Domain-Specific Aspect Languages for Modularizing Crosscutting Concerns in Grammars

Damijan Rebernak Marjan Mernik
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova ul. 17, 2000 Maribor, Slovenia
{damijan.rebernak, marjan.mernik}@uni-mb.si

Hui Wu Jeff Gray
University of Alabama at Birmingham
Department of Computer and Information Sciences
1300 University Blvd, Birmingham, AL 35294, USA
{wuh, gray}@cis.uab.edu

Abstract

The emergence of crosscutting concerns can be observed in various representations of software artifacts (e.g., source code, models, requirements, and language grammars). Although much of the focus of AOP has been on aspect languages that augment the descriptive power of general purpose programming languages, there is also a need for domain-specific aspect languages that address particular crosscutting concerns found in software representations other than traditional source code. This paper discusses the issues involved in the design and implementation of domain-specific aspect languages that are focused within the domain of language specification. Specifically, the paper outlines the challenges and issues that we faced while designing two separate aspect languages that assist in modularizing crosscutting concerns in grammars.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics, syntax; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects, data type and structures, frameworks, inheritance patterns; D.3.4 [*Programming Languages*]: Processors—compilers, debuggers, interpreters, parsing, preprocessors, compiler generators

General Terms Algorithms, Design, Languages.

Keywords Aspect-Oriented Programming, Domain-Specific Languages, grammars, language specification, join point models

1. Introduction

Over the past decade, many aspect-oriented languages have been proposed, designed and implemented (e.g., AspectC [2], AspectC# [3], and AspectJ [4]). However, the majority of these efforts are devoted to general-purpose aspect languages (GPALs), despite the fact that preliminary work in AOP had its genesis with domain-specific aspect languages (DSALs) [23]. A DSAL is focused on the description of specific crosscutting concerns (e.g., concurrency and distribution) that provide language constructs tailored to the particular representation of such concerns. Examples of DSALs include [9, 12, 30, 32]. In comparison, a GPAL is an aspect language that is not coupled to any specific crosscutting concern and provides general language constructs that permit modularization of a broad range of crosscutting concerns. The majority of DSALs have been developed for languages that are general-purpose programming languages (GPLs) [30]; i.e., the aspect-language is focused on a specific concern, but it is applied to a GPL such as Java or C++. The scope of this paper is focused on the concept of a DSAL that is applied to a domain-specific language (DSL) [24]; i.e., the aspect language is focused on a specific concern and applied to a DSL that also captures the intentions of an expert in a particular domain. The distinction is highlighted by the partitioning of the aspect language (which can be either a GPAL or a DSAL) from the associated component language (which can be either a GPL or DSL). In the DSAL/DSL combination explored in this paper, a different join point model was needed. This paper discusses several issues associated with DSALs applied to DSLs, rather than GPLs.

The focus of this paper is in the well-established domain of programming language definition and compiler generation. Historically, the development of the first compilers in the late-fifties were implemented without adequate tools, resulting in a very complicated and time consuming task. To assist in compiler and language tool construction, formal methods were developed that made the implementation of programming languages easier. Such formal methods contributed to the automatic generation of compilers/interpreters. Several concepts from general programming languages have been adopted into the formalisms used to specify languages, such as object-oriented techniques [27]. To achieve modularity, extensibility and reusability to the fullest extent, new techniques such as aspect-orientation are being used to assist in modularizing the semantic concerns that crosscut many language components described in a grammar [15, 20, 21, 28, 36].

Within a language specification, modularization is typically based on language syntax constructs (e.g., declarations, expressions, and commands). Adding new functionality to an existing language sometimes can be done in a modular way by providing separate grammar productions associated with the extension. For example, additions made to specific types of expressions within a language can be made by changing only those syntax and semantic productions associated with expressions. In such cases, a new feature does not crosscut other productions within the language specification. However, there are certain types of language extensions (e.g., type checking and code generation) that may require changes in many (if not in all) of the language productions represented in the grammar. Because language specifications are also used to generate language-based tools automatically (e.g., editors, type checkers, and debuggers) [16], the various concerns associated with each language tool are often scattered throughout the core language specification. Such language extensions to support tool generation emerge as aspects that crosscut language components [36]. As such, these concerns often represent refinements

over the structure of the grammar [5]. This paper shows how application of aspect-oriented principles toward language specification can assist in modularizing the concerns that crosscut the language grammar.

This paper describes two approaches to integrate AOP with specifications that describe language grammars. Although the approaches are both focused on the common domain of language specification, the two resulting aspect languages apply to different compiler generators; namely, LISA [26] and ANTLR [1]. LISA relies on attribute grammars and ANTLR uses syntax-directed translation. Furthermore, LISA specifications enable higher modularity, extensibility, and reusability through concepts such as multiple attribute grammar inheritance and templates [25]. These differences among LISA and ANTLR contribute to proposing two DSALs that are quite different.

The organization of the paper is as follows. Challenges associated with the design and usage of GPALs and DSALs are discussed in Section 2. The various issues that are encountered when developing domain-specific join point models are presented in Section 3. In Sections 4 and 5, two separate DSALs for language definition (namely, AspectLISA and AspectG) are described. Related work is summarized in Section 6 followed by concluding remarks in Section 7.

2. Challenges Facing General-Purpose and Domain-Specific Aspect Languages

A DSL is a programming language for solving problems in a particular domain that provides built-in abstractions and notations for that domain. DSLs are usually small, more declarative than imperative, and more aligned to the needs of an end-user than general-purpose languages. Use of DSLs has been adopted for a variety of applications because of opportunities for systematic reuse and easier verification [24]. Because of these benefits, DSLs have become more important in software engineering [10, 14]. Moreover, DSLs offer possibilities for analysis, verification, optimization, parallelization, and transformation of DSL code, at a level of specialization not available with general-purpose code.

Similar conclusions can be drawn among GPALs and DSALs. Although GPALs are useful, certain crosscutting concerns are simply best described using DSALs [9, 30]. More importantly, domain-specific analysis and verification can be described by DSALs to prevent the occurence of subtle errors [31]. Yet in other cases, adding new aspects might produce inefficient code and domain-specific optimization is needed [19]. Furthermore, current GPALs (e.g., AspectJ) are not expressive enough to separate all concerns (e.g., structure-shy concerns [30]).

Clearly, in order to address fully the problem of separation of concerns, domain-specific solutions are needed. This has been observed also by other researchers. Gray recognized that specific domains will have numerous dominant decompositions and hence different crosscutting concerns [13]. Consequently, different aspect weavers will be required, even at various levels of abstraction (e.g., models). Hugunin defined four key areas of research that can improve the power and usability of AOP [18]: 1) improved separate compilation and static checking, 2) increased expressiveness for pointcuts, 3) simpler use of aspects in specialized domains, and 4) enhanced usability and extensibility of AOP development tools. Currently, AspectJ has initial support, but not completely sufficient, for particular domains by use of abstract aspect libraries. Not surprisingly, domain-specific aspects are one of the key future research areas in AOSD.

Several of the challenges of using GPALs can be overcome by DSALs. However, DSALs also have their own drawbacks. The most notable challenges of DSALs are: the cost of DSAL development and maintenance, inter-operability with other tools, and user training. One of the most formidable challenges is the extra effort required to design and implement a DSAL. Without an appropriate methodology and tools, the associated costs of introducing a new DSAL can be higher than the savings obtained through usage. With respect to DSLs, there are several techniques available to assist in implementation [24], such as: compiler/interpreter, embedding, preprocessing, and extensible compiler/interpreter. In addition, several tools [22, 26] exist to facilitate the DSL implementation process. We believe that such tools can also assist in DSAL design and implementation.

Another disadvantage of DSALs is that some domains have concerns that require several different DSALs to be developed. In such cases, several DSALs have to coordinate with each other and also interact with a component language. This imposes additional challenges in the design and implementation of DSALs, as well as in user training. It could be argued that it is not feasible to introduce many DSALs because it could overload the ability of the programmers to learn many different languages. However, conscious language design enables programmers to program at much higher abstraction levels and with less code. Conversely, programmers need to write more low-level code without DSALs [30].

3. Domain-Specific Join Point Models

When designing a new DSAL, a completely different join point model (JPM) might be needed as an alternative to the JPM used by a GPAL like AspectJ. The main issues in designing a JPM for a DSAL include:

- What are the join points that will be captured in the DSAL?
- Are the DSAL join points static or dynamic?
- What granularity is required for these join points?
- What is an appropriate pointcut language to describe these joinpoints?
- What are advice in this domain?
- Is extension/refinement only about behavior, or also structure?
- How is information exchanged between join points and associated advice (context exchange)? Is parameterization of advice needed?

In specific domains such as context-dependent computing (e.g., service-oriented and ubiquitous computing), AOP needs to address context passing concerns. Several specific approaches have been proposed such as: contextual pointcut expressions [8], temporal-based context aware pointcuts [17], and context-aware aspects [33]. Such concerns are more easily addressed through DSALs than GPALs [7]. A DSAL designer must also consider the issue of aspect ordering (i.e., how inter-aspect dependencies are handled) and if there is a need to dynamically add/remove aspects during the execution.

Another issue to be considered in the design of a DSAL is the degree that abstraction, reusability, modularity, and extensibility are needed to specify a crosscutting concern that is domain-specific. An abstraction is an entity that embodies a computation [35]. The abstraction principle shows that it is possible to construct abstractions over any syntactic class, provided the phrases of that class specify some kind of computation (e.g., function abstraction, procedure abstraction, and generic abstraction). A GPL provides a large set of powerful abstraction mechanisms, whereas a DSL strives to offer the correct set of predefined abstractions. This is reasonable because a GPL cannot possibly provide the right abstractions needed for all possible applications. Because a DSL has a restricted domain, it is possible to provide some, if not all, of the

desired abstractions. Many DSLs do not provide general-purpose abstraction mechanisms because it is often possible to define a fixed set of abstractions that are sufficient for all the applications in a domain. Hence, we can expect that some DSALs will use fixed and predefined pointcuts and advice with limited possibility for general abstraction. On the other hand, a DSAL that is applied to a general-purpose component language should have more sophisticated constructs for pointcuts and advice. For example, pointcuts should be generic, reusable, comprehensible and not tightly coupled to an application's structure. Tourwe et al. proposed an annotation of inductively generated pointcuts as a solution to this problem [34].

4. AspectLISA

This section describes our investigation into applying aspects to our own language definition tool called LISA. The first subsection introduces LISA and is followed by a discussion of how AspectLISA extends a LISA language specification through aspects that crosscut the language definition.

4.1 LISA: A Domain-Specific Component Language

LISA [22] is a tool that automatically generates a compiler and other language related tools from formal language specifications. The LISA specification notation that is used to define a new language is based on multiple attribute grammar inheritance [25], which enables incremental language development and reusability of specifications. The LISA specification language consists of regular definitions, attribute definitions, rules (which are generalized syntax rules that encapsulate semantic rules), and methods on semantic domains.

The lexical part of a new language definition is denoted by the reserved word **lexicon**. From this part, LISA generates Java source code that implements a scanner for the defined lexicon. Tokens are defined using named regular expressions. Each regular expression has a unique name and can be extended or redefined in a derived language. In the example below, we have two regular definitions: `Commands` and `ReservedWord`. Reserved word **ignore** is used to define characters and tokens that are ignored by the scanner (i.e., not included in the token list). The syntax and semantic parts of a language specification are encapsulated into generalized LISA rules (denoted by the reserved word **rule**). LISA follows the well-known standard BNF notation for defining the syntax of a programming language. Context-free productions are specified in the rule part of a language definition (e.g., `START ::= begin commands end`). Generalized LISA rules serve as an interface for language specifications and may be extended through inheritance. A new language specification inherits the properties of its ancestors and may introduce new properties that extend, modify or override its inherited properties. The semantic part of a language specification is defined by an attribute grammar. Semantic actions must be provided for every production in the **compute** part of a context-free production. To pass values in the syntax tree, non-terminals have attributes. Semantic rules (i.e., attribute calculations) are defined in Java (i.e., the right-hand side of the semantic equation).

In order to illustrate the LISA specification language, the definition of a toy language for robotic control is given below. The robot can move in different directions (left, right, down, up) and the task is to compute its final position. An example of the program is **begin** `down right down` **end** with the meaning {`outp.x=1, outp.y=2`}.

```
language Robot {
lexicon {
  Commands left | right | up | down
  ReservedWord  begin | end
  ignore [\0x0D\0x0A\ ]   // skip whitespaces
}

attributes Point *.inp, *.outp;

rule start {
 START ::= begin COMMANDS end  compute {
   START.outp = COMMANDS.outp;
    // robot position in the beginning
   COMMANDS.inp = new Point(0, 0); };
 }

rule moves {
 COMMANDS ::= COMMAND COMMANDS  compute {
   COMMANDS[0].outp = COMMANDS[1].outp;  // propagation of position
   COMMAND.inp = COMMANDS[0].inp;        // to sub-commands
   COMMANDS[1].inp = COMMAND.outp; }
 | epsilon  compute {  // epsilon (empty) production
   COMMANDS.outp = COMMANDS.inp; };
 }

rule move {
 // each command changes one coordinate
 COMMAND ::= left  compute {
   COMMAND.outp = new Point((COMMAND.inp).x-1,(COMMAND.inp).y); };
 COMMAND ::= right  compute {
   COMMAND.outp = new Point((COMMAND.inp).x+1,(COMMAND.inp).y); };
 COMMAND ::= up  compute {
   COMMAND.outp = new Point((COMMAND.inp).x,(COMMAND.inp).y+1); };
 COMMAND ::= down  compute {
   COMMAND.outp = new Point((COMMAND.inp).x,(COMMAND.inp).y-1); };
 }
}
```

From this language specification, LISA generates highly efficient Java source code that represents the scanner/parser/compiler of the defined language. Additional information about LISA (including software, tutorial, and examples), can be found on LISA's web page [22] and in [25, 26].

4.2 AspectLISA: A Domain-Specific Aspect Language

In language specification there are situations when new semantic aspects crosscut basic modular structure. For example, some semantic rules have to be repeated in different productions; i.e., the introduction of an assignment statement requires variables, which imply definition of the environment and its propagation in all the defined productions. We also identified some other crosscuting concerns, as described in Section 1.

In consideration of the questions stated in Section 3 regarding the JPM for specific domains, join points in AspectLISA are static points in a language specification where additional semantic rules can be attached. These points can be syntactic production rules or generalized LISA rules. A set of join points in AspectLISA is described by a pointcut that matches rules/productions in the language specification. To define a pointcut in AspectLISA, two different wildcards are available. The wildcard '..' matches zero or more terminal or non-terminal symbols and can be used to specify right-hand side matching rules. The wildcard '$*$' is used to match parts or whole literals representing a symbol (terminal or non-terminal symbol). Some examples of pointcut specifications are shown below:

`*.* : * ::= .. ;`
matches any production in any rule in all languages across the current language hierarchy

`Robot.m* : * ::= .. ;`
matches any production in all rules which start with `m` in the `Robot` language

`Robot.move : COMMAND ::= left ;`
matches only a production `COMMAND ::= left` in the rule `move` of the `Robot` language

Pointcuts in AspectLISA are defined using the reserved word **pointcut**. Each pointcut has a unique name and a list of actual parameters (terminals and non-terminals) that denote the public interface for advice. An example of a pointcut that identifies all productions with `COMMAND` as the left-hand non-terminal is:

```
pointcut Time<COMMAND> *.move : COMMAND ::= * ;
```

In AspectLISA, advice are parameterized semantic rules written as native Java assignment statements that can be applied at join points specified by a pointcut. Advice defines additional semantics (extension/refinement) and does not impact the structural (syntax) level of a language specification. In AspectLISA, there is only one way to apply advice on a specific join point due to the fact that attribute grammars are declarative. The order of semantic rules is calculated during compilation/evaluation time when dependencies among attributtes are identified. Therefore, applying advice before/after a join point is not applicable. For the same reason ordering of different aspects is not necessary.

Suppose that the Robot language needs to be extended to incorporate the concept of time. An example advice for time calculation that is applied on join points specified by pointcut `Time` is:

```
advice TimeSemantics<C> on Time { C.time=1; }
```

After weaving takes place, the semantic function `COMMAND.time=1;` is added to all productions within rule `move`. In addition, advice in AspectLISA can have an **apply** part for applying predefined semantic patterns such as: value distribution, list distribution, value construction, list construction, bucket brigade, and propagate value. These semantic patterns are common in attribute grammars and represent fixed abstractions. For example, to propagate environment attributes over an entire evaluation tree, the semantic pattern `bucketBrigadeLeft` should be used (`inEnv` and `outEnv` are attributes used to store and propagate the environment).

```
pointcut All<> *.* : * ::= .. ;
advice EnvProp<> on All apply bucketBrigadeLeft(inEnv, outEnv) { }
```

Modularity, reusability and extensibility of language specification have been much improved in LISA using multiple attribute grammar inheritance [25]. In AspectLISA, pointcuts and advice are also subjects of inheritance. All pointcuts of predecessors can be used in all ancestors. Pointcuts with the same signature (name and parameters) as in ancestors can be used but cannot be extended in inherited languages. Such pointcuts are overriden by default. Advice inherited from ancestors using the **extends** keyword must be merged with the advice in the specific language. If advice exists in the inherited parent language, then the semantic functions of the advice must be merged; otherwise, advice are simply copied from the inherited language to the current language. Advice can also override the semantics of its parent using the keyword **override**.

An example of inheritance on advice is shown below. Note that the pointcut on which this advice is applied is inherited.

```
advice extends TimeSemantics<C> {
C.time=1.0 / C.inspeed; C.outSpeed = C.inspeed; }
```

4.2.1 Aspect Weaving in AspectLISA

The crucial part of every aspect-oriented compiler is an aspect weaver that is responsible for appending advice code into appropriate places described by pointcuts.

Weaving takes place after the initial phase of LISA's compiler, which is responsible for parsing the LISA source and generating the necessary data structures for pointcuts and advice. The main weaving algorithm is described by Algorithm 1.

Algorithm 1 Main weaving algorithm

```
method weaveAll(lastLanguage)
// lastLanguage is last language in hierarchy
Language_list ← allDefinedLanguages
for all L ∈ Language_list do
  L ← nextElement(Language_list)
  // if L is not part of language hierarchy the weaving
  // in that Language is not necessary
  if L is reachable from lastLanguage then
    weave(L)
  end if
end for
```

Weaving starts at the first (parent) language (component) defined by the developer and follows its hierarchy. The same algorithm is applied to each language specification over the entire hierarchy of languages. The weaving procedure for each user-defined language is described by Algorithm 2. Note that the lookup method (*pointcutLookup(A, L)*) works the same as in most compilers for object-oriented languages.

Algorithm 2 Weaving algorithm for one Language

```
method weave(L)
Advice_set ← getAllAdvice(L)
for all A ∈ Advice_set do
  A ← nextElement(Advice_set)
  // advice must not be overriden by none of its successors
  if A is not overriden then
    // find appropriate pointcut in current or parrent languages
    pointcut ← pointcutLookup(A, L);
    // find all production rules that match pointcut
    productionRules ← findProductions(pointcut, L);
    for all prodRule ∈ productionRules do
      prodRule ← nextElement(productionRules)
      // substitute formal parameters of advice with actual
      // parameters of pointcut and apply semantic functions
      // to the production
      addSemanticsToRule(prodRule, A, L)
    end for
  end if
end for
```

5. AspectG

This section describes our second investigation into a DSAL for language specification. The first subsection introduces ANTLR as the DSL representing the component language. The second subsection provides a discussion of AspectG, which is our DSAL that weaves crosscutting concerns into ANTLR grammars.

5.1 ANTLR: A Domain-Specific Component Language

ANTLR (ANother Tool for Language Recognition) is a parser generator that provides a framework for constructing various programming language related tools (e.g., recognizers, compilers, and translators) from grammatical specifications [1]. The ANTLR specification language is based on EBNF notation and enables syntax-directed generation of a compiler. The tokens comprising the lexical part of the grammar for the new language are defined using named regular expressions. The parser representing the semantic part of the language specification is defined as a subclass of the grammar specification and encapsulates semantic rules within each

grammar production. The semantic actions within each production rule are written in a GPL (e.g., Java, C#, C++, or Python). The Robot language described in Section 4 has been rewritten in ANTLR and partially provided below. This simple example illustrates the ANTLR specification language with semantic rules defined in Java. AspectG follows the DSL implementation pattern using a pre-processor that serves as a compiler and application generator to perform a source-to-source transformation (i.e., the DSL source code is translated into the source code of an existing GPL). The ANTLR specification of the Robot language translates Robot code into the equivalent Java code (e.g., Robot.java) that can be compiled and executed on the Java Virtual Machine.

```
// The following class represents the Robot parser in ANTLR
class P extends Parser; {FileIO fileio=new FileIO();}
root:(
       BEGIN
       {
       fileio.print("public class Robot");
       fileio.print("{");
       fileio.print("public static void main(String[] args) {");
       fileio.print("int x = 0;");
       fileio.print("int y = 0;");
       }
      commands
      END EOF!
       {
       fileio.print("System.out.println(\"x coord= \" + x +
                    \" \" + \"y coordinator= \" + y);");
       fileio.print("   }");
       fileio.print("}");
       fileio.end();
       }
      );
commands:(  command commands
        |
         );
command :(
         LEFT  {fileio.print("x=x-1;");
                fileio.print("time=time+1;");}
        |RIGHT {fileio.print("x=x+1;");
                fileio.print("time=time+1;");}
        |UP    {fileio.print("y=y+1;");
                fileio.print("time=time+1;");}
        |DOWN  {fileio.print("y=y-1;");
                fileio.print("time=time+1;");});

// The following class represents the Robot lexer in ANTLR
class L extends Lexer;
BEGIN : "begin";
END   : "end";
LEFT  : "left";
RIGHT : "right";
UP    : "up";
DOWN  : "down";
// whitespace
WS  :   (   ' '
        |   '\t'
        |   '\r' '\n' { newline(); }
        |   '\n' { newline(); }
        )       {$setType(Token.SKIP);} ;
```

From the above language specification, ANTLR generates Java source code representing the scanner and parser for the Robot language. Additional information about ANTLR can be found on the ANTLR web page [1].

5.2 AspectG: A Domain-Specific Aspect Language

In our past work [36], we noticed that crosscutting concerns emerged within the grammar of the language specification. In particular, the implementation hooks for various language tools (e.g., debugger and testing engine) required modification to be made to every production in the grammar. Manually changing the grammar through invasive modifications proved to be a very time consuming and error prone task. It is difficult to build new testing tools for each new language of interest and for each supported platform because each language tool depends heavily on the underlying operating system's capabilities and lower-level native code functionality [29].

We developed a general framework called the DSL Testing Tool Studio (DTTS), which assists in debugging, testing, and profiling a program written in a DSL. Using the DTTS, a DSL debugger and unit test engine can be generated automatically from the DSL grammar provided that an explicit mapping is specified between the DSL and the translated GPL. To specify this mapping, additional semantic actions inside each grammar production are defined. A crosscutting concern emerges from the addition of the explicit mapping in each of the grammar productions. The manual addition of the same mapping code in each grammar production results in much redundancy that can be better modularized using an aspect-oriented approach applied to grammars. In the case of generating a debugger for the Robot language, the debug mapping for the Robot DSL debugger was originally specified manually at the Robot DSL grammar level shown below. For example, line 12 to line 18 represents the semantic rule of the LEFT command. Line 12 keeps track of the Robot DSL line number; line 14 records the first line of the translated GPL code segment; line 16 marks the last line of the translated GPL code segment; line 17 and line 18 generate the mapping code statement used by the DTTS. These semantic actions are repeated in every terminal production of the Robot grammar.

```
10  command
11   :( LEFT {
12       dsllinenumber=dsllinenumber+1;
13       fileio.print(" x=x-1;");
14       gplbeginline=fileio.getLinenumber();
15       fileio.print(" time=time+1;");
16       gplendline=fileio.getLinenumber();
17       filemap.print("mapping.add(newMap(" + dsllinenumber +
                        ",\"Robot.java\"," +
18                    gplbeginline + "," + gplendline + "));");}
19     |RIGHT {
20       dsllinenumber=dsllinenumber+1;
21       fileio.print(" x=x+1;");
22       gplbeginline=fileio.getLinenumber();
23       fileio.print(" time=time+1;");
24       gplendline=fileio.getLinenumber();
25       filemap.print("mapping.add(newMap(" + dsllinenumber +
                        ",\"Robot.java\"," +
26                    gplbeginline + "," + gplendline + "));");}
```

The same mapping statements for the RIGHT command appear in lines 20, 22, and 24 to 26. Although the Robot DSL is simple, it is not uncommon to have grammars with hundreds of production rules. In such cases, much redundancy will exist because the debug mapping code is replicated across each production. Of course, because the debug mapping concern is not properly modularized, changing any part of the debug mapping has a rippling effect across the entire grammar. An aspect-oriented approach can offer much benefit in such a case. We have created AspectG as a tool to help us manage crosscutting concerns in ANTLR language specifications.

The AspectG pointcut model can match on both the syntax of the grammar and the semantic rule within each production (written in Java). Join points in ANTLR are static points in the language specifications where additional semantic rules can be attached. A set of join points in AspectG is described with pointcuts that define the location where the advice is to apply. A wildcard can be used within the signature of a pointcut. The wildcard '$*$' matches zero or more terminal or non-terminal symbols to represent a set of qualified join points. Some examples of pointcut specifications are shown below:

`*.*;` *matches any production in the entire Robot language*
`command.*;` *matches any production in a command production in the Robot language*

Pointcuts in AspectG are defined using the reserved word **pointcut** and two keywords (e.g., **within** and **match**). The **within** predicate is used to locate grammar productions at the syntax level and **match** is used to define the location of a GPL statement within a semantic rule. Each pointcut has a unique name and a list of actual parameter signatures (terminals and non-terminals) and semantic rules. Considering the following pointcut:

```
pointcut productions(): within(command.*);
```

The pointcut called `productions` is defined with the wild card `command.*` and matches each join point that is a command production in a grammar (e.g., RIGHT). As an example of a pointcut that combines both predicate types, consider the following:

```
pointcut count_gpllinenumber(): within(command.*) &&
match(fileio.print("time=time+1;"));
```

The pointcut `count_gpllinenumber` is a pattern specification corresponding to command productions having a semantic action with a statement matching the signature `fileio.print("time=time+1;")`. The advice in AspectG is defined in a similar manner to AspectJ, which brings together a pointcut that selects join points and a body of code representing the effect of the advice [4]. The advice are semantic rules written as native Java statements that can be applied at join points specified by pointcuts. Unlike LISA, in ANTLR the order of GPL statements in semantic rules is very important. Therefore, in AspectG the ability to apply advice before/after a join point is necessary, as shown in the example below.

```
before(): productions() { dsllinenumber=dsllinenumber+1;}
after(): count_gpllinenumber() {
gplbeginline=fileio.getLinenumber();}
```

The **before** advice defined on the `productions` pointcut means that before the parser proceeds with execution of each command production, the DSL line number is incremented (i.e., `dsllinumber=dsllinenumber+1;`). The **after** advice associated with the `count_gpllinenumber` means that line numbers for the GPL are updated (i.e., `gplbeginline=fileio.getLinenumber();`) after the parser matches a timer increment (i.e., `fileio.print("time=time+1;");`).

The changes in terms of aspects automatically propagate into the generated parser through the modified grammar productions. After weaving a grammar aspect and parsing the Robot DSL code, the new ANTLR grammar can generate the mapping information that contains the information needed by the DTTS (i.e., each Robot DSL code statement line number along with its corresponding generated Java statement line numbers is recorded in the grammar).

5.2.1 Aspect Weaving in AspectG

Unlike AspectLISA's compiler approach, AspectG uses a program transformation system (specifically, we use DMS - the Design Maintenance System [6]) to perform the underlying weaving on the language specification. The AspectG abstraction hides the details of the accidental complexities of using the transformation system from the users; i.e., a user of AspectG focuses on describing the crosscutting grammar concerns at a higher level of abstraction using an aspect language, rather than writing lower level program transformation rules [36]. In AspectG, each of the crosscutting concerns is modularized as an aspect that is weaved into an ANTLR grammar using parameterized low-level transformation functions.

We have developed four weaving functions to handle four different types of join points that may occur within a grammar. The four possible join points provided by AspectG are: before a semantic action; after a semantic action; before a specific statement that is inside a semantic action; and, after a specific statement that is inside a semantic action. These joint points are represented in AspectG by **before** and **after** keywords within the context of a semantic action or specific statement. Weaving takes place after the initial phase of AspectG's compiler, which is responsible for parsing the AspectG specification and generating the program transformation rules. The generated program transformation rules provide bindings to the appropriate weaving function parameters corresponding to the pointcut and advice defined in the aspect language. Algorithm 3 describes the weaving procedure for AspectG. Note that the algorithm requires two parameters (advice and a join point) and weaves the advice parameter into the join points designated by the pointcut predicate. Additional technical details are provided in [11].

Algorithm 3 AspectG weaving

```
for all jp ∈ pointcuts_list do
  for all a ∈ advices_list do
    if jp's name equals a's pointcut name then
      weave(jp, a)
    end if
  end for
end for
```

The actual weaving of the language specification is done by the DMS program transformation engine according to the different program transformation rules generated by the AspectG compiler. The weave method first looks for all potential pointcut positions in the semantic sections of a grammar. The weaver then back tracks to the pointcut's ancestor node type and value in the syntax level to filter out the unqualified pointcut positions. Finally, the advice is inserted in the correct position of the grammar specification using the ASTInterface API provided by DMS, which provides methods for modifying a given syntax tree to regenerate a new tree structure.

6. Related work

AspectASF [21] is a simple DSAL for language specifications written in the ASF+SDF formalism. Only rewrite rules are supported. Therefore, join points in AspectASF are static points in equation rules describing semantics of the language. The pointcut pattern language in AspectASF is a simple pattern matching language on the structure of equations where only labels and left-hand sides of equations can be matched. Pointcuts can be of two types: entering an equation (after a successful match of left-hand side) and exiting an equation (just before returning the right-hand side). Advice specify additional equations that are written in the ASF formalism. The AspectASF weaver transforms the original language specifications by augmenting the base grammar with new concerns (i.e., additional equations are appended to appropriate places in the grammar).

An early approach of aspect-orientation in language specifications is presented in the compiler generator system JastAdd [15]. The JastAdd system is a class weaver: it reads all the JastAdd modules (aspects) and weaves the fields and methods into the appropriate classes during the generation of the AST classes. This approach does not follow the conventional join point model where join points are specified using a pointcut pattern language. However, it can be seen as inter-type declarations in AspectJ where join points are all non-anonymous types in the program and pointcuts are the names of classes or interfaces. Hence, JastAdd uses implicit joint points while AspectLISA and AspectG use explict joint points described by pointcuts. Moreover, JastAdd does not enable inheritance on advice and pointcuts as AspectLISA does.

7. Conclusion

Domain-specific aspect languages (DSALs) represent a focused approach toward providing a language that allows a programmer or end-user to define a specific type of concern. DSALs can be contrasted with general-purpose aspect languages (GPALs) that provide a more general language for capturing a broader range of crosscutting concerns. Within the research on DSALs, much of the application is centered on specific concerns for a language like Java or C++. This paper differs from the scope of general research by describing our investigation into DSALs for DSLs such as language specification.

The paper summarized the challenges of DSAL development and presented two separate case studies of different DSALs applied to two different languages. Future work includes new pointcut predicates that assist in specifying the control flow within a grammar. Such a predicate would allow aspects associated with various forms of run-time analysis to be specified and captured.

References

[1] ANTLR – ANother Tool for Language Recognition. http://www.antlr.org, 2006.

[2] AspectC. http://www.aspectc.org/, 2006.

[3] AspectC#. http://www.castleproject.org/index.php/aspectsharp, 2006.

[4] AspectJ. http://eclipse.org/aspectj/, 2006.

[5] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

[6] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformation for practical scalable software evolution. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 625–634, 2004.

[7] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of Joint European Software Engineering Conference (ESEC)*, pages 88–98, 2001.

[8] T. Cottenier and T. Elrad. Contextual pointcut expressions for dynamic service customization. In *Dynamic Aspects Workshop (DAW)*, pages 95–99, 2005.

[9] C. Courbis and A. Finkelstein. Towards aspect weaving applications. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 69–77, 2005.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[11] DSL Testing Tool Studio. http://www.cis.uab.edu/wuh/ddf/, 2006.

[12] J. Fabry and T. Cleenewerck. Aspect-oriented domain-specific languages for advanced transaction management. In *Proceedings of International Conference on Enterprise Information Systems (ICEIS)*, pages 428–432, 2005.

[13] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM, Special Issue on Aspect-Oriented Programming*, pages 87–93, October 2001.

[14] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software factories: assembling applications with patterns, models, frameworks and tools*. Wiley Publishing, 2004.

[15] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[16] P. Henriques, M. Varanda Pereira, M. Mernik, M. Lenič, J. Gray, and H. Wu. Automatic generation of language-based tools using LISA. *IEE Proceedings - Software Engineering*, 152(2):54–69, April 2005.

[17] C. Herzeel, K. Gybels, and P. Costanza. A temporal logic language for context awareness in pointcuts. In *ECOOP Workshop: Revival of Dynamic Languages*, 2006.

[18] J. Hugunin. The next steps for aspect-oriented programming languages. In *NSF Workshop on Software Design and Productivity*, 2001.

[19] J. Irwin, J. Loingtier, J. R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *Proceedings of International Scientific Computing in Object-Oriented Parallel Environments*, 1997.

[20] K. T. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. *Electr. Notes Theor. Comput. Sci*, 147(1):5–30, 2006.

[21] P. Klint, T.van der Storm, and J.J. Vinju. Term rewriting meets aspect-oriented programming. Technical report, CWI, 2004.

[22] LISA. http://marcel.uni-mb.si/lisa, 2006.

[23] C. Lopes. *Aspect-Oriented Programming: A Historical Perspective*. In Aspect-Oriented Software Development, R. Filman, T. Elrad, M. Aksit, S. Clarke (eds.), Addison-Wesley, 2004.

[24] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[25] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Multiple Attribute Grammar Inheritance. *Informatica*, 24(3):319–328, 2000.

[26] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. LISA: An Interactive Environment for Programming Language Development. In *Proceedings of International Conference on Compiler Construction (CC)*, pages 1–4, 2002.

[27] M. Mernik, X. Wu, and B. Bryant. Object-oriented language specifications: Current status and future trends. In *ECOOP Workshop: Evolution and Reuse of Language Specifications for DSLs (ERLS)*, 2004.

[28] D. Rebernak, M. Mernik, P. R. Henriques, and M. J. V. Pereira. AspectLISA: an aspect-oriented compiler construction system based on attribute grammars. In *Workshop on Language Descriptions, Tools and Applications (LDTA)*, pages 44–61, 2006.

[29] J. B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley and Sons, 1996.

[30] M. Shonle, K. Lieberherr, and A. Shah. XAspects: An extensible system for domain-specific aspect languages. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 28–37, 2003.

[31] J. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: An aspect-based composition tool for real-time systems. In *Real-Time Applications Symposium*, pages 58–69, 2003.

[32] D. Suvee, W. Vanderperren, and V. Jonckers. Jasco: An aspect-oriented approach tailored for component based software development. In *Proceedings of International Conference on Aspect-oriented Software Development (AOSD)*, pages 21–29, 2003.

[33] E. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-aware aspects. In *Proceedings of International Symposium on Software Composition*, pages 227–249, 2006.

[34] T. Tourwe, A. Kellens, W. Vanderperren, and F. Vannieuwenhuyse. Inductively generated pointcuts to suppport reafctoring to aspects. In *AOSD Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT)*, 2004.

[35] D. A. Watt. *Programming Language Concepts and Paradigms*. Prentice-Hall, 1990.

[36] H. Wu, J. Gray, S. Roychoudhury, and M. Mernik. Weaving a debugging aspect into domain-specific language grammars. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, pages 1370–1374, 2005.