the original XAJ proposal [11], the code to replace an extension node is produced using generative programming with a quasiquote notation, a resource that hides most AST implementation details. In the current version of the compiler, this feature is not implemented yet, so the code in Figure 2 uses a factory and explicit references to the XAJ predefined classes for AST representation.

IV. Implementation

We have used the *AspectBench Compiler* (*abc*) [1] to implement the first version of a compiler for XAJ, which we call *xajc*. The main reason for choosing abc was that it gives a modular and efficient implementation of AspectJ. The front-end of abc is built using Polyglot [16], a highly extensible compiler front-end for Java, so syntax extensions for abc are defined using the same principles used by Polyglot. In fact, abc is itself a Polyglot extension. The XAJ language is also a Polyglot extension, adding the productions presented in Figure 1 to the AspectJ grammar defined by abc.

The first step of the xajc compiler is the extraction of the extension information from the syntax classes, followed by the compilation of these classes, before the rest of the source code. Then, the parse table is extended with the new constructs, the remaining code is parsed using the new parse table and an AST is built, including nodes which represent user-defined extensions. This AST is translated into pure AspectJ by the code inserted in *desugar* methods, before the final bytecode generation. Figure 3 presents a simplified scheme of the compilation process adopted by xajc, which is explained bellow.

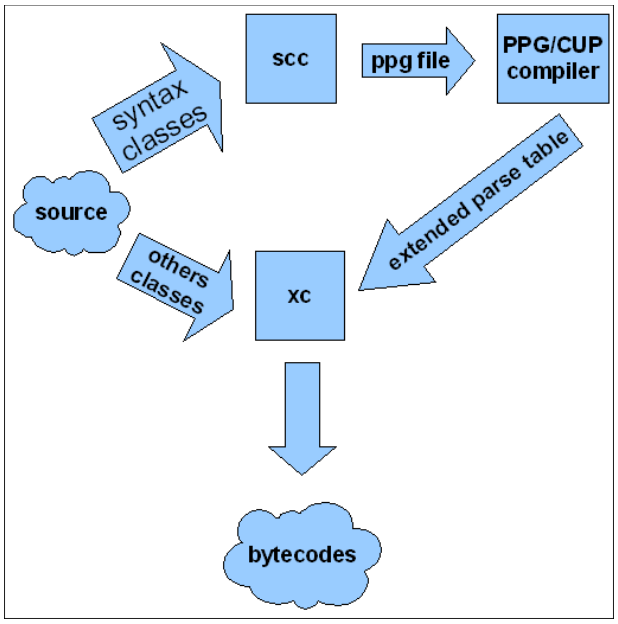

Figure 3. Compilation scheme.

The first step of the xajc compiler is performed by a program which we call *scc* (*Syntax Class Compiler*). This program creates a *ppg* file [7], based on grammar productions extracted from syntax classes. This file is used by Polyglot to extend grammars. The PPG compiler (from Polyglot) is then executed over the ppg file, generating a new parse table including syntax information of the user-defined extensions. PPG uses LALR parsing, so extensions may generate conflicts when they are combined with the XAJ grammar. The programmer is supposed to solve these conflicts modifying the new productions of the language. Our plans for future versions of xajc include features for automatically minimize conflicts.

An extended compiler is built combining the new parse table generated by the PPG compiler and the code from *desugar* methods of syntax classes. The code inserted on *desugar* methods may not contain extended syntax or AspectJ elements, it must be pure Java code.

The source code (excluding syntax classes) is parsed using the generated extended compiler. An AST is built including nodes which represent user-defined extensions. The *desugar* methods are executed over extension nodes and the resulting AST represents only pure AspectJ code. The final translation to bytecode is executed by the abc compiler.

The Polyglot architecture is based on an extensible set of passes. The first pass executes parsing on the input program and builds an AST. The input of all subsequent passes is this AST, that must be processed and possibly modified. The execution of most passes over the AST is based on an alternative version of the *Visitor* pattern [12], calling specific methods on the extension nodes. Polyglot allows users to define new passes that may be executed in any desired order. We have defined a new pass as a subclass of the abc pass for type analysis. The new pass replaces the original one, calling *desugar* methods on extension nodes, before proceeding to type analysis. This approach allows the translation of extension nodes to pure AspectJ using also information about the context where constructs are inserted in the input program.

V. Related Work

The goal of XAJ is to serve as a tool for defining AspectJ extensions and DSALs embedded into AspectJ, with modularity and portability as important issues. In this section, we compare XAJ with similar tools, analyzing the resources for language extensibility, generation of code with cross-cutting features, modularity and portability of extensions.

The idea of XAJ *syntax classes* is borrowed from XJ [9], a proposal for extensible Java. From the original XAJ proposal, important differences between the two approaches are: in XJ, every new construct must be prefixed with the "@" character, having only a local effect; the semantics of the existing Java constructs may not be modified; extensions in XJ must be translated to pure Java code, with no cross-cutting resources. Syntactically, XAJ syntax classes have become less similar to XJ syntax classes with the improvements on automatic code generation described in Section III.

Josh [8] is an AspectJ-like language with an extensible pointcut language and a few mechanisms for generic description. It does not have any resources for syntax extensibility, but it allows the definition of new pointcut designators and a generic and reusable description of advices. The current version of xajc does not offer resources for weave-time manipulation like Josh, but the original XAJ proposition allows the definition of new pointcut designators with clear separation between run-time and weave-time processing. Future versions of xajc will implement these resources.

A methodology for language extension using Meta-AspectJ is presented in [13]. Similarly to XAJ, it is an approach that uses AspectJ as the underlying "assembly" language, translating user-defined constructs to pure AspectJ code. But it is another example of tool that lacks mechanisms for generic syntax extensions, which are defined using only annotations.

XAspects [17] defines a plug-in mechanism for developing DSALs embedded into AspectJ. The only point for syntax extension is the definition of aspects, a very restrictive approach for language extensibility. XAspects addresses modularity issues, but it does not consider portability, since extensions are completely dependent on plugin implementations.

Maya [3] is similar to XAJ in a sense that extensions are completely defined with the language itself, improving portability. An advantage of XAJ is that extensions may be translated into AspectJ, while Maya supports only Java. The use of AspectJ may allow easier translation for extensions with cross-cutting features.

In [2], DSALs are considered as syntactic abstractions over transformation libraries, analogous to the way DSLs are syntactic abstractions over base libraries in the subject language. Stratego/XT [5] is chosen as the program transformation tool which implements the cross-cutting features. This approach does not address portability, since it is completely dependent on a specific tool.

The *AspectBench Compiler* (*abc*) [1] is a highly flexible implementation of an AspectJ compiler, providing resources for building extensions in several dimensions. XAJ has used abc for its own implementation, adding a higher level of abstraction to the definition of extensions. Advantages of using XAJ, when compared to using directly abc, may be exemplified by the definition of *global pointcuts*, proposed in the abc documentation. A XAJ solution for this problem was first outlined in [11] and the implementation with the current version of xajc is completely encapsulated in a syntax class and it is independent of specific development tools.

A formal definition of the AspectJ syntax, implemented by a scannerless generalized LR parser, is presented in [6]. This technique allows the combination of language extensions in a much powerful way than LALR parsing, since it supports the full class of context-free grammars. We have used abc on the implementation of XAJ, although it uses LALR parsing, because it is a full AspectJ compiler and our plans involve also weave-time analysis. Better results on extension combinations could be achieved by integrating a generalized LR parser with the front-end of abc.

VI. Conclusions and Future Work

This paper presents a formal syntax definition of the XAJ language, explains the semantics of some of its most important elements and describes a first implementation. AspectJ extensions defined with XAJ are encapsulated in units called *syntax classes*. Syntax classes are an attempt to improve the modularity of extension definitions, encapsulating syntax, semantics and the representation of AST nodes. Portability is also an important goal: as extensions are totally defined inside the language itself, they do not depend on specific tools or implementations.

During the implementation of the xajc compiler using abc, a significant problem was detected, related to the efficiency of the compilation process. The parse table must be modified in compile time by xajc, generating a new parser each time a XAJ program is compiled. Polyglot and other important parser builders are not well prepared to deal with this requirement, since generating a parse table is not a frequent task in most systems. When defining a language extension, the modifications on the original parser table are punctual, so techniques that keep most of the table unaltered and generate only additional information for the extensions would be more efficient. Works like [14], although presented long ago, offer interesting solutions for this new requirement. The use of a parse table with an efficient implementation of the dynamic behaviour described above is part of our future plans for XAJ.

The original specification of XAJ allows the definition of new pointcut designators with clear separation between runtime and weave-time processing. The current version of the xajc compiler does not implement this feature yet. Other important XAJ proposed features not implemented yet by the current version of the compiler are the use of generative programming with quasiquote notation for building AST nodes and a comprehensive library to manipulate AST information. Both features may help writing the code for *desugar* methods. All these features will be addressed in future versions of xajc.

A domain-specific aspect-oriented language derived from AspectJ or embedded into this language may be implemented by defining a set of AspectJ extensions. We believe that XAJ is an ideal tool for this task. For example, *AJSynchro* [10], an application-specific DSAL for synchronization, has been completely implemented using the current version of xajc. Our plans include also testing the compiler and its future versions on the implementation of several other significant DSALs and comparing the XAJ implementation with the original ones.

Acknowledgment

The authors would like to thank the brazilian research support agencies Fapemig and CNPq.

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 *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2005, pp. 87–98.

[2] A. H. Bagge and K. T. Kalleberg, “Dsal = library+notation: Program transformation for domain-specific aspect languages,” in *Proceedings of the Domain-Specific Aspect Languages Workshop*, October 2006.

[3] J. Baker and W. C. Hsieh, “Maya: multiple-dispatch syntax extension in java,” in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. New York, NY, USA: ACM, 2002, pp. 270–281.

[4] D. Batory, B. Lofaso, and Y. Smaragdakis, “JTS: Tools for Implementing Domain-Specific Languages,” in *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*. Washington, DC, USA: IEEE Computer Society, 1998, p. 143.

[5] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, “Stratego/XT 0.16. Components for transformation systems,” in *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)*. Charleston, South Carolina: ACM SIGPLAN, January 2006.

[6] M. Bravenboer, E. Tanter, and E. Visser, “Declarative, formal, and extensible syntax definition for AspectJ,” in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2006, pp. 209–228.

[7] M. Brukman and A. C. Myers, “PPG: A parser generator for extensible grammars,” 2008, http://www.cs.cornell.edu/Projects/polyglot/ppg.html.

[8] S. Chiba and K. Nakagawa, “Josh: an open aspectj-like language,” in *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2004, pp. 102–111.

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

[10] V. O. Di Iorio, C. C. Goulart, L. V. S. Reis, and M. Oikawa, “An application-specific language for synchronization using aspect-oriented programming concepts,” in *Proceedings of the II Latin American Workshop on Aspect-Oriented Software Development*. Institute of Computing, UNICAMP, 2008, pp. 70–79.

[11] V. O. Di Iorio, L. V. S. Reis, R. S. Bigonha, and M. A. S. Bigonha, "A proposal for extensible AspectJ," in *DSAL '09: Proceedings of the 4th workshop on domain-specific aspect languages*. New York, NY, USA: ACM, 2009, pp. 21–24.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Bookman: Addison-Wesley Publishing Company, 1995.

[13] S. S. Huang and Y. Smaragdakis, "Easy language extension with meta-aspectj," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 865–868.

[14] A. J. Korenjak, "Efficient LR(1) processor construction," in *STOC '69: Proceedings of the first annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1969, pp. 191–200.

[15] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005.

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

[17] M. Shonle, K. Lieberherr, and A. Shah, "Xaspects: an extensible system for domain-specific aspect languages," in *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2003, pp. 28–37.