Specifying languages using aspect-oriented approach: AspectLISA

Damijan Rebernak, Marjan Mernik
University of Maribor, Faculty of Electrical Engineering and Computer Science
Smetanova ulica 17, 2000 Maribor, Slovenia
{damijan.rebernak, marjan.mernik}@uni-mb.si
tel: ++386-2-220-7462 fax: ++386-2-2511-178

Pedro Rangel Henriques, Daniela da Cruz
University of Minho, Department of Computer Science
Campus de Gualtar
4710 - 057 Braga, Portugal
{prh, danieladacruz}@di.uminho.pt

Maria João Varanda Pereira
Polytechnic Institute of Bragança
Campus de Sta. Apolónia
Apartado 134 - 5301-857, Bragança, Portuga
mjoao@ipb.pt

Abstract. *Object-oriented techniques and concepts have been successfully used in language specification and formalization. They greatly improve modularity, reausability and extensibility. In spite of using OO paradigms in language specification, some semantic aspects still crosscut many language constructs. Improvements can be achieved with aspect-oriented techniques. The paper describes AspectLISA tool wich uses aspect-oriented approach for language specification (aspect-oriented attribute grammars). An example will be worked out in order to illustrate the approach. We will show how to identify an aspect, specify it in the concrete AspectLisa syntax, and how to gather parts in order to develop a complete language processor.*

Keywords. Attribute grammars, aspect oriented programming, compiler/interpreter generator

1 Introduction

It is a well-known fact that programming language definitions are hard to be efficiently modularized. Moreover, new programming languages are hard to build simply by incorporating different language components due to complex interactions among different language features. Here, object-oriented techniques and concepts, like encapsulation and inheritance, have much to offer and improve language specifications toward better modularity, reusability, and extensibility. Object-oriented notations have been integrated with attribute grammars a long time ago [13]. In this case context-free grammars define the class hierarchy. 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. One of the shortcomings of this approach is that right-hand nonterminals cannot have inherited attributes and the other is that only small features can be added to the language. In other words, language can not evolve dramatically. Another problem is that the class hierarchy defines the modularization based on language syntax constructs, whereas the language developer also wants to have modules based on different aspects (e.g. name analysis, type checking, code generation, etc). The goal of intentional programming (IP) [2] was also a modular language implementation system where intentions are plug-and-play components. Achieving independence of components (intentions) was the main technical challenge. Modularity and reusability was achieved using forwarding [15], a variation of inheritance in attribute grammars, and aspects. A programming language can be built simply by importing an appropriate set of such components or can be extended by a rich set of features, and each of these features is a reusable component. The IP project failed despite state-of-the art modularity of language specifications being achieved. In our opinion the reason is that it was too ambitious, expecting death of programming languages. Moreover, again it was proven how complex the interactions of different language features are. Modularity and reusability can be achieved also using other non object-oriented techniques. One of the recent achievements regarding better reusability and modularity of action semantics is reported in [4]. The authors propose a finer modular structure where a new semantic equation module is constructed for each production. The final language definition module is obtained simply by importing them together, assuming that the symbols they share correspond to common features. It is our belief that a fine modular structure is not feasible for real programming languages, just as a monolithic structure is infeasible, since optimal granularity is somewhere between two extreme options. Modularity and extensibility of specifications based on denotational semantics are much harder to achieve. Some attempts were made in [8]. Despite their usefulness language specification languages are not popular. Among the reasons are classical ones such as that they are hard to understand, modify and maintain. Many of these problems can be attributed to non-modularity, non-extensibility and non-reusability of language specifications.

As already mentioned, the use of object-oriented techniques and concepts, like encapsulation and inheritance, greatly improves language specifications towards better modularity, reusability and extensibility. However, additional improvements can be achieved with aspect-oriented techniques since semantic aspects also crosscut many language constructs. Indeed, aspect-oriented constructs have already been added to some language specifications.

In this paper an aspect-oriented extension to LISA (AspectLISA) specification language is presented. The LISA system is the compiler/interpreter generator based on object-oriented attribute grammars. The paper is organized as follows. A brief introduction to aspect-oriented approach in language development and related work is presented in section 2. Section 3 is most important and describes our aspect-oriented approach and tool for language development. A case study which illustrate our ideas follows in section 4. The concluding comments are mentioned in section 5.

2 Specifying a language with aspects

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 [3]. However some issues, called aspects, are system wide and cannot be put into a single functional component. 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, difficult to develop, understand and evolve. A solution is provided by aspect-oriented programming (AOP) [6] 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 [6, 9] similar problems exists in domain-specific languages [10]. For example, in language specifications modularization is usually based on language syntax constructs (e.g., declarations, expressions, commands). Adding new functionality to the existing language, such as a new expression, can be usually done in a modular way. Only syntax production and semantics for expressions have to be changed. In this case a new feature does not crosscut other language components. However, many language extensions (e.g., type checking, code generation) required changes in many if not in all language components. Clearly, such language extensions are aspects that crosscut language components. Therefore, in this case the language modularization based on different aspects would be more beneficial. To overcome this problem aspect-oriented techniques should be used in

language specifications.
Introduction of AOP in language development increases modularity, readability and reuse of language specifications. Different concepts are defined separately and are therefore not part of original language. With introduction of aspects, semantics can be (depends on developer) detached from syntax and can be therefore used in different languages or/and in different productions.

2.1 Related Work

Aspect-oriented programming is a very promising approach and have been successfully used in tools for language definition and implementation. Aspects has been used for many different tasks, such as extension for weaving debugging information into DSL specifications [14].
In this section we briefly describe three of the more relevant contributions in the field, *using aspects in language specification or implementation*.
JastAdd [5] is a Java-based system for compiler construction. JastAdd is centered around object-oriented representation of the abstract syntax tree (AST). It 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. The idea of aspect-orientation in JastAdd is to define each aspect of language in separate class and then weave them together at appropriate places (pointcuts). With separation of different language aspects among different classes developers have the possibility to use all features of Java programming language to specify aspects.
The aspect-oriented language AspectG [1] was created for modular implementation of crosscutting concerns in ANTLR language definition. Since ANTLR belongs to syntax directed translations (semantic rules are not declaratively specified and order of semantic rules is important) AspectG uses 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 particular syntax,
- advice are similar to AspectJ notion (before and after) and brings together a pointcut (to pick out join points) and a body of code (semantic rules).

At last, AspectASF [7] is a simple aspect language for language specifications written in ASF+SDF formalism. Only rewrite rules are supported. Therefore, join points in AspectASF are static points in semantic equations. Aspects specify additional equations which are written in ASF formalism and are appended to semantic equations at appropriate places (join points). The aim of aspects in AspectASF is to declaratively specify which rules should be adapted to incorporate additional semantics (e.g. side-effect, rule tracing, etc.). To declare how/where these aspects will be weaved into original specifications AspectASF uses pointcut pattern language and advice which are applied to specified pointcut.

3 Aspects in LISA

In the LISA project [11, 12], 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 override 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 [13]. We call this multiple attribute grammar inheritance. With our approach, the language designer is able to add new features (syntax constructs and/or semantics) to the language in a simple manner by extending lexical, syntax and semantic specifications.

3.1 AspectLISA

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 section.

LISA features like multiple attribute grammar inheritance, simplify language specifications and contribute towards better reusability, modularity and extensibility. 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. To avoid this unpleasant situation, an aspect-oriented attribute grammar has been incorporated into LISA language specifications. This extension is called AspectLISA, which is LISA extended with mechanism that enables to specify where to apply additional semantic rules. These points are known as join points in AOP. 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 rules and also on productions which are members of production rules. One pointcut can match rules/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 [6]. 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 following parts:

```
language L_1 [extends L_2, ..., L_N] {
   lexicon {
      [[Q] overrides | [Q] extends] R regular expr.
           :
   }
   attributes type At_1, ..., At_M
      :
   pointcut P< [S_1, ..., S_r] > L.Y : LhsP ::= RhsP ;
      :
   advice [[B] extends | [B] overrides] A< [T_1, ..., T_r] > on P {
      semantic functions
   }
      :
   rule [[Y] extends | [Y] overrides] Z {
      X ::= X_11 X_12 ... X_1p compute {
         semantic functions }
           :
      |
         X_r1 X_r2 ... X_rt compute {
         semantic functions }
      ;
   }
      :
   method [[N] overrides | [N] extends] M {
      operations on semantic domains
   }
   ...
}
```

Let's focus only on new aspect-oriented features of LISA specification language which are pointcuts and advice. As can be seen in formal AspectLISA language specifications, new features are part of language specifications. Every LISA specifications without new features can be used and extended with aspect-oriented features.

Pointcuts are defined using reserved word **`pointcut`**. Each pointcut has a unique name and a list of 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. Some examples of pointcut specifications are shown below:

`*.* :  * ::= .. ;`

matches any production in any rule in all languages across current language hieararchy

`nLPD.T* :  * ::= .. ;`

matches any production in all rules which start with `T` in `nLPD` language

`*.* :  TIP* ::= ..  *S ;`

matches all productions in any rule whose left hand side symbol satisfy pattern "TIP" and the right-hand side's last symbol ends with `S`*

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 contains information about the pointcut where advice will appear. An example of advice which is attached to pointcut `Test` is shown below:

```
pointcut Test<N, T, V> nLPD1.T* : * ::= .. ;

advice Beg<N, T, V> on Test{
        N.outProlog     = V.outProlog;
        T.inProlog      = "";
        V.inTableTypes  = T.outTableTypes;
        V.inProlog      = T.outProlog;
     }
```

In section 4 more examples of advice and pointcuts are provided.

3.2 AOP in LISA and inheritance

As we already mentioned pointcuts and advice can be reused using inheritance. All pointcuts of predecessors can be used in all ancestors. Pointcuts with same signature (name and parameters) as in ancestors can be used but cannot be extended in inherited languages and are overriden by default. Advice inherited from ancestors using **`extends`** keyword must be merged with the advice in the specific language. If advice exists in parent and inherited language then semantic functions of advice must be merged, otherwise advice are simply copied from inherited to current language. Advice can also override semantics of its parent using keyword **`override`**. Overriden advice cannot by weaved afterwards it has once been overriden.

4 Case Study

Typical examples of aspects in language specifications can be additional code generation, different language extensions (e.g., exception handling, aspects, new paradigms), language specification debugging, attribute tracking. In this section a small example is presented on language called nLPD, which has been used in teaching compilers at University of Minho.

This example will be used to show how we can identify an aspect; the difference between an extension and an aspect; how to specify an aspect using AspectLisa syntax and how to gather extensions and aspects in order to develop a language processor.

In nLPD all variables need to be declared after type declarations, where one must define the length of the type (bytes occupied by variable of the particular type). There are no pre-defined types in nLPD. Variables are allocated in memory continually from address 0.

The following tasks need to be computed:

1. Construct the type table.
2. Compute total memory space occupied by all declared variables in program.
3. Translate nLPD program into a set of Prolog-facts.

In order to perform these three tasks, semantic evaluations will be added to the nLPD grammar. The language is divided into two main parts: type definitions and variable definitions. So, the grammar has a set of productions related with types and another related with variables. For the first task (type table construction) just the first part of the grammar is used. An excerpt of LISA specifications for type table construction is:

```
language nLPD1
{
  lexicon {
     Number       [0-9]+
     Id           [a-z]+
     ...
  }
attributes Hashtable *.inTableTypes,
                     *.outTableTypes;

rule nLPD {
  NLPD ::= TIPOS VARS compute {
     // type table is stored in attribute outTableTypes
     NLPD.outTableTypes = TIPOS.outTableTypes;
     // initialize type table
     TIPOS.inTableTypes = new Hashtable();
  };
}
...
rule Type {
  TIPO::= #Id #Number compute {
     // store info about type name and type lenght into type table
     TIPO.outTableTypes =
            addItem(TIPO.inTableTypes, #Id.value(),
      integer.valueOf(#Number.value()).intValue());
  };
 }
}
```

To compute total memory space previous specifications are extended because this second task is related with another part of the grammar (variables). Note that only new semantic rules need to be specified. All others are inherited.

```
language nLPD2 extends nLPD1
{
   rule extends nLPD {
     NLPD ::= TIPOS VARS compute {
       NLPD.outTotalMem   = VARS.outTotalMem;
       NLPD.outDecls      = VARS.outDecls;
       VARS.inDecls       = new Vector();
     };
   }
...
   rule Single {
     SINGLE ::= IDS \: #Id compute {
       IDS.inTableTypes   = SINGLE.inTableTypes;
       IDS.inDecls        = SINGLE.inDecls;
       SINGLE.outTotalMem = SINGLE.inTotalMem +
                          ((IDS.outVars).size()) *
         lookupLengthType(SINGLE.inTableTypes,#Id.value());
       SINGLE.outDecls    = IDS.outDecls;
     };
   }

   rule Ids {
     IDS ::= #Id RIDS compute {
       RIDS.inVars= addElementVector(IDS.inTableTypes,
```

```
                          IDS.inVars,#Id.value());
      RIDS.inTableTypes  = IDS.inTableTypes;
      RIDS.inDecls = addElementVector(IDS.inTableTypes,
                         IDS.inDecls,#Id.value());
      IDS.outDecls = RIDS.outDecls;
      };
    }
   ...
}
```

To generate additional Prolog code, aspect-oriented specifications are used. This third task is performed using all the grammar productions. There is no grammar extension just a new aspect of the same productions will be specified. Implementing this task as an aspect we have to define a set of pointcuts and a set of advice in order to add new attribute evaluation statements in grammar productions.

```
language nLPD3  extends nLPD2 {

pointcut Begin<NLPD,TIPOS,VARS>
          *.nLPD: NLPD::=TIPOS VARS;
...
pointcut Type<TIPO,#Id,#Number>
           *.Type: TIPO::= #Id #Number ;
...
pointcut Sing<SINGLE,Ids,#Id>
           *.Single: SINGLE ::= IDS \: #Id;
...
advice Init<N,T,V> on Begin{
        N.outProlog      = V.outProlog;
        T.inProlog       = "";
        V.inTableTypes   = T.outTableTypes;
        V.inProlog       = T.outProlog;
     }
...
advice GenType<T,I,N> on Type {
 T.outProlog = T.inProlog + "type(" + I.value() +
                                 "," + N.value() + ")\n";
}
...
advice GenMem<S,I,Id> on Sing{
   S.memAdressOut = S.memAdressIn +((I.outVars).size())*
                lookupLengthType(S.inTableTypes,Id.value());
   S.outProlog    = S.inProlog +
          writeVars(S.inTableTypes,I.outVars,
                    Id.value(),S.memAdressIn);
   I.inVars       = new Vector();
  }
}
```

Therefore, `nLPD3` implements an aspect using the grammar specified by `nLPD2` and this will allow to perform the third proposed tasks.

5 Conclusion

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. In the near future we will work out more and more complex case studies in order to set up a method to decide whether a new feature is an extension and whether it is an aspect, clarifying when and how to use aspects in language definitions instead of the extension mechanism. We will also assess the efficiency of our weaving algorithm and decide if it deserves further improvement.

References

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

[2] O. de Moor. Intentional Programming. Invited talk at British Computer Society. http://web.comlab.ox.ac.uk/oucl/work/oege.demoor/talks/ip.pdf.gz, 2001.

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

[4] K. Doh and P. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36, 2003.

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

[6] 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.

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

[8] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *Proceedings of 6th European Symposium on Programming*, volume 1058, pages 219–234, 1996.

[9] 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.

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

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

[12] M. Mernik and V. Žumer. Incremental programming language development. *Computer Languages, Systems and Structures*, pages 1–16, 2005.

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

[14] 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.

[15] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of 11th Int. Conf. Compiler Construction*, pages 128–142, 2002.