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.

1 Introduction

Over the past decade, many aspect-oriented programming (AOP) 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) [18]. 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, 10, 26, 27]. 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) [26]; 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) [20]; 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 within the well-established domain of programming language definition and compiler generation. Historically, the development of the first compilers in the late-1950s was implemented without adequate tools, resulting in a very complicated and time consuming task. To assist in compiler and language tool construction, formal methods for language specification and supporting tools were developed that made the implementation of programming languages easier. Such formal methods contributed to the automatic generation of compilers and interpreters. Several concepts from general-purpose programming languages have been adopted into the formalisms used to specify languages, such as object-oriented techniques [23]. 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. Examples of new techniques include JastAdd [11], AspectStratego [14], AspectASF [15], AspectLISA [24], and AspectG [32].

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, environment propagation, 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) [12], 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 [32]. 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 [22] and ANTLR [1]. LISA relies on attribute grammars (AGs) 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 in AGs [21]. These differences among LISA and ANTLR contribute to proposing two DSALs that are quite different.

The organization of the paper is as follows. The various issues that are encountered when developing domain-specific join point models are presented in Section 2. In Sections 3 and 4, two separate DSALs for language definition (namely, AspectLISA and AspectG) are described. Section 5 discusses the lessons learned from our experiences in designing AspectLISA and AspectG. The section compares the two languages and also summarizes related work. The paper concludes with Section 6, which summarizes the contribution of the paper and overviews some challenges representing future work in the area of aspect languages for grammars.

2 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:

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

In specific domains such as context-dependent computing (e.g., service-oriented and ubiquitous computing), an aspect language may need to address context passing concerns. Several specific approaches have been proposed, such as: contextual pointcut expressions [8], temporal-based context aware pointcuts [13], and context-aware aspects [28]. 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 or 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 [30]. 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 [29].

3 AspectLISA

This section describes our own language definition tool called LISA and investigations into applying aspects to its formal specification language. The first subsection introduces the LISA tool and its Domain-Specific Component Language for language definition. A discussion of how AspectLISA extends a LISA language specification through aspects that crosscut the language definition follows. The section is concluded with an explanation of an example language defined in AspectLISA.

3.1 LISA: A Domain-Specific Component Language

LISA [17] is a tool (compiler-compiler) that automatically generates a compiler and other language related tools from formal attribute grammar (AG) [16] based language specifications. LISA uses a special specification notation (i.e., a specially designed DSL) to define a new language. The specification language is designed to support modularity, reusability and incremental language development through mechanisms like multiple inheritance, templates in AGs and aspects. Specifications are stored in one or more `.lisa` files (reusable language components) that are then transformed by multiple steps into a monolithic language specification. The output from the last transformation phase are three separate monolithic specifications (lexical, syntactic, and semantic) from which the source code (in Java) for a scanner, parser, and evaluator are generated. One can use the LISA tool in textual mode, as a compiler generator, or LISA's IDE can be used to specify, generate, compile-on-the fly, and execute programs in a newly specified language. When using LISA's IDE the users can also benefit from various language-based tools that are generated from input specifications [12]. LISA also generates a syntax-aware editor and various inspectors, such as a finite state automata visualizer and syntax and semantic tree animators that are useful for understanding the behavior of the newly specified language. Users of the generated compiler or interpreter also have the possibility to visually observe the work of syntax and semantic analyzers by watching the animation of a parse and semantic tree. The animation shows the parser in action and the values of the attributes as they change. Also, the syntax and the semantic trees are automatically updated as the program executes. Animated visualizations help to explain the inner workings of programs and are a useful tool for debugging and teaching.

LISA uses a specially designed DSL to define a new language. The LISA specification language consists of the following components:

Lexical definitions

The lexical part of a 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 (i.e, regular definitions). Each regular expression has a unique name and can be extended or redefined in a derived language through inheritance. In the example shown in Figure 1, there are 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).

Generalized syntax rules

The syntax and semantic parts of a language specification are encapsulated into generalized syntax rules that are 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 ;` where the first character of non-terminal symbols must be a capital letter). Generalized LISA rules serve as an interface for language specifications and may be extended or overriden in a derived language through inheritance. 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 generalized syntax rules. In order to pass values in the syntax tree, non-terminals have attributes that define semantics forming a semantic tree. Attribute evaluation rules are associated with each context-free production. Semantic rules (i.e., attribute calculations) are defined in Java (i.e., the right-hand side of the semantic equation) and can take advantage of all standard Java libraries.

Attribute definitions

Using an attribute grammar approach, the semantics of a programming language are defined by attributes that are provided for each non-terminal symbol. The type of an attribute in LISA can be any valid Java type. Usually, the same attribute name is attached to many different non-terminals. For that reason, the wildcard `*` can be used. The attribute definition part of a language specification is denoted by the reserved word **attributes**.

Operations on semantic domains

Semantic domains and operation on these domains in LISA can be written in the method section of specifications (denoted by reserved word **method**; depicted at the end of the example in Figure 3). Because LISA generates the source code of a parser and compiler in Java, it is natural to define the semantic domain in the same programming language. Operations on semantic domains can also be imported from existing packages (e.g., `import Robot.calculations.* ;`).

In order to illustrate the LISA specification language, the definition of a toy language for robotic control is depicted in Figure 1. The robot can move in different directions (left, right, down, up) and the task is to compute its final position. An example Robot program is **begin** `up right up` **end** with the result `{outp.x=1, outp.y=2}` (assuming robot position is initialized to 0).

LISA enables modular and incremental language development with a mechanism called multiple attribute grammar inheritance. A new language specification can inherit the existing features of its ancestor(s) and may introduce new features that extend or override its inherited features. All features in the language definition provided in LISA (e.g., lexical definitions, generalized syntax rules, attribute definitions, and operations on semantic domains) are the subject of inheritance. One can reuse or modify (reserved word **extends**) or redefine (reserved word **override**) some of the features of ancestor language(s). Further discussion on multiple attribute grammar inheritance is outside of the scope of this paper. Additional information about LISA, and its features (including software, tutorial, and examples), can be found on LISA's web page [17] and in [21, 22].

3.2 AspectLISA: A Domain-Specific Aspect Language

In language specification there are situations when new semantic aspects crosscut basic modular structure. For example, when the same 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 defined productions. Changes of such magnitude are hard to implement with conventional

```
language Robot {
 lexicon {
  Commands      left | right | up | down
  ReservedWord  begin | end
  ignore        [\0x0D\0x0A\ ]
 }
 attributes java.awt.Point *.inp, *.outp;
 rule start {
  START ::= begin COMMANDS end  compute {
    START.outp = COMMANDS.outp;
    COMMANDS.inp =  new java.awt.Point(0, 0);
  };
 }
 rule commands {
  COMMANDS ::= COMMAND COMMANDS  compute {
    COMMANDS[0].outp = COMMANDS[1].outp;
    COMMAND.inp = COMMANDS[0].inp;
    COMMANDS[1].inp = COMMAND.outp;
  } |  epsilon  compute {
    COMMANDS.outp = COMMANDS.inp;
  };
 }
 rule command {
  COMMAND ::= left  compute {
    COMMAND.outp =
       new java.awt.Point(( int)COMMAND.inp.getX()-1, ( int)COMMAND.inp.getY());
  };
  COMMAND ::= right  compute {
    COMMAND.outp =
       new java.awt.Point(( int)COMMAND.inp.getX()+1, ( int)COMMAND.inp.getY());
  };
  COMMAND ::= up  compute {
    COMMAND.outp =
       new java.awt.Point(( int)COMMAND.inp.getX(), ( int)COMMAND.inp.getY()+1);
  };
  COMMAND ::= down  compute {
    COMMAND.outp =
       new java.awt.Point(( int)COMMAND.inp.getX(), ( int)COMMAND.inp.getY()-1);
  };
 }
}
```

Figure 1: Specifications for the `Robot` language in LISA

inheritance, because each change crosscuts all parts of the language. In order to increase modularity and improve reusability of language components, we introduced aspect-oriented features into LISA's specification language, called AspectLISA. Because LISA uses attribute grammars as a formal model for language definition, we call this extension Aspect-Oriented Attribute Grammars (AOAGs) [24].

In order to introduce aspect-oriented features into LISA, we extended the original language with pointcut definitions, advice, and advice application. One of the crucial parts of an aspect-oriented language is the join point model (JPM). In consideration of the questions stated in Section 2 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 in AspectLISA are production rules.

Pointcut

A set of join points in AspectLISA is described by a pointcut that matches rules or 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 that 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 must have a unique name. An example of a pointcut that identifies all productions with `COMMAND` as the left-hand non-terminal in the generalized LISA rule `move` is:

```
pointcut Command *.move :  COMMAND ::= ..  ;
```

Advice

In AspectLISA, advice are polymorphic abstractions of semantic rules parametrized with any number of terminals, non-terminals or attributes. These parameters can be associated with many production rules with different non-terminal and terminal symbols. The mechanism for specifying semantics in advice is similar as templates in AGs [19, 21]. Advice define additional semantics through extension or refinement, and do not impact the structural (i.e., syntax) level of a language specification, which is a crucial difference between templates in AGs and advice. Advice can be applied at join points specified by one or more pointcuts. In AspectLISA, there is only one way to apply advice on a specific join point due to the fact that AGs are declarative. The order of semantic rules is calculated during compilation or evaluation time when dependencies among attributes are identified. Therefore, applying advice before or after a join point is not applicable. For the same reason, ordering of different aspects is not necessary. Advice are defined using reserved word **advice**. Each advice has a unique name and a list of formal parameters.

Suppose that the Robot language needs to be extended to incorporate the concept of time (i.e., language `RobotTime`). An example advice for time calculation is shown as follows:

```
advice CommandTime<C> { C.time = 1; } ;
```

The above advice can be applied to several different pointcuts using the `apply` construct in AspectLISA. The abstractions of semantic rules that are independent of the structure of production rules can be specified in AspectLISA. These abstractions can be used for specifying common patterns in language specifications, such as value distribution, value construction, bucket brigade, list distribution, and many others. As an example, a bucket brigade left pattern is described as follows:

```
Y ::= X1 X2 ... XN
 { X1.in = Y.in; X2.in = X1.out;
 .
 .
 .
 XN.in = XN-1.out; Y.out = XN.out; }
```

An advice describing this pattern is as follows:

```
  if (empty(RHS)) {
   LHS.outAtt = LHS.inAtt ;
  }
  else {
   first(RHS).inAtt = LHS.inAtt;
   RHS.inAtt = pred(RHS).outAtt;
   LHS.outAtt = last(RHS).outAtt;
  }
}
```

Reserved words LHS and RHS denote a left-hand side non-terminal, and a list of right-hand side non-terminals of a production. Together with a list of arguments, some functions are defined that can be used for variable list manipulation (`first, last, succ,` and `pred`). A successor (function `succ`) for the last argument, and predecessor for the first argument (function `pred`) do not exist; therefore, a semantic rule is not generated in that case.

Advice application

The advice application part of a language specification in AspectLISA is the part where advice are applied to join points. This can be done using the reserved word **apply** followed by the advice name with actual parameters, reserved word **on**, and a list of pointcuts. The first step of applying advice is to lookup and compare actual parameters of each advice. The lookup for appropriate pointcuts follows similarly. After pointcuts are known, the next step is to apply the semantics defined in each advice as bound by a pointcut. This will be explained in more detail in subsection 3.2.2. As an example:

```
apply CommandTime<COMMAND> on Command ;
```

defines an application of aspect `CommandTime`, where pointcut `Command` affects all productions in rule `move`. Without aspect-oriented constructs, it is necessary to repeat and manually add several semantic actions in many productions. The "original" specifications of this part of the language would be:

```
rule  extends command {
 COMMAND ::= left  compute {
   COMMAND.time = 1;
 };
 COMMAND ::= right  compute {
   COMMAND.time = 1;
 };
 COMMAND ::= up  compute {
   COMMAND.time = 1;
 };
 COMMAND ::= down  compute {
   COMMAND.time = 1;
 };
}
```

When advice is applied to join points, the semantic rule for a particular attribute may already be defined. The question arises, whether to overwrite this semantic rule or ignore its application? By default, the application of such semantic rules defined in advice is ignored. For that reason we introduce reserved word **overwrite** that serves as a flag to the aspect weaver to redefine already defined semantic rules in a production rule. For an example, if we would use **overwrite** in the following statement:

```
apply overwrite bucketBrigadeLeft<inEnv, outEnv> on *.* :  * ::= ..  ;
```

some already defined semantic rules in productions that match pointcut `*.* : * ::= .. ;` (i.e., in this case, all productions) would be overwritten.

3.2.1 Multiple Aspect-Oriented Attribute Grammar Inheritance

Modularity, reusability and extensibility of language specifications have been much improved in LISA using multiple attribute grammar inheritance [21]. In order to increase reusability and modularity of aspect-oriented features in AspectLISA, pointcuts and advice are also subjects of inheritance. Formal definition of multiple AG inheritance needs to be adopted to integrate aspect-oriented features - we call this extension multiple aspect-oriented attribute grammar inheritance. An explanation of how multiple aspect-oriented attribute grammar inheritance works for each aspect-oriented feature of AspectLISA is out of the scope of this paper.

3.2.2 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. We extend LISA's compiler in order to implement weaving in AspectLISA. Therefore, from a user point of view, the weaving process in AspectLISA is hidden within the compile phase of LISA's compiler generator. The main weaving algorithm is described as follows:

```
method weaveAll(lastLanguage)
 // lastLanguage is last language in hierarchy, where starting production is defined
 for each defined_apply_statement  do
  apply_st = next(defined_apply_statement);
  advice_pointcuts_set = find_matching_pairs(apply_st); // (advice, set of pointcuts)
  // lookup for advice and pointcuts is implemented in similar manner as in all OO languages
  for each advice_pointcuts_set  do
   advice = next(advice_pointcuts_set).advice;
   pointcuts = next(advice_pointcuts_set).pointcuts;
   apply(advice, pointcuts);
  endfor
 endfor
endmethod

method apply(advice, pointcuts)
 for each pointcuts  do
  pointcut = next(pointcuts);
  // 1. find all sets of production rules
  // 2. substitute formal parameters of advice with actual parameters
  // 3. apply semantic rules to each production rule
  apply_advice(advice, pointcut);
 endfor
endmethod
```

Weaving starts with the language production that is at the highest level in the language components hierarchy. The same algorithm is applied to each **apply** statement that defines a unique advice–pointcut pair. The first step is to find[1] all advice and matching set of pointcuts (due to pointcut inheritance, each pointcut can define a set of pointcuts). Then, semantic rules of each advice are merged with semantic rules of production rules. If **overwrite** is defined, then all semantic rules are copied to the production rule. Already existing (duplicated) semantic rules for a particular attribute are substituted with semantic rules defined in advice. When **override** is not specified, the semantic rules from the advice are ignored. The search for appropriate join points (i.e., production rules) also take place at weaving time.

3.3 AspectLISA Illustrative Example

In the example of extending the `Robot` language with the semantics for time calculations (e.g., the language `RobotTime` example from section 3.2), it can be observed that aspect-oriented features are very useful for extending language semantics. In this example, we need to extend (i.e., redefine in derived language component) all seven production rules in three generalized LISA rules and add a new semantic rule to each of them. Because the same semantic rule has to be applied several times, this emerges as a crosscutting concern and can be solved more effectively using aspects. In this case the semantic rule is specified only once in advice and can be easily changed or removed.

Another benefit of the same advice–pointcut definition can be noticed on the `RobotCalc` example in Figure 2. In this case we integrate features from another language component in order to extend the original language through multiple inheritance. Because we changed the syntax of the language, all the productions where syntax is changed have to be extended with new semantic rules in order to take into account new attributes from the parent language(s). A semantic rule for time calculation defined in the advice does not have to be redefined in derived production rules because a pointcut match from an ancestor language component also changes the production rules of a new extension. With a manual approach using only inheritance, all semantic rules for time calculation would have to be redefined (i.e., written once more) in all redefined production rules, which is a time consuming and error-prone task. A fragment of the specifications for language `RobotCalc` is depicted in Figure 2. The non-terminal `EXPR` is inherited from language component `Expressions`, which implements a language for arithmetic expressions. An example of a program written

[1]Advice and pointcut lookup works the same as in most compilers for object-oriented languages with multiple inheritance.

```
language RobotCalc  extends RobotTime, Expressions {
 ...
 rule command {
   COMMAND ::= left EXPR  compute {
     COMMAND.outp =   new java.awt.Point(( int)COMMAND.inp.getX()-EXPR.val, ( int)COMMAND.inp.getY());
   };
   COMMAND ::= right EXPR  compute {
     COMMAND.outp =   new java.awt.Point(( int)COMMAND.inp.getX()+EXPR.val, ( int)COMMAND.inp.getY());
   };
   ...
 }
 ...
}
```

Figure 2: Specifications for the `RobotCalc` language

in the `RobotCalc` language is: **begin** `down 5 right 3*(2+1)` **end** with the result of `{time=2; outp.x=9; outp.y=-5}`.

In Figure 3, we introduce an assignment statement into the `RobotCalc` language. An example of such a program is: **begin** `left ID=10*(5+2) up ID-10` **end** with the meaning `{time=2; outp.x=70; outp.y=60}`. An assignment statement requires variables that imply the definition of an environment and its propagation in all defined production rules. From a developers point of view, this means all generalized LISA rules have to be extended in all production rules (i.e., 17 rules; from all ancestor language components) where new semantics for environment propagation have to be added. Because environment propagation can be identified as a `bucketBrigadeLeft` pattern, the implementation should be simplified. One of the improvements can be achieved using templates in AGs. The semantics for environment propagation is in this case defined in a template that is further instantiated at appropriate production rules. The drawback of this approach (at least in this particular case) is instantiation of a template that has to be defined in all production rules. The developer must therefore extend all generalized LISA rules, redefine all production rules and instantiate a template in each of them. In this case lines of code (LOC) decrease, but number of (re)defined production rules remain the same. Using an aspect-oriented approach, environment propagation (e.g., `bucketBrigadeLeft` pattern) can be defined in an advice and further applied at appropriate production rules (i.e., join points in AspectLISA) that can be defined also in ancestor language component(s). Only production rules with additional semantics (i.e., additional to environment propagation), and production rules for assignment statement with some specific semantics for environment propagation have to be (re)defined in this case (i.e., two additional production rules and starting production). Only three production rules have

```
language RobotExpressions  extends RobotCalc {
 lexicon {
   extends    Operator  \=
   extends    Separator \;
  Identifier [A-Z]+
}
 attributes java.util.Hashtable *.inEnv, *.outEnv;
 attributes  int STMT.val;
 pointcut EnvPropagate *.* : * ::= .. ;  // all productions in the language
 advice bucketBrigadeLeft<inAtt, outAtt> {
  if (empty(RHS)) {
   LHS.outAtt = LHS.inAtt ;
 }
  else {
   first(RHS).inAtt = LHS.inAtt;
   RHS.inAtt = pred(RHS).outAtt;
   LHS.outAtt = last(RHS).outAtt;
 }
}
 apply bucketBrigadeLeft<inEnv, outEnv>  on EnvPropagate;
 pointcut ValEXPR RobotExpressions.ExprStart : EXPR ::= STMT ;
 pointcut ValSTMT RobotExpressions.Statement : STMT ::= .. E ..;
 advice ValPropagate<RSide> {
  LHS.val = RSide.val;
}
 apply ValPropagate<STMT>  on ValEXPR;
 apply ValPropagate<E>  on ValSTMT;
 rule  extends start {
  START ::= begin COMMANDS end  compute {
    COMMANDS.inEnv = newEnvironment();
  };
}
 rule Statement {
 STMT ::=  #Identifier \= E   compute {
   STMT.outEnv = putEnvironment(E.outEnv, #Identifier.value(),  new Integer(E.val));
 };
}
 rule  extends Expression {
 E ::= #Identifier  compute {
    E.val = ((Integer)getEnvironment(E.inEnv, #Identifier.value())).intValue();
 };
 method EnvironmentOperations {
  import java.util.Hashtable;
 Hashtable putEnvironment(Hashtable aEnv, String aName, Object aValue) {
   Hashtable Env = (Hashtable)aEnv.clone();
   Env.put(aName, aValue);
    return Env;
 }
 Hashtable newEnvironment() {
    return  new Hashtable();
 }
 Object getEnvironment(Hashtable aEnv, String aName) {
    return aEnv.get(aName);
  }
 }
}
```

Figure 3: Specifications for the `RobotExpressions` language

Table 1: Metrics for RobotExpressions example

LISA specs	ELOC	NOP	DP	RDE
Original	123	17	43	/
Templates	99	17	43	19.5 %
Aspects	67	17	25	45.5 %

to be redefined, instead of 17, when an aspect-oriented approach is used.

In Table 1, some metrics for our example are summarized (ELOC = Effective Lines of Code, NOP = Number Of Productions, DP = Defined Productions in AspectLISA, RDE = Reduced Developer Effort). As can be seen from the results, only 25 productions, instead of 43 defined productions in all language components have to be defined if one uses an aspect-oriented approach. This reduces the language developer effort by 45.5% in our example and reduces the possibilities to make mistakes when a language is reused or extended. Note, these evaluation metrics are for a toy Robot language. We believe that the benefits are more extensive for larger languages.

4 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.

4.1 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 language translator. 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). Additional information about ANTLR can be found on the ANTLR web page [1].

The Robot language described in Section 3 has been rewritten in ANTLR and partially provided in Figure 4. This simple example illustrates the ANTLR specification language with semantic rules defined in Java. From the language specification in Figure 4, ANTLR generates Java source code representing the scanner and parser for the Robot language. The Robot language follows the DSL implementation pattern using a pre-processor that serves as a compiler and application generator to perform a source-to-source transformation [20] (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.

4.2 AspectG: A Domain-Specific Aspect Language for Grammars

In our past work [32], 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 [25]. In this section, the syntactical extension of a language (as demonstrated with AspectLISA in the previous section) is not the primary focus. The motivation for AspectG is to assist in modularizing grammar adaptations that enable new tools (e.g., debugger and testing engine) for a language.

We developed a general framework called the DSL Testing Tool Studio (DTTS), which assists in debugging and testing a program written in a DSL [31]. 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 certain grammar productions. The manual addition of the same mapping code in the grammar productions 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 within the Robot DSL grammar, as shown in Figure 5. For example, lines

```
// 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);} ;
```

Figure 4: Specifications for the `Robot` language in ANTLR

12 to 17 represent 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 generates the mapping code statement used by the DTTS. These semantic actions are repeated in every terminal production of the Robot grammar. Note that lines 13 and 15 represent the original semantic actions of the Robot language.

```
...
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\"," + gplbeginline +
                     "," + gplendline + "));");}
18     |RIGHT {
19       dsllinenumber=dsllinenumber+1;
20       fileio.print(" x=x+1;");
21       gplbeginline=fileio.getLinenumber();
22       fileio.print(" time=time+1;");
23       gplendline=fileio.getLinenumber();
24       filemap.print("mapping.add(newMap(" + dsllinenumber + ",\"Robot.java\"," + gplbeginline +
                     "," + gplendline + "));");}
```

Figure 5: Specifications for the Robot language in ANTLR (with debug concern added)

The same mapping statements for the RIGHT command appear in lines 19 through 24. 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 mapping code is replicated across each production. Of course, because the mapping concern is not properly modularized, changing any part of the 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 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, which is written in Java. Join points in ANTLR are static points in the language specification 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 set in the Robot language*

Pointcuts in AspectG are defined using the reserved word **pointcut** and two keywords representing pointcut predicates (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 patterns (composed by terminals, non-terminals, and wild cards) and semantic rules. The patterns are used to identify the location of join points. They are passed into weaving functions to weave the semantic rules into the language grammar. Consider the following pointcut:

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

The pointcut called `productions` is defined with the wildcard `command.*` and matches `command` productions in the Robot grammar (e.g., `RIGHT` and `LEFT`). 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** or **after** a join point is necessary, as shown in the example below.

```
before():  productions() { dsllinenumber=dsllinenumber+1; }
after():  count_gpllinenumber() { gplendline=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., `gplendline=fileio.getLinenumber();`) after the parser matches a timer increment (i.e., `fileio.print ("time =time+1;");`).

4.2.1 Aspect Weaving at the DSL Grammar Level

We have used a program transformation system (e.g., DMS - the Design Maintenance System [6]) to weave crosscutting concerns into the actual language grammar. To modularize concerns in a grammar, DMS is used to weave the debugging concern directly into the grammar itself, rather than the ANTLR generated GPL source code. In order to transform language grammars written in ANTLR, a DMS language domain was created that is capable of parsing and transforming grammars specified in ANTLR. Using DMS, we created a parser for ANTLR and developed a series of lower level general transformations on the ANTLR representation. The debugging concern is weaved at an earlier stage in the grammar itself before the lexing and parsing. The changes that are encapsulated as aspects are propagated into the generated parser through the modified grammar productions. After weaving a grammar aspect and parsing the Robot DSL program, the new Robot 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 are recorded in the grammar).

We have developed a domain-specific aspect language called AspectG, which weaves the aspects into grammars by generating low-level transformation rules written in DMS that use pattern matching and rewrite specifications on the AST of a source program (in this case, the source is actually an ANTLR grammar file). This approach has the side benefit of language independence. It does not matter which GPL serves as the generated target and also does not matter which GPL serves as the embedded semantic code. The DMS ANTLR domain is capable of parsing the grammar and weaving the aspects for a large set of programming languages that are pre-defined in DMS (e.g., Ada, C, C++, C#, COBOL, FORTRAN, HTML, Java, PHP, SQL, and XML).

In order to weave the aspects into DSL grammars, the first step is to construct a parser for ANTLR. As shown in Figure 4, the Robot DSL syntax is specificed in ANTLR notation where the semantic actions are written in Java by separating the action code within each production with curly brackets. This is an example of an embedded DSL where a host language (e.g., ANTLR) has another language (e.g., Java) embedded within it. The embedded Java within ANTLR presents a challenge because two different syntactic constructs (i.e., ANTLR and Java) must be parsed using any one particular parser. A simplified solution would be to include all the tokens and productions from both language domains to form a combined grammar and then generate the parser using the DMS parser generator. However, this approach does not make use of the existing DMS Java grammar or parser. A better approach that we adopted reuses the existing DMS Java tools and separates the ANTLR grammar productions from the Java grammar productions, but still parses the input source containing tokens from both languages. The ANTLR parser that we created in DMS has the ability to switch between different parse modes based on whether an ANTLR production has the focus, or if control has passed to a semantic action written in Java.

A grammar aspect can be specified in AspectG, which generates DMS transformation rules to configure the low-level transformation functions. Before the grammar is even processed by the ANTLR parser, it is first pre-processed by AspectG using DMS in order to weave the grammar aspect into the original grammar productions. The transformed grammar is then submitted to ANTLR in order to generate the parser and lexer for a specific GPL. The contribution of this approach is the transformation of the grammar itself, rather than the generated parser code. The specification of the debug mapping is modularized in a single place - the AspectG specification.

4.2.2 AspectG Implementation

Unlike AspectLISA's compiler approach, AspectG uses a program transformation system 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 [32]. 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 (i.e., before the first statement of one semantic action code segment); after a semantic action (i.e., after the last statement of one semantic action code segment); before a specific statement that is inside a semantic action; and, after a specific statement that is inside a semantic action. These join 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, 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.

4.2.3 AspectG Illustrative Example

This sub-section illustrates the process of using AspectG to weave a debugging aspect into the Robot language grammar. The mapping between the DSL (in this case, ANTLR) and the generated GPL (in this case, Java) is represented by a line number counter that keeps track of which DSL line number corresponds to the current line of GPL code being debugged. Figure 6 shows the DSL line number counter aspect in AspectG notation. The DSL line number counter update statement (`dsllinenumber=dsllinenumber+1;`) must be inserted after each Robot language statement [31]. The pointcut called `productions` (shown earlier) matches the production rules of the Robot language grammar within any instance production whose name begins with `command`. It specifies code (`dsllinenumber=dsllinenumber+1;`) to run at a join point matched by the pointcut `productions` to update the DSL line number counter every time there is a DSL statement defined in the `command` production rule set.

```
...
aspect dsllinenumber (

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

   after(): productions() {dsllinenumber=dsllinenumber+1;}
...
```

Figure 6: DSL Line Number Counter Aspect in AspectG Notation

From the high-level aspects specified in AspectG, a series of low-level transformation rules are generated that are executed on the DMS transformation engine in order to weave changes into an ANTLR grammar. Specifically, rules are generated in the Rule Specification Language (RSL) of DMS. A series of template RSL rules have been designed that correspond to the four types of weaving that may occur within AspectG (i.e., weaving before or after a production, and weaving before or after a semantic action). Figure 7 shows the low-level RSL specification generated from the `productions` pointcut shown earlier, which is used to weave a debugging DSL line number into the the Robot grammar specified in ANTLR. The weaving process will insert the statements that map the DSL line number statement into the appropriate places of the Robot grammar. The first line of the rule establishes the default base language domain to which the transformations are applied (in this case, ANTLR). In this example, the pattern `after_advice` (line 3) is an external library function that was written to perform the actual process of sub-typing, naming, and weaving. The rule `print_after_tree` on line 8 triggers the transformation on the Robot grammar by invoking the specified external pattern. Notice that there is a condition associated with this rule (line 10), which describes a constraint stating that the rule should be applied only to those join points where a transformation has not occurred already. This is because the DMS re-write engine will continue to apply all sets of rules until no rules can be fired. It is possible to have an infinite set of rewrites if the transformation results are monotonically increasing (i.e., when one stage of transformation continuously introduces new trees that can also be the source of further pattern matches). After applying this rule to the Robot language grammar, a new semantic segment will be generated with the line number update statement inserted at the end of every production in the `command` set (i.e., `RIGHT, LEFT, DOWN UP`). The essence of the transformation can be seen in line 9 of the transformation, which states that a `java_seq` is rewritten ("->") with a parameterized call to the `after_advice` external function.

After the DSL line number counter aspect is weaved into the Robot language grammar, during the parsing phase of this modified grammar the inserted line number counter of the advice executes throughout the `command` production propagation. This line number information helps keep track of each Robot statement and is also passed to the source code mapping component for Robot language debugger generation [31].

5 Lessons Learned and Related Work

This section summarizes some of the lessons that we learned from designing AspectLISA and AspectG. The section also compares our two DSALs to related work in the area of aspects for language specification.

```
default base domain Antlr.

external pattern after_advice(af_adv:statement_string,
                       lefthandside: IDENTIFIER,
                       orig_stmt:semantic):
                       semantic = 'after_advice' in domain Antlr.

rule print_after_tree(java_seq: semantic): semantic -> semantic
      = " \java_seq " -> "\after_advice\(\aft_advice\(\) \, \lefths\(\)\, \java_seq\)"
     if java_seq ~= "\:semantic \after_advice\(\aft_advice\(\) \, \lefths\(\)\, \java_seq\)".

 pattern aft_advice(): statement_string = "dsllinenumber=dsllinenumber+1".

 pattern lefths(): IDENTIFIER ="command".

 public ruleset a =  print_after_tree .
```

Figure 7: Low-level DMS Transformation Rule Generated from AspectG

5.1 Lessons Learned

Some DSLs are executable (e.g., the Robot language introduced in this paper), but many DSLs are non-executable [20]. The syntax specification formalism BNF is an example of a DSL with a purely declarative character that can also act as an input language for a parser generator. Moreover, BNF is not primarily meant to be executable but nevertheless useful for application generation. This is the main reason that we used static join points in both AspectLISA and AspectG. Because concerns that crosscut language specifications are mainly of a semantic nature addressing behavior, we decided not to support syntactic structural changes. However, syntax extensions are effectively covered by inheritance in AspectLISA. This decision helped us to identify what advice and join points should be in each of our two languages. Advice in AspectLISA and AspectG are additional semantic rules that have to be attached to particular productions. Hence, join points are syntax production rules and the designed pointcut language should be able to match arbitrary syntax production. Because AspectLISA uses a more complicated specification structure due to inheritance, the pointcut language is richer with respect to wildcards and from selection on hierarchy of syntax productions. However, AspectG uses ANTLR, which is a syntax directed translator where the order of semantic rules are important (i.e., defined by language designer). This has the consequence that new semantic rules specified in advice have to be weaved at join points that are between semantic rules of particular syntax production. Hence, the pointcut language in AspectG consists of predicate `match` to locate an appropriate point in the

Table 2: Summarized design decisions for AspectLISA and AspectG

Q	AspectLISA	AspectG
1	Production rules	Semantic rules with particular production rules
2	Static Join Points	Static Join Points
3	Coarse-grained (rule and production level only)	Fine-grained (within, match, before, after)
4	Matching rules on hierarchy of BNF productions (two wildcards: *, ..)	Matching rules on BNF production and semantic rules (one wildcard: *)
5	Semantic rules	Semantic rules
6	Refinement/extension on behavior	Testing tools generation
7	Advice are parametrized	No parametrization

language specification. Moreover, in AspectG it must be specified whether new semantic rules are weaved before or after the matched location. This is the reason that a full pointcut pattern language in AspectG is more complicated than in AspectLISA which uses attribute grammars (AG). The order of semantic rules is not important in AG because the order is computed by the AG evaluator at compiler generation time. Hence, only appropriate syntax production need to be specified by the AspectLISA pointcut language. This also simplifies the implementation of weaving. It can be said that AspectLISA uses coarse granularity for join points and AspectG uses fine granularity. A primary goal of LISA was to achieve improved reusability of language specifications. This was achieved by inheritance and templates in AGs. In AspectLISA, advice are also parametrized. Enhanced reusability is also achieved by separating advice from the specific application of advice (i.e., using the `apply` construct in AspectLISA).

Table 2 lists the questions that were posed in Section 2, and the specific way each question is addressed in AspectLISA and AspectG. As can be seen from the results presented in this table, DSALs from the same domain can be designed and implemented quite differently. Our main lessons learned from designing these two DSALs are as follows:

- AOP can be beneficial in language specifications and other specification languages. DSAL design depends on the component language, which can be a GPL or DSL.
- Similar problem domains might require a different design approach, including a different JPM.
- Decision to use static or dynamic join points in DSALs depends also on the executability level of the component language (DSL). Many DSLs are not meant to be executable.
- The structure and behavior of the DSL component language must be understood well. Advice and join

points belong to the component language.

- Different problems might be solved in similar domains. AspectLISA's intent is to refine and extend a target language's behavior and AspectG's motivation is to generate testing tools for target languages.
- Weaving of a fine-grained pointcut language (e.g., AspectG using DMS) is harder to implement than a coarse-grained pointcut language (e.g., AspectLISA extends LISA compiler).
- For weaver implementation, existing language transformation tools (e.g, DMS, Stratego, ASF+SDF, LISA) can be used.

5.2 Related work

AspectASF [15] 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 (i.e., after a successful match of left-hand side) and exiting an equation (i.e., 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 [11]. The JastAdd system is a class weaver: it reads all the JastAdd modules (i.e., 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 join points while AspectLISA and AspectG use explicit join points described by pointcuts. Moreover, JastAdd does not enable inheritance on advice and pointcuts as AspectLISA does.

6 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 from the same domain. Future work includes new pointcut predicates that assist in specifying the control flow within a grammar, which 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, 2007.

[2] AspectC. http://www.aspectc.org/, 2007.

[3] AspectC#. http://www.castleproject.org/index.php/aspectsharp, 2007.

[4] AspectJ. http://eclipse.org/aspectj/, 2007.

[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] 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.

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

[12] 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.

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

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

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

[16] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[17] LISA web page. http://labraj.uni-mb.si/lisa, 2007.

[18] 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.

[19] M. Mernik and M. Lenič and E. Avdičaušević and V. Žumer. The Template and Multiple Inheritance Approach into Attribute Grammars. *IEEE and ACM International Conference on Computer Languages, ICCL98, Chicago*, pages 102 – 110, 1998.

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

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

[22] 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.

[23] 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.

[24] D. Rebernak, M. Mernik, P. R. Henriques, and M. J. V. Pereira. AspectLISA: An aspect-oriented compiler construction system based on attribute grammars. *Electr. Notes Theor. Comput. Sci.*, 164(2):37–53, 2006.

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

[26] 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.

[27] 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.

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

[29] 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.

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

[31] H. Wu, J. Gray, and M. Mernik. Grammar-driven generation of domain-specific language testing tools. *Software: Practice and Experience*, accepted, 2007.

[32] 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.