AspectLISA: an aspect-oriented compiler construction system based on attribute grammars

Damijan Rebernak, Marjan Mernik [1,2]

University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova ul. 17, 2000 Maribor, Slovenia

Pedro Rangel Henriques [3]

University of Minho
Department of Computer Science
Campus de Gualtar
4710 - 057 Braga, Portugal

Maria João Varanda Pereira [4]

Polytechnic Institute of Bragança
Campus de Sta. Apolónia
Apartado 134 - 5301-857, Bragança, Portugal

Abstract

The use of object-oriented techniques and concepts, like encapsulation and inheritance, greatly improves language specifications towards better modularity, reusability and extensibility. Additional improvements can be achieved with aspect-oriented techniques since semantic aspects also crosscut many language constructs. Indeed, aspect-oriented constructs have been already added to some language specifications. The LISA compiler construction system follows an object-oriented approach and has already implemented mechanisms for inheritance, modularity and extensibility. Adding aspects to LISA will lead to more reusable language specifications. In the paper, aspect-oriented attribute grammars are introduced, and the underlying ideas are incorporated into AspectLISA, an aspect-oriented compiler generator based on attribute grammars.

Key words: Attribute grammars, aspect-oriented programming, compiler generators.

©*2006 Published by Elsevier Science B. V.*

1 Introduction

The challenge in programming language definition is to support modularity and abstraction in a manner that supports reusability and extensibility. A language designer often wants to include new language features incrementally as the programming language evolves. This is especially true in developing domain-specific languages (DSLs) which change more frequently than general-purpose programming languages [17]. Ideally, a language designer would like to build a language simply by reusing different language definition modules (e.g., language components), such as modules for expressions, declarations, etc., regardless of the different formal methods that may be used to specify such language components. This approach is common in component-based programming [24] where components can be simply plug-ins.

This cannot be done now, even if we restrict ourselves to just one of the formal methods (abstract state machines, action semantics, algebraic specifications, attribute grammars, denotational semantics, operational semantics, two-level grammars, etc. [23]) since different compiler-compilers (automatic compiler generation systems) use different and incompatible specification languages (e.g., despite the fact that Eli [6] and FNC-2 [9] both rely on attribute grammars one can not exchange language definition modules written in the other system). Moreover, the same is usually true even in the case of the same specification language since syntax entities (e.g., non-terminals and terminals) and semantic entities (e.g., attributes and semantic rules in the case of attribute grammars) are not constituents of the hidden part of the module, nor are the parameters of language definition modules. For example, when importing a module for expressions some non-terminals may clash with existing non-hidden non-terminals producing undesirable effects. Such a module can be parameterized using non-terminals as parameters to solve renaming problems. However, modules with dozens of parameters are hard to use.

Compared to modern programming languages, such as object-oriented or functional languages, language specifications of the 1980's and early 1990's were far less advanced, specifically concerning provisions for abstraction, modularization, extensibility and reusability. Recently, concepts from general programming languages have been successfully incorporated into language specifications. Among them, object-oriented techniques are one of the most successful. Indeed, this had several benefits on language specifications. To fully achieve modularity, extensibility and reusability these techniques need to be combined with aspect-oriented techniques because semantic aspects also crosscut many language constructs [19]. These observations have been taken into account in extending the LISA specification language [18] with aspect-oriented features. The paper presents AspectLISA,

[1] This work is sponsored by bilateral project BI–PT/04–06–008 "Grammar Based Systems" between Slovenia and Portugal.
[2] Email: {damijan.rebernak, marjan.mernik}@uni-mb.si
[3] Email: prh@di.uminho.pt
[4] Email: mjoao@ipb.pt

which is an aspect-oriented compiler generator based on attribute grammars.

The paper is organized as follows. A brief introduction to aspect-oriented programming is given in section 2 followed by related work presented in section 3. The main part of the paper constitutes section 4 where AspectLISA is discussed. A small case study illustrating ideas is given in section 5. The concluding comments are mentioned in section 6.

2 Aspect-oriented programming

The major abstraction technique in software engineering is to divide the system into functional components in such manner that changes to a particular component do not propagate through the entire system [5,22]. However, some issues, called aspects, are system wide and cannot be put into a single functional component. As examples, failure handling, persistence, communication, coordination, memory management, are aspects of a system behavior that tend to crosscut groups of functional components. As a consequence, functional components are tangled with aspect code. This tangling problem makes functional components less reusable and difficult to develop, understand, and evolve. A solution is provided by aspect-oriented programming (AOP) [12] which is a programming technique for modularizing concerns that crosscut the basic functionality of programs. In AOP, aspect languages are used to describe properties which crosscut basic functionality in a clean and a modular way. Despite that the main part of AOP research is devoted to general-purpose languages [11,16] similar problems exists in domain-specific languages. For example, in language specifications modularization is usually based on language syntax constructs, whereas the modularization based on different aspects (e.g. name analysis, type checking, code generation, etc.) would be more beneficial. To overcome this problem aspect-oriented techniques can be used.

In order to achieve the desired properties of the system, we need an aspect weaver that combines the component and the aspect language by weaving advice at appropriate join points and may involve merging components, modifying and optimizing them.

3 Aspects in language development

Aspect-oriented programming is a very promising approach and has been successfully used in tools for language definition and implementation [4,7,10,13,15,26,27]. In this context, aspects have been used for many different tasks (e.g., in [26] an extension for weaving debugging information into DSL specifications is reported). In the rest of this section we describe in more detail some of the more relevant contributions in the field, *using aspects in language specification or implementation*.

3.1 JastAdd

JastAdd [7] is a Java-based system for compiler construction. JastAdd is centered around object-oriented representation of the abstract syntax tree (AST). Nonterminals act as abstract super classes and productions act as specialized concrete subclasses that specify the syntactic structure, attributes and semantic rules. All these elements can be inherited, specialized, and overridden in subclasses. The idea of aspect-orientation in JastAdd is to define each aspect of the language in a separate class and then weave them together at appropriate places. The JastAdd system is a class weaver: it reads all the JastAdd modules and weaves the fields and methods into the appropriate classes during the generation of the AST classes. With separation of different language aspects among different classes, developers have the possibility to use all features of Java to specify aspects. In the following example, taken from [7], two different aspects are described in separate classes. The first one (`typechecker.jadd`) performs type checking for expressions and computes the boolean field `typeError`. The `unparser.jadd` (second example) implements an unparser which makes use of the field `typeError` to report type-checking errors.

```
// typechecker.jadd
class Exp {
 abstract void typeCheck(String expectedType);
}
class Add {
   boolean typeError;
 void typeCheck(String expectedType) {
   getExp1().typeCheck("int");
    getExp2().typeCheck("int");
    typeError= expectedType != "int";
  }
}
```

```
// unparser.jadd
import Display;
class Stmt {
  abstract void unparse (Display d);
}
class Add {
  void unparse (Display d) {
    ...
    if (typeError)
      d.showError("type mismatch");
    }
}
```

Every field, method, import declaration is weaved to all generated AST classes, as can be seen in following example.

```
 class ASTAdd extends ASTExp {
   // Access interface
  ASTExp getExp1() { ... }
  ASTExp getExp2() { ...}
   // From typechecker.jadd
  boolean typeError;
  void typeCheck(String expectedType) {
    getExp1().typeCheck("int");
    getExp2().typeCheck("int");
    typeError = expectedType != "int";
```

```
  }
  // From unparser.jadd
  void unparse(Display d) {
    ...
    if (typeError)
      d.showError("type mismatch");
    ...
  }
}
```

As can be seen in the example above, this approach does not follow the conventional AOP join point model (JPM) where join points are specified using a pointcut pattern language. However, it can be seen as inter-type declarations in AspectJ [11] where join points are all non-anonymous types in the program and pointcuts are the names of classes or interfaces.

3.2 AspectG

To generate an additional language-based tool (e.g., debugger) new specifications need to be added in several places in language specifications [8]. These new additions can be seen as aspects (e.g., debugging aspects). It was observed [26] that such aspects crosscut basic language specifications. Hence, the aspect-oriented language AspectG [2] was created for modular implementation of crosscutting concerns in the ANTLR language definition [1]. Since ANTLR belongs to syntax directed translations (semantic rules are not declaratively specified and order of semantic rules is important) AspectG uses the following model:

- join points are static points in language specifications where additional aspects can be weaved,
- pointcuts specify join points and include not only the syntax level of the grammar but also the semantics associated with a particular syntax (see within and match constructs in the example below),
- advice are similar to AspectJ notion (before and after) and brings together a pointcut and a body of code.

An example of pointcut and advice in AspectG is shown below.

```
...
command
 :( RIGHT
   {
    fileio.print("//move right");
    fileio.print("x=x+1;");
    fileio.print("time=time+1;");
  }
...

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

after(): count_gpllinenumber()
 {gplbeginline=fileio.getLinenumber();
  gplendline=fileio.getLinenumber();}
```

The result of weaving is:

```
...
command
 :( RIGHT
   {
    fileio.print("//move right");
    fileio.print("x=x+1;");
    gplbeginline=fileio.getLinenumber();
    gplendline=fileio.getLinenumber();
    fileio.print("time=time+1;");
  }
...
```

3.3 AspectASF

AspectASF [13] is a simple aspect language for language specifications written in the ASF+SDF [25] 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 very 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 succesfull match of left-hand side) and exiting an equation (just before returning the right-hand side). Examples (all examples in this section are taken from [13]) of pointcuts in AspectASF language are:

`[_]`	matches all equations
`[_] eval(_, _)`	matches all equations with outermost symbol eval
`[_] eval(_, Env)`	matches all equations with 2nd arg an Env variable
`[int*] _ or [real*] _`	matches all equations with label int.. or real..

Advice code specify additional equations which are written in the ASF formalism. There are two types of advice: after entering an equation (concatenating equations to the beginning of the list of equations that is matched by the pointcut) and before exiting an equation (concatenating equations to the end of the list of equations that is matched by the pointcut). An example of AspectASF is shown below.

```
[1] Env'  := evs(Stat, Env),
    Env'' := evs(Stat*, Env')
    ============================
    evs(Stat ; Stat*, Env) = Env''

pointcut statementStep: entering [_] evs(Stat ; Stat*, Env)
after: statementStep tide-step(get-location(Stat))
```

After weaving takes place the aspects are weaved into the original language specifications. In other words, additional equations are appended to appropriate places.

```
[1] tide-step(get-location(Stat)),
    Env'  := evs(Stat, Env),
    Env'' := evs(Stat*, Env')
    ============================
    evs(Stat ; Stat*, Env) = Env''
```

4 AspectLISA

4.1 Introduction to LISA

In the LISA project [18,20], one of the main goals was to enable incremental language development. It was soon recognized that inheritance can be very helpful since it is a language mechanism that allows new definitions to be based on the existing ones. A new specification can inherit the properties of its ancestors, and may introduce new properties that extend, modify or defeat its inherited properties. In object-oriented languages the properties that consist of instance variables and methods are subject to modification. The corresponding properties in language definitions based on attribute grammars are:

- lexical regular definitions,
- attribute definitions,
- rules which are generalized syntax rules that encapsulate semantic rules, and
- operations on semantic domains.

Therefore, regular definitions, production rules, attributes, semantic rules and operations on semantic domains can be inherited, specialized or overridden from ancestor specifications. In this approach the attribute grammar as a whole is subject to inheritance employing the "Attribute grammar = Class" paradigm [21].

A very simple language for moving a robot can illustrate our incremental language development approach [20]. The language for robot movement is defined in Fig. 1. The robot can move in different directions and the task is to compute its final position. Over time, the language is extended with new features. For example, we would like to know when the robot will reach the final position. The new language (RobotTime) is specified as an extension of the Robot language (Fig. 2). This is a good example of how different aspects can be modularized in our approach. In the Robot language just the semantic rules for robot movement have been described, while the RobotTime language contains just the semantic rules for time calculation. The RobotTime language inherits regular definitions, syntax constructs and semantic rules from the Robot language and adds new semantic rules for time calculation. Note that the same effect is obtained by implicit pointcuts in aspect-oriented systems like JastAdd [7] (see section 3).

As already mentioned, object-oriented techniques and concepts need to be combined with aspect-oriented techniques to achieve better modularity, extensibility and reusability. This issue is further described in the following sections.

4.2 Aspect-oriented Attribute Grammars

Aspect-oriented attribute grammar (AspectAG) is an attribute grammar [14] extended with pointcut and advice specifications [12], $AspectAG = (G, A, R, Pc, Ad)$. Context-free grammar $G = (N, T, S, P)$, set of attributes A, and set of semantic rules R have the same standard meaning of attribute grammars, as for example described in [18].

```
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 {
    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); };
  }
}
```

Fig. 1. Robot Language using LISA

Pointcuts Pc is a set of **pointcut productions**, $Pc = \{pc_1, ..., pc_m\}$, where pointcut production pc_i, $1 \leq i \leq m$, has the following form:

$$pc_i < X_1, ..., X_r > \ : \ LHS \ \rightarrow \ RHS$$

In pointcut production pc_i special wildcard symbols (.., $*$) can be used. Wildcard symbol '$*$' denotes a symbol or some part of its name and can be used in the LHS and RHS. Wildcard symbol '..' denotes zero or more symbols and can be used only in the RHS. Symbols $X_i, 1 \leq i \leq r$, are symbols from LHS and RHS and denote the public interface for advice. A pointcut production $pc_i < X_1, ..., X_r >$: $LHS \rightarrow RHS$, selects a production $p : X_0 \rightarrow X_1...X_n \in P$ if X_0 matches LHS and X_1 ... X_n match RHS. Let Pm_i denote the set of productions selected by pointcut production pc_i, $Pm_i = \{p_i | p_i \in P \ and \ p_i \ is \ matched \ by \ pc_i\}$. Matched productions Pm selected by pointcuts Pc is then defined as $Pm = \bigcup_{i=1..m} Pm_i$, $Pm \subseteq P$. To match productions Pm, additional semantic rules specified in advice Ad are attached.

Ad is a set of **advice**, $Ad = \{ad_1, ..., ad_l\}$, where advice ad_k, $1 \leq k \leq l$, has

```
language RobotTime extends Robot {

  attributes double *.time;

  rule extends start {
   compute {
       // initial position is inherited
      START.time = COMMANDS.time; }
  }

  rule extends moves {
   COMMANDS ::= COMMAND COMMANDS compute {
     // total time is sum of times spent in sub-commands
      COMMANDS[0].time = COMMAND.time + COMMANDS[1].time; }
   | epsilon compute {
     COMMANDS.time = 0; };
  }

  rule extends move { // each command spent 1 time step
   COMMAND ::= left compute {
     COMMAND.time = 1; };
   COMMAND ::= right compute {
     COMMAND.time = 1; };
   COMMAND ::= up compute {
     COMMAND.time = 1; };
   COMMAND ::= down compute {
     COMMAND.time = 1; };
  }
}
```

Fig. 2. RobotTime Language using LISA

the following form:

$$ad_k < S_1, ..., S_r > \ on\ pc_i\ \{Rs_k\}$$

Semantic rules Rs_k has the following form:

$$Rs_k = \{S_j.a = f(y_1, ..., y_k) | a \in A(S_j), y_i \in (A(S_1) \cup ... \cup A(S_r)),\ 1 \leq i \leq k\}$$

Defi ning attributes attached to symbols $S_j, 1 \leq j \leq r$, are defi ned by semantic rules in Rs_k. Advice ad_k is applied on pointcut pc_i, which match productions Pm_i. For each match production $p_i \in Pm_i$, the actual set of semantic rules Ra_{ki} is obtained by replacing formal symbols S_j (specifi ed in ad_k) by actual symbols X_j (specifi ed in pc_i) in Rs_k. The set of semantic rules Ra obtained from advice Ad and pointcuts Pc is defi ned as $Ra = \bigcup_{k=1..l, i=1..m} Ra_{ki}$ and needs to be merged with ordinary semantic rules Rp_i, to obtain well defi ned attribute grammar $AG = (G, A, R)$ in the following manner: $Rp'_i = Rp_i \cup (\bigcup_{k=1..l} Ra_{ki})$, $R' = \bigcup_{i=1..n} Rp'_i$. Note that $(G, A, R, Pc, Ad) = (G, A, R')$. Therefore, an aspect-oriented attribute grammar is an attribute grammar where some semantic rules are not attached explicitly to production rules but implicitly as advice into productions selected by pointcuts. When semantic rules are merged, only one semantic rule for each defi ning attribute must exist, otherwise the attribute grammar is not well defi ned [14]. The following illustrates a simple example:

```
Ordinary attribute grammar specifications:
p0:  A → B C {A.x = B.x + C.x; B.y = 0; C.y = 1;}   // Rp0
p1:  B → a B {B0.x = B1.x; B1.y = B0.y + 1;}        // Rp1
p2:  B → ϵ {B.x = B.y;}                             // Rp2
p3:  C → c {C.x = C.y + 2;}                         // Rp3
Pointcuts:
pc1 <B> :  B → ..                                   // matches p1 and p2
pc2 <A, B> :  A → B *                               // matches p0
Advice:
ad1 <X> on pc1 {X.z=1;}                             // Ra11 = {B.z=1;}
                                                    // Ra12 = {B.z=1;}
ad2 <Y, X> on pc2 {Y.w = X.z;}                      // Ra20 = {A.w = B.z;}
Final semantic rules:
Rp'0  =  Rp0 ∪ Ra20  =  {A.x = B.x + C.x; B.y = 0; C.y = 1; A.w = B.z;}
Rp'1  =  Rp1 ∪ Ra11  =  {B0.x = B1.x; B1.y = B0.y + 1; B.z = 1;}
Rp'2  =  Rp2 ∪ Ra12  =  {B.x = B.y; B.z = 1;}
Rp'3  =  Rp3         =  {C.x = C.y + 2;}
```

4.3 AspectLISA constructs

As seen from Fig. 1 and Fig. 2 LISA enables good modularity and extensibility of attribute grammar specifications. However, there are still situations when new semantic aspects crosscut basic modular structure. In other words, some semantic rules need to be repeated in different productions (e.g., semantic rule `COMMAND.time = 1;` which has to be repeated several times in generalized production `move` of `RobotTime` language). To avoid this unpleasant situation, an aspect-oriented attribute grammar, as specified in subsection4.2, has been incorporated into LISA language specifications. This extension is called AspectLISA. Join points in AspectLISA are static points in language specifications where additional semantic rules can be attached. These points can be syntactic production rules or generalised LISA rules. The production matching takes place on productions which are members of generalized LISA rules. One pointcut can match productions in different languages over the entire hierarchy of languages. For each pointcut we can define several advice which are parameterized semantic rules written as native Java assignment statements. In AOP, several different approaches of applying aspects to pointcuts exists, like before, after and around [11]. In AspectLISA there is only one way to apply advice on a specific pointcut, since attribute grammars are declarative and the order of equations in semantic rules is not important. Therefore, applying advice before/after a join point is not applicable.

The AspectLISA specification language, including apect-oriented features, pointcuts and advice, has the following parts (note how pointcuts and advice defined in section 4.2 are written in the LISA specification language):

```
language L1 [extends L2, ..., LN] {
    lexicon {
        [[Q] overrides | [Q] extends] R regular expr.
            ⋮
    }
    attributes type At1, ..., AtM
        ⋮
    pointcut P< [S1, ..., Sr] > L.Y : LhsP ::= RhsP ;
        ⋮
    advice [[B] extends | [B] overrides] A< [T1, ..., Tr] > on P {
        semantic functions
    }
        ⋮
    rule [[Y] extends | [Y] overrides] Z {
        X ::= X11 X12 ... X1p compute {
                semantic functions }
            ⋮
        |
            Xr1 Xr2 ... Xrt compute {
                semantic functions }
        ;
    }
        ⋮
    method [[N] overrides | [N] extends] M {
        operations on semantic domains
    }
    ⋮
}
```

Symbols used in formal AspectLISA specifications above have following meaning:

- L – language name,
- Q and R – regular expression name,
- At – attribute name,
- P – pointcut name,
- S – actual symbol,
- $LhsP$ and $RhsP$ – left and right-hand side of pointcut production,
- A – advice name,
- T – formal symbol,
- X – grammar symbol,
- Y and Z – grammar rules,
- N and M – method name.

This section focuses only on the new aspect-oriented features of the LISA specification language which are pointcuts and advice.

Pointcuts are defined using the reserved word **`pointcut`**. Each pointcut has a unique name and a list of actual parameters (terminals and non-terminals used

in semantic functions of advice). As we already mentioned, join points are static points in language specifications where advice can be applied. In the pointcut definition one can use two wildcards. The wildcard '..' matches zero or more terminal or non-terminal symbols and can be used only to specify right-hand side matching rules. The wildcard '∗' is used to match parts or whole literal representing a symbol. To illustrate the AspectLISA pointcut model, we present some examples of pointcut specifications, defined over the Robot languages (Fig.1, Fig. 2).

`*.* : * ::= ..`	*matches any production in any rule in all languages across current languge hieararchy*
`RobotTime.m* : * ::= ..`	*matches any production in all rules which start with* `m` *in* `RobotTime` *language*
`*.* : COMM* ::= .. *D`	*matches all productions in any rule whose left-hand side symbol satisfy pattern "COMM*" and the right-hand side's last symbol ends with* `D`
`Robot.move : COMMAND ::= left`	*matches only a production* `COMMAND ::= left` *in the rule* `move` *of* `Robot` *language*

Advice in AspectLISA are additional semantics that can be appended at a specific join point. In order to increase reusability, advice are parameterized. Parameters can be terminal or non-terminal symbols and are evaluated at weaving time. Advice are defined using the reserved word **`advice`** and contain information about the pointcut where advice will appear. Below is an example of advice; more examples of advice and pointcuts are provided in section 5.

```
pointcut SimpleCommand<COMMAND> *.move : COMMAND ::= *;
advice SetTime<C> on SimpleCommand { C.time = 1;}
```

The result of weaving advice `SetTime` on pointcut `SimpleCommand` in the RobotTime language is an additional semantic rule `COMMAND.time = 1;` in all productions of rule `Robot.move`. The notation is much simpler as in Fig. 2. The new aspect of the language, namely time calculation, is described at one place (advice) and is not repeated in several productions.

4.4 AspectLISA inheritance

The AspectLISA specification language is an extension of LISA with two new mechanisms (pointcuts and advice). Obviously, pointcuts and advice can also be inherited from ancestor specifications. Formal definition of multiple attribute grammar inheritance as described in [18] needs to be adopted. Due to lack of space in this paper only the formal definition of inheritance of pointcuts and advice are given. For theoretical background and further details readers are referred to [18].

Properties of aspect-oriented attribute grammars consist of lexical regular definitions, attribute definitions, rules which are generalized syntax rules that encapsulate semantic rules, **pointcuts**, **advice** and methods on semantic domains.

$$Property = Regdef Name + AttributeName + RuleName + PointcutName + AdviceName + MethodName$$

For each pointcut pc in the language l, a $Pointcuts(l)(pc)$ is a finite set of matching productions P, that match to the pointcut pc, over the hierarchy of language l.

$$Pointcuts : Language \rightarrow PointcutName \rightarrow MatchingProductionRules$$
$$Pointcuts(l)(pc) = \{p_i \mid p_i \in P, p_i : X_{i0} \rightarrow X_{i1}X_{i2}...X_{in}, match(p_i, pc)\}$$

For each advice ad attached to pointcut pc in the language l, $Advice(l)(ad)(pc)$ is a finite set ($ProdSem$) of pairs (p, R_p), where p is a production and R_p is a union of finite set of semantic rules associated with the production p, and semantic rules ($definedR_p$) defined by advice ad, where formal symbols of advice are replaced by actual symbols defined in pointcut pc.

$$Advice : Language \rightarrow AdviceName \rightarrow PointcutName \rightarrow ProdSem$$
$$Advice(l)(ad)(pc) = \{(p, R_p) | p \in Pointcuts(l)(pc),$$
$$p : X_0 \rightarrow X_1X_2...X_n,$$
$$R_p = \{X_i.a = f(X_{0.b}, \ldots, X_{j.c}) |\ X_i.a \in DefAttr(p)\} \cup definedR_p(ad, pc)\}$$

Multiple aspect-oriented attribute grammar inheritance is defined as follows.

Let $AspectAG_1$, $AspectAG_2$, ..., $AspectAG_m$ be aspect-oriented attribute grammars formally defined as:

$$AspectAG_1 = (G_1, A_1, R_1, Pc_1, Ad_1),$$
$$AspectAG_2 = (G_2, A_2, R_2, Pc_2, Ad_2),$$
$$\vdots$$
$$AspectAG_m = (G_m, A_m, R_m, Pc_m, Ad_m), \text{ then}$$

$$AspectAG = AspectAG_2 \oplus \ldots \oplus AspectAG_m \oplus \triangle AspectAG_1 ,$$
where $AspectAG_1$, which inherits from $AspectAG_2, \ldots, AspectAG_m$, is defined as:

$$AspectAG = (G, A, R, Pc, Ad), \text{ where}$$
$$G = G_2 \oplus \ldots \oplus G_m \oplus \triangle G_1,$$
$$A = A_1 \ominus \ldots \ominus A_m,$$
$$R = R_1 \otimes \ldots \otimes R_m,$$
$$Pc = Pc_1 \ominus \ldots \ominus Pc_m,$$
$$Ad = Ad_1 \otimes \ldots \otimes Ad_m .$$

Therefore, inheritance on pointcuts is defined in a similar manner as for attributes [18]. Pointcuts as well as attributes cannot be extended, but can be inherited from ancestor attribute grammars. On the other hand, it is possible that some pointcut is redefined in current specifications which override pointcut specified in ancestor specifications. Inheritance on advice is defined in a similar manner as for semantic rules R [18]. This should not be surprising, because advice are just additional semantic rules which need to be weaved at appropriate join points.

4.5 AspectLISA novelty

AspectLISA is first specification language based on attribute grammars that use an explict pointcut model. Note that in JastAdd the pointcut model is implicit. The poincut model in AspectG is more complicated because syntax as well as semantic level are involved in the specification. This is due to using syntax directed translation instead of attribute grammars. AspectASF uses very simple pattern matching language where only labels and left-hand sides of equations written in ASF formalism can be matched. None of the existing systems enable inheritance on advice and pointcuts. Moreover, advice in AspectLISA are parameterized on grammar symbols and hence more reusable.

5 Using AspectLISA

Each LISA language specification is also a regular AspectLISA specification. In section 3.3 the RobotTime language has been specified as an extension of the Robot language using multiple attribute grammar inheritance. As can be noticed, semantic rule (`COMMAND.time=1;`) has to be repeated in several productions (`COMMAND ::= left`, `COMMAND ::= right`, `COMMAND ::= up`, `COMMAND ::= down`). New semantics in the RobotTime language can be seen as a new aspect which crosscuts the language structure. Therefore, the RobotTime language can be better specified using aspect-oriented attribute grammars. The RobotTime language specifications written in AspectLISA are shown in Fig. 3. Note that four pointcuts have been specified which match all seven productions in the Robot language. For example, pointcut `Begin` matches production `START ::= begin COMMANDS end` and pointcut `SimpleCommand` matches productions `COMMAND ::= left`, `COMMAND ::= right`, `COMMAND ::= up`, and `COMMAND ::= down`. To each pointcut, advice is attached which define the new semantics of matched productions (e.g., semantic for simple command is that each command spent one time slot `C.time = 1;`).

In [20], the RobotSpeed language has been defined as an extension of the RobotTime language. An additional `speed` construct has been added to the language such that the robot can now move with different speed. The RobotSpeed language can be specified purely with aspect-oriented techniques as shown in Fig. 4. Note that all pointcuts from the RobotTime language have been inherited. Only new advice have to be defined with additional semantics about speed of the movement. Hence, new advice extends previous advice that is inherited.

In Fig. 3 and Fig. 4 illustrative examples of AspectLISA are shown. The approach is scalable to larger languages and has been used in re-specifying the AspectCOOL language [3] which is an aspect-oriented extension of COOL (Class Object-Oriented Language) [5]. Typical examples of aspects in language specifications can be additional code generation, different language extensions (e.g., excep-

[5] Our COOL language should not be confused with the early domain-specific aspect-oriented COOL language by Lopes.

```
language RobotTime extends Robot {

    attributes double *.time;

    pointcut Begin<START, COMMANDS>  *.start : START ::= .. COMMANDS .. ;
    pointcut SimpleCommand<COMMAND>  *.move  : COMMAND  ::= * ;
    pointcut NoCommands<COMMANDS>  *.moves : COMMANDS ::= epsilon ;
    pointcut SeqCommands<COMMANDS[0], COMMAND, COMMANDS[1]>
                              *.moves : COMMANDS ::= COMMAND COMMANDS ;

    advice Init<S,C> on Begin {
       S.time = C.time;
    }

    advice SetTime<C> on SimpleCommand {
       C.time=1;
    }

    advice ClearTime<Cs> on NoCommands {
       Cs.time=0;
    }

    advice SumTime<C0, CM, C1> on SeqCommands {
       C0.time = CM.time + C1.time;
    }
}
```

Fig. 3. RobotTime Language using AspectLISA

tion handling, aspects, new paradigms), language specification debugging, attribute tracking.

6 Conclusion

In the paper, aspect-oriented attribute grammars has been proposed and formally defined. The concept has been incorporated into AspectLISA, an aspect-oriented compiler generator based on attribute grammars. Aspect-oriented programming is a very promising approach and has been successfully used in tools for language definition and implementation. Some of the known contributions in this field were reviewed, as a motivation for our proposal. LISA already has mechanisms to support inheritance and modularity. These mechanisms support nicely the notion of object-oriented aspects; on the other side, adding aspects will allow to write simpler specifications avoiding, for example, the repetition of semantic rules. The challenge in programming language definition is also to support reusability and extensibility: aspects will reinforce these features. Aspect-oriented features of the AspectLISA tool increase modularity since different concepts of programming language can be designed and implemented separately in different modules. These modules are also more reusable due to inheritance, which is successfully incorporated into our tool.

7 Acknowledgements

We would like to thank Jeff Gray for useful comments on earlier versions and Mitja Lenič for ideas and tips on implementation.

```
language RobotSpeed extends RobotTime {
 lexicon {
   Commands speed
   Number [0-9]+  }

 attributes int *.inspeed, *.outspeed;

 rule  extends start {
      compute {
     }
 }

 rule speed {
   COMMAND ::= speed #Number  compute {
     COMMAND.time = 0;    // no time is spent for this command
     COMMAND.outspeed = Integer.valueOf(#Number.value()).intValue();
      // this command does not change the position
     COMMAND.outp = COMMAND.inp;
   };
 }

 advice  extends Init<S,C> {
   C.inspeed = 1;  // beginning speed
   S.outspeed = C.outspeed;
 }

 advice SpeedPropagation  extends SumTime<C0, CM, C1> {
    CM.inspeed  = C0.inspeed;    // speed propagation
    C1.inspeed = CM.outspeed;    // to sub-commands
    C0.outspeed = C1.outspeed;
 }

 advice SameTime  extends ClearTime<Cs> {
    Cs.outspeed = Cs.inspeed;
 }

 advice  CalculateTime  extends SetTime<C> {
    C.time = 1.0/C.inspeed;
    C.outspeed = C.inspeed;
 }
}
```

Fig. 4. RobotSpeed Language using AspectLISA

References

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

[2] AspectG. http://www.cis.uab.edu/wuh/ddf/index.html, 2006.

[3] E. Avdičaušević, M. Lenič M. Mernik, and V. Žumer. AspectCOOL: An experiment in design and implementation of aspect-oriented language. *ACM SIGPLAN Notices*, 36(12):84–94, December 2001.

[4] O. de Moor, S. L. Peyton Jones, and E. Van Wyk. Aspect-oriented compilers. In *Generative and Component-Based Software Engineering (GCSE)*, pages 121–133, 1999.

[5] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[6] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, 1992.

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

[8] 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.

[9] M. Jourdan, D. Parigot, C. Julie, O. Durin, and C. Le Bellec. Design, implementation and evaluation of FNC-2 attribute grammar system. In *Proc. of the ACM Sigplan'90 Conference on Programming Language Design and Implementation*, pages 209–222, 1990.

[10] K. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. In N.M.-O. Horatiu Cirstea, editor, *Proceedings of the 6th International Workshop of Rule-Based Programming (RULE)*. ENTCS, Nara, Japan, Elsevier, April 2005.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Communications of the ACM (Special issue on Aspect-Oriented Programming)*, 44(10):59–65, October 2001.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP97 Object-Oriented Programming, Lecture Notes in Computer Science*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[13] P. Klint, T.van der Storm, and J.J. Vinju. Term rewriting meets aspect-oriented programming. Technical report, Centrum voor Wiskunde en Informatica, 2004.

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

[15] J. Kollár and M. Tóth. Temporal logic for pointcut definitions in AOP. *Acta Electrotechnica et Informatica*, 5(2):15 – 22, 2005.

[16] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of AOP with generative programming in aspectC++. In *GPCE*, pages 55–74, 2004.

[17] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4), 2005. To appear.

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

[19] 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.

[20] M. Mernik and V. Žumer. Incremental programming language development. *Computer Languages, Systems and Structures*, (31):1–16, 2005.

[21] J. Paakki. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196 – 255, 1995.

[22] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[23] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.

[24] C. A. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison Wesley, Second edition, 2002.

[25] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF + SDF meta-environment: A component-based language development environment. *Lecture Notes in Computer Science*, 2027:365–370, 2001.

[26] H. Wu, J. Gray, S. Roychoudhury, and M. Mernik. Weaving a debugging aspect into domain-specific language grammars. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1370–1374, New York, NY, USA, 2005. ACM Press.

[27] E. Van Wyk. Aspects as modular language extensions. *Electronic Notes in Theoretical Computer Science*, 82(3), 2003.