ISSN 1392 – 1215 *ELECTRONICS AND ELECTRICAL ENGINEERING* 2011. No. 10(116)
ELEKTRONIKA IR ELEKTROTECHNIKA

T 120 *SYSTEM ENGINEERING, COMPUTER TECHNOLOGY*
SISTEMŲ INŽINERIJA, KOMPIUTERINĖS TECHNOLOGIJOS

Aspect-Oriented Attribute Grammars

M. Mernik, D. Rebernak
Faculty of Electrical Engineering and Computer Science, University of Maribor,
Smetanova 17, 2000 Maribor, Slovenia, phone: +386 2 220 7455, e-mails: marjan.mernik@uni-mb.si,
damijan.rebernak@uni-mb.si

crossref http://dx.doi.org/10.5755/j01.eee.116.10.894

Introduction

Attribute grammars were introduced by D. Knuth [1] and since then have proved to be useful in specifying the semantics of programming languages and in the automatic construction of compilers and interpreters. While implementation of programming languages is the original and most-widely recognized area of attribute grammars, they are also used in many other areas such as [2]: natural language interfaces, graphical user interfaces, visual programming, pattern recognition, hardware design, communication protocols, software engineering, static analysis of programs, databases, etc. However, only a few commercial compilers have been developed using attribute grammars as a design and implementation tool. It has been argued that attribute grammars are unsuitable for the production of high-speed compilers for general-purpose programming languages, since they are just a model of compilation and, thus, too primitive for real engineering discipline, and that they do not directly support the generation and optimization of machine code. The first problem is concerned with the pragmatic aspects of ordinary attribute grammars. Ordinary attribute grammars have deficiencies which become apparent in specifications for real programming languages. Such specifications are large, unstructured, and hard to understand, modify and maintain. Yet worse still, small modifications of some parts in the specification have widespread effects on the other specification parts. There has been a lot of research work on augmenting ordinary attribute grammars with extensions in order to overcome the deficiencies of attribute grammars, such as lack of modularity, extensibility and reusability. Several concepts, such as remote attribute access, object-orientation, templates, rule models, symbol computations, high order features etc., have been implemented in various attribute grammar specification languages. This problem is still insufficiently solved, despite all these different approaches. The aspect-oriented attribute grammar, introduced in this paper, is proposed to better solve this problem. In aspect-oriented attribute grammar, concepts from aspect-oriented programming [3] have been integrated with attribute grammars. Aspect-oriented programming is a recently proven approach for describing crosscutting concerns in a modular manner. The second problem of attribute grammars, the lack of proper support for code generation and optimization, has also been insufficiently solved as yet and is out of the scope of this paper. An immediate conclusion might be, however, that attribute grammars are more appropriate in the design and implementation of Domain-Specific Languages (DSLs) [4] rather than for development of conventional optimizing compilers. Domain-specific languages, such as database query languages, robot-control languages, hardware design languages, and mark-up languages are usually developed with an emphasis on high-level application abstractions rather than on optimal performance. On the other hand, there is a strong case for formally specifying the syntax and semantics of DSLs [4]. The proposed aspect-oriented attribute grammar is a feasible formalism from which a DSL compiler, as well as related tools, (e.g., debuggers) [5] can be automatically generated.

Attribute grammars

Attribute grammars are a generalization of Context-Free Grammars (CFGs) in which each symbol has an associated set of attributes that carry semantic information. Attribute values are defined by attribute evaluation rules associated with each production of context-free grammar. These rules specify how to compute the values of certain attribute occurrences as a function of other attribute occurrences. Semantic rules are localized to each context-free grammar production. Formally an attribute grammar consists of three components, a context-free grammar G, a set of attributes A, and set of semantic rules R

$$AG = (G, A, R). \tag{1}$$

A grammar G = (T, N, S, P), where T and N are sets of terminal and non-terminal symbols; $S \in N$ is the start symbol, which appear only on the left-hand side of the first production rule; and P is a set of productions ($P = \{p_0, p_1, ..., p_z\}$, $z > 0$) in which elements (also called grammar symbols) of set $V \in N \cup T$ appear in the form of pairs $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in V^*$. An empty right-hand side of production is denoted by the symbol ε. A set of attributes $A(X)$ is associated with each symbol $X \in V$. $A(X)$ is divided into two mutually disjointed subsets $I(X)$ of inherited attributes and $S(X)$ of synthesized attributes. Now $A = \cup A(X)$.

A set of semantic rules R is defined within the scope of a single production. A production $p \in P$, $p{:}X_0 \rightarrow X_1...X_n$ ($n \geq 0$) has an attribute occurrence $X_i.a$ if $a \in A(X_i)$, $0 \leq i \leq n$. A finite set of semantic rules R_p contains rules for computing values of attributes that occur in the production p, i.e., it contains exactly one rule for each synthesized attribute $X_0.a$ and exactly one rule for each inherited attribute $X_i.a$, $1 \leq i \leq n$. Thus R_p is a collection of rules of the form $X_i.a = f(y_1, ..., y_k)$, $k \geq 0$, where y_j, $1 \leq j \leq k$, is an attribute occurrence in p and f is a semantic function. In the rule $X_i.a = f(y_1, ..., y_k)$, the occurrence $X_i.a$ depends on each attribute occurrence y_j, $1 \leq j \leq k$. Now set $R = \cup R_p$. For each production $p \in P$, $p{:}X_0 \rightarrow X_1...X_n$ ($n \geq 0$) the set of defining attribute occurrences is $DefAttr(p)=\{X_i.a | X_i.a=f(...) \in R_p\}$. An attribute X.a is called synthesized ($X.a \in S(X)$) if there exists a production $p{:}X \rightarrow X_1...X_n$ and $X.a \in DefAttr(p)$. It is called inherited ($X.a \in I(X)$) if there exists a production $q{:}Y \rightarrow X_1...X...X_n$ and $X.a \in DefAttr(q)$.

The meaning of a program (values of the synthesized attributes of starting non-terminal symbol) is defined during the attribute evaluation process where the values of attribute occurrences are calculated for each node of an attributed tree of a particular program. More details about attribute grammars can be found in [1, 2].

Aspect-oriented attribute grammars

Several extensions of attribute grammars have been proposed [2, 6], where authors have tried to improve the modularity and reusability of attribute grammars. However, some problems still remain or haven't been sufficiently solved yet. In this section we propose an extension of attribute grammars with features known from aspect-oriented programming [3], in order to address some of these problems.

Aspect-oriented programming provides a way of modularizing crosscutting concerns. Crosscutting concerns can be found in various representations of software artifacts and in different steps of software life cycle (e.g., source code, models, requirements, language grammars), and attribute grammars are no exception. If we take a closer look at extensibility and reusability of language specifications as written in attribute grammars, we discover several points where the features of new language cannot be specified modularly (specifications of semantics of new language crosscut with basic specifications). There are certain types of language extensions (e.g., type checking, environment propagation, code generation) that may require changes in many (if not all) of the grammar productions. Because language specifications are also used to generate parsers, compilers, and language-based tools automatically (e.g., editors, type checkers, and debuggers) [7], the various concerns associated with each language tool are often scattered throughout the core language specification. Such language extensions for supporting tool generation emerge as aspects that crosscut language components. As such, these concerns often represent refinements over the structure of the grammar. Note, that we are only dealing with the crosscutting concerns of semantic part of specifications. The problem of extending/modularizing the syntax part of specifications is already sufficiently solved using mechanisms such as inheritance and templates in attribute grammars [8].

An important part of aspect-oriented languages is the join-point model (JPM). If we want to extend attribute grammars with aspect-oriented paradigms we should define the join-point model first. Since attribute grammar specifications are non-executable and declarative, join-points are static points where additional semantics might be applied. In attribute grammars semantics is specified within the scope of a single production. Therefore, join-points are grammar productions. The aspect part of specifications is defined within the advice and advice application part. Semantic concepts that crosscut basic grammar structures are defined in advice, and are further applied to join-points (defined by pointcuts) using the advice application part of specifications [10].

The definition of Aspect-Oriented Attribute Grammar (AOAG) starts here, but the reader is kindly invited to check the example given in the next section while reading the definition. AOAG is an attribute grammar (AG) extended with pointcut specifications, advice specifications and the advice application part. AOAG is, therefore, defined as

$$AOAG = (G, A, R, Pc, Ad, Aa). \qquad (2)$$

A set of pointcuts Pc is a set of pointcut productions, $Pc = \{pc_1, ..., pc_m\}$, where pointcut production pc_i, $1 \leq i \leq m$, is used to match a set of grammar productions. Pointcut production has the form

$$pName : LeftS \rightarrow RightS, \qquad (3)$$

where pName is an unique identifier, LeftS is the matching rule for the lefthand side non-terminal of a grammar production, and RightS is the matching rule for right-hand side of a grammar production. A pointcut production $pName : LeftS \rightarrow RightS$, selects a production $p \in P$, $p : X_0 \rightarrow X_1...X_n$ ($n \geq 0$) if X_0 matches LeftS and $X_1 ... X_n$ match RightS. Let Pm_i denote the set of grammar productions selected by a pointcut production pc_i, $Pm_i = \{p_i \mid p_i \in P \wedge matched(p_i, pc_i)\}$. Matched productions Pm (a set of join points) selected by pointcuts Pc are then defined as $Pm = \cup_{i=1..m} Pm_i$, $Pm \subseteq P$.

Next component, Ad is a set of advice specifications $Ad = \{ad_1, ..., ad_l\}$, where advice ad_k, $1 \leq k \leq l$, has the following form

$$aName < F_1, ..., F_r > \{ Rs_v \}, \qquad (4)$$

where aName is an unique identifier, symbols $F_r \in V \cup A$, $r \geq 0$, are formal parameters in semantic rules (Rs_v) specified in advice ad_k, and Rs_v, $v \geq 0$ is a set of semantic rules with following form:

$$Rs_v = \{X_j.a = f(y_1, ..., y_k) \mid a \in A(X_j),\ y_i \in A(X_0) \cup ... \cup A(X_n),\ 0 \le i \le k\}, \quad (5)$$

where $X_0, ..., X_n$ are non-terminal symbols from grammar production matched by a pointcut. The abstractions of semantic rules that are independent of the production rules structure can be specified in the body of advice. 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. Two additional pseudo-identifiers have been defined for this reason. Pseudo-identifiers LHS and RHS denote a left-hand side non-terminal, and a list of the right-hand side non-terminal symbols of a production.

Last component, Aa is a set of advice application statements, $Aa=\{aa_1,...,aa_t\}$, where advice application statement aa_u, $1 \le u \le t$, has the following form

$$\text{apply aName} < S_1, ..., S_q > \text{on pName}, \quad (6)$$

where apply is a reserved word, aName is the name of existing advice specification, pName is the name of existing pointcut specification, and $S_q \in V \cup A$, $q \ge 0$, is a set of actual parameters which are substituted with formal parameters of Ad during aspect weaving.

Aspect weaving is a process of composing core functionality modules with aspects, thereby yielding a working system. Attribute grammar aspect weaving is, therefore, a process of formulating monolite attribute grammar specifications from core specifications and additional modules and aspects. There are many different mechanisms for weaving. We propose static weaving for aspect oriented attribute grammars, which means final attribute grammar specifications are constructed from aspect-oriented specifications before attribute grammar specifications are further processed (e.g., by a compiler generator tool where the compiler is generated automatically). The weaving process in aspect-oriented attribute grammars is defined as follows. Defining attributes attached to symbols X_j , $0 \le j \le n$, defined in Ad, are defined by semantic rules in Rs_v. Advice ad_k (aName) is weaved on pointcut pc_i (pName), which match productions Pm_i. For each matched production $p_i \in Pm_i$, the actual set of semantic rules Ra_{ki} is obtained by replacing formal parameters F_j (specified in ad_k) by actual parameters S_j (specified in aa_u) in Rs_v (number of actual parameters of aa_u and formal parameters of ad_k must be the same; q = r). The set of semantic rules Ra obtained from advice Ad and pointcuts Pc is defined as

$$Ra = \cup_{k=1..l,i=1..m} Ra_{ki} \quad (7)$$

and needs to be weaved with core semantic rules Rp_i to obtain well defined attribute grammar AG = (G, A, R’) in the following manner:

$$Rp'_i = Rp_i \cup (\cup_{k=1..l} Ra_{ki}), \quad (8)$$

$$R' = \cup_{i=1..z} Rp'_i\ , \quad (9)$$

$$(G, A, R') = (G, A, R, Pc, Ad, Aa). \quad (10)$$

The weaving algorithm is presented in Fig. 1 (the input of the algorithm is aspect-oriented attribute grammar).

```
for u = 1 to t do
  // find matching pair (advice, pointcut)
  advice pointcut = find matching pair(aau);
  ad = advice_pointcut.get advice();
  pc = advice_pointcut.get pointcut());
  // find a list of matching production rules
  P = match(pc);
  for x = 1 to P.size() do
    // substitute formal parameters of advice with
    // actual parameters
    // apply semantic rules to a grammar production rule
    // obtained by P.get(x)
    weave(ad, P.get(x));
  end for
end for
```

Fig. 1. The weaving algorithm

An example

Some of the benefits regarding the aspect-oriented features of attribute grammars can be observed in the following small example. It can be observed that aspect-oriented features are very useful for extending language semantics in a modular manner thus avoiding repetition of the same semantic rules in many grammar productions. Another useful feature is generic advice specifications which can be applied to many syntax-independent grammar productions. Note, that this is a small example for proof of concept. The benefits are more extensive for larger languages [9]. The first part of the example presents ordinary attribute-grammar specifications. Each production has semantic rules which define attributes x (synthesized attribute) and y (inherited attribute). The attribute x represents the total number of symbols ‘a‘,‘b‘,‘c‘, and ‘d‘ in a given string, where the symbols ‘a‘ and ‘b‘ count once, the symbol ‘c‘ twice, and the symbol ‘d‘ three times.The pointcuts (pc_1, ..., pc_5) match productions where additional semantics from advice can be attached to original productions. Special wildcard symbols (‘..’, ‘*’) can be used for defining LeftS and RightS matching rule in pointcut production pc_i. Wildcard symbol ‘*’ denotes a grammar symbol or some part of its name and can be used in the LeftS and RightS. Wildcard symbol ‘..’ denotes zero or more grammar symbols $\alpha \in V$, and can be used only in the RightS. The pointcuts (pc_1, ..., pc_5) match productions where additional semantics from advice can be attached to original productions. The ‘advice’ part of example shows advice (ad_1, ..., ad_4) with semantics. Semantics of ad_4 is a generic abstraction of semantic rules and presents a value distribution pattern. In the ‘apply’ part of the example apply rules are presented. The final part of example shows results after weaving advice semantics to selected productions. As can be seen from the example, advice ad_3 is applied to three different pointcuts which match different productions. This is possible due to parameterization of the advice and advice application part.

Fig. 2 depicts the semantic tree of ordinary attribute grammar before additional semantic rules are added using aspect-oriented approach. The semantic tree after weaving is depicted in Fig. 3. As can be seen from the figures, many semantic operations can be added easily in a modular way, using the aspect-oriented attribute grammars.

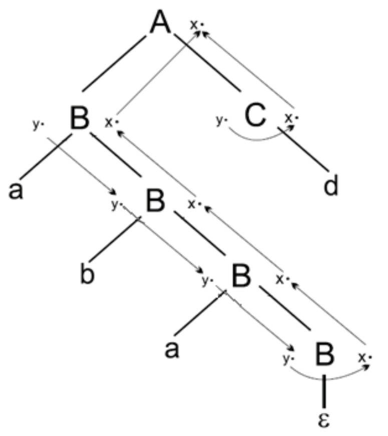


Fig. 2. Semantic tree before weaving

Ordinary attribute grammar specifications:

p_0: A → B C {A.x = B.x + C.x; B.y = 0; C.y = 1;}	// Rp_0
p_1: B → a B {B_0.x = B_1.x; B_1.y = B_0.y + 1;}	// Rp_1
p_2: B → b B {B_0.x = B_1.x + 1; B_1.y = B_0.y;}	// Rp_2
p_3: B → ε {B.x = B.y;}	// Rp_3
p_4: C → c {C.x = C.y + 1;}	// Rp_4
p_5: C → d {C.x = C.y + 2;}	// Rp_5

Pointcuts:

pc_1 : A → B C	// matches p_0
pc_2 : B → * B	// matches p_1 and p_2
pc_3 : B → ε	// matches p_3
pc_4 : C → ..	// matches p_4 and p_5
pc_5 : * → ..	// matches all productions

Advice:

ad_1 <X, Y, Z, val> {X.val = Y.val + Z.val;}
ad_2 <X, val> { X_0.val = X1.val + 1;}
ad_3 <X, val, value> {X_0.val = value;}
ad_4 <value> { RHS.value = LHS.value;}

Advice application:

apply ad_1 <A, B, C, cost> on pc_1
// Ra_{10} = {A.cost = B.cost + C.cost;}

apply ad_2 <B, cost> on pc_2
// Ra_{21} = {B_0.cost = B_1.cost + 1;}
// Ra_{22} = {B_0.cost = B_1.cost + 1;}

apply ad_3 <B, cost, 0> on pc_3
// Ra_{33} = {B.cost = 0;}

apply ad_3 <C, cost, 2> on pc_4
// Ra_{34} = {C.cost = 2;}
// Ra_{35} = {C.cost = 2;}

apply ad_3 <A, val, 0> on pc_1
// Ra_{30} = {A.val = 0;}

apply ad_4 <val> on pc_5
// Ra_{40} = {B.val=A.val; C.val=A.val;}
// Ra_{41} = {B_1.val = B_0.val;}
// Ra_{42} = {B_1.val = B_0.val;}

Final semantic rules (after weaving):

$Rp'_0 = Rp_0 \cup Ra_{10} \cup Ra_{30} \cup Ra_{40}$ =
{A.x = B.x + C.x; B.y = 0; C.y = 1;
A.cost = B.cost + C.cost; A.val = 0;
B.val = A.val; C.val = A.val;}

$Rp'_1 = Rp_1 \cup Ra_{21} \cup Ra_{41}$ =
{B_0.x = B_1.x; B_1.y = B_0.y + 1;
B_0.cost = B_1.cost + 1; B_1.val = B_0.val;}

$Rp'_2 = Rp_2 \cup Ra_{22} \cup Ra_{42}$ =
{B_0.x = B_1.x + 1; B_1.y = B_0.y;
B_0.cost = B_1.cost + 1; B_1.val = B_0.val; }

$Rp'_3 = Rp_3 \cup Ra_{33}$ =
{B.x = B.y; B.cost = 0;}

$Rp'_4 = Rp_4 \cup Ra_{34}$ =
{C.x = C.y + 1; C.cost = 2;}

$Rp'_5 = Rp_5 \cup Ra_{35}$ =
{C.x = C.y + 2; C.cost = 2;}

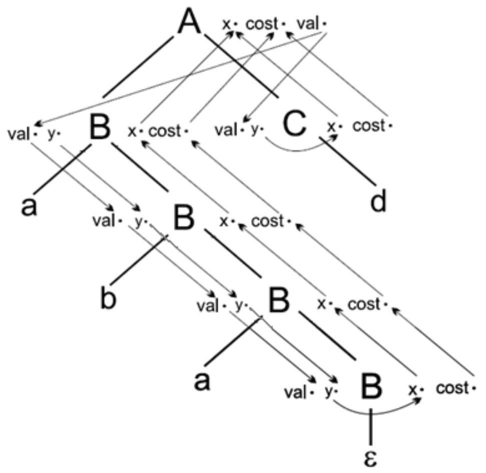
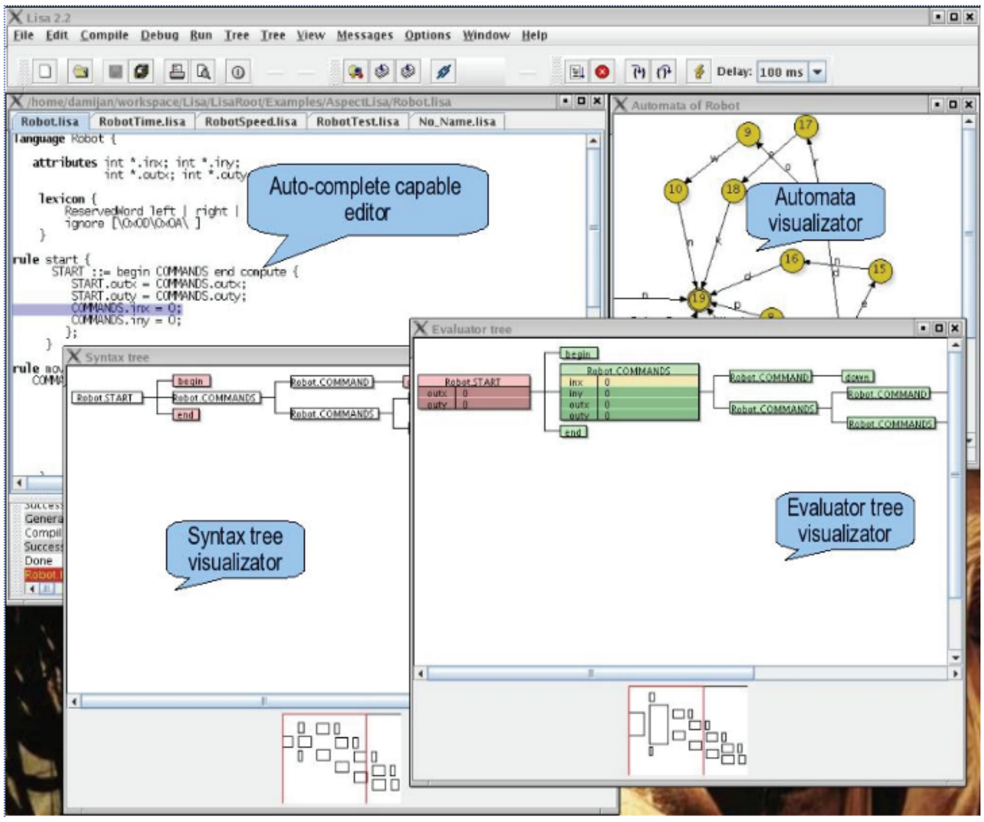


Fig. 3. Semantic tree after weaving

Experiences and conclusions

The proposed aspect-oriented attribute grammars have already been implemented and incorporated into our compiler-generator tool LISA (Fig. 4). LISA tool is also suitable for lifelong learning courses [9] and it is available at marcel.uni-mb.si/lisa. From initial experiments in the implementation of various small domain-specific languages, we have noticed several benefits of aspect-oriented attribute grammar specifications. Such specifications are not only shorter but, more importantly, more modular and reusable. Repetition of semantic rules can be completely avoided and several generic modules can easily be reused. The initial study shows that a developer's effort decreases down to 50% [10], which is encouraging enough to proceed with our research.

In this paper, aspect-oriented attribute grammars have been proposed and formally defined with the aim of better addressing crosscutting concerns that appear in language specifications (e.g. environment propagation, code generation, additional semantic rules needed to generate various language-based tools). Such specifications become more modular and reusable. The proposed aspect-oriented attribute grammars is a feasible formalism from which a DSL compiler [4, 11, 12] can be automatically generated, as well as related tools such as editors, simulators, and animators [13]. The approach can be useful also in model-driven engineering where code is automatically generated from models [14–16].


Lisa 2.2
File Edit Compile Debug Run Tree Tree View Messages Options Window Help
Delay: 100 ms
Robot.lisa RobotTime.lisa RobotSpeed.lisa RobotTest.lisa No_Name.lisa
Auto-complete capable editor
Automata of Robot
Automata visualizator
Syntax tree
Syntax tree visualizator
Evaluator tree
Evaluator tree visualizator

```
language Robot {

  attributes int *.inx; int *.iny;
             int *.outx; int *.outy

   lexicon {
       ReservedWord left | right |
       ignore [\0x0D\0x0A\ ]
   }

rule start {
     START ::= begin COMMANDS end compute {
       START.outx = COMMANDS.outx;
       START.outy = COMMANDS.outy;
       COMMANDS.inx = 0;
       COMMANDS.iny = 0;
      };
   }
```

Fig. 4. Tool LISA

References

1. **Knuth D.** Semantics of context–free languages // Mathematical Systems Theory. – Springer, 1968. – No. 2(2). – P.127–145.
2. **Paakki J.** Attribute Grammar Paradigms – A High–Level Methodology in Language Implementation // ACM Computing Surveys. – ACM, 1995. – No. 2(27). – P.196–255.
3. **Kiczales G.** Aspect–oriented programming // ACM Computing Surveys. – ACM, 1996. – No. 4(28). – Article No. 154.
4. **Mernik M., Heering J., Sloane A.** When and how to develop domain–specific languages // ACM Computing Surveys. – ACM, 2005. – No. 4(37). – P.316–344.
5. **Wu H., Gray J., Mernik M.** Grammar–driven generation of domain–specific language debuggers // Software Practice and Experience. – Wiley, 2008. – No. 10(38). – P.1073–1103.
6. **Dueck G. D. P., Cormack G. V.** Modular attribute grammars // Computer Journal. – Oxford University Press, 1990. – No. 2(33). – P. 164–172.
7. **Henriques P., Varanda Pereira M. J., Mernik M., Lenič M., Gray J., Wu H.** Automatic generation of language–based tools using LISA // IEE Proceedings – Software Engineering. – Institution of Engineering and Technology, 2005. – No. 2(152). – P. 54–69.
8. **Mernik M., Lenič M., Avdičaušević E., Žumer V.** Multiple Attribute Grammar Inheritance // Informatica. – Slovenian Society Informatika, 2000. – No. 3(24). – P. 319–328.
9. **Zeman T., Hrad J.** Lifelong Learning Courses – Not Just an Alternative Way // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 6(102). – P. 23–26.
10. **Rebernak D., Mernik M., Wu H., Gray J.** Domain–specific aspect languages for modularizing crosscutting concerns in grammars // IET Software. – Institution of Engineering and Technology, 2009. – No. 3(3). – P.184–200.
11. **Živanov Ž., Rakič P., Hajdukovič M.** Using Code Generation Approach in Developing Kiosk Applications // Journal on Computer Science and Information Systems. – ComSIS Consortium, 2008. – No. 1(5). – P.41–59.
12. **Slivnik B., Vilfan B.** Producing the left parse during bottom–up parsing // Information Processing Letters. – Elsevier, 2005. – No. 96. – P. 220–224.
13. **Topaloglu N., Gürdal O.** A Highly Interactive PC based Simulator Tool for Teaching Microprocessor Architecture and Assembly Language Programming // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 2(98). – P. 53–58.
14. **Pavalkis S., Nemuraite L., Tarvydas P., Noreika A.** Specification of Finite Element Model of Electronic Device using Model–driven Wizard–based Guidance // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 2(98). – P. 59–62.
15. **Schmidt C.** Guest Editor's Introduction: Model–Driven

Engineering // IEEE Computer. – IEEE, 2006. – No. 2(39). – P. 25–31.

16. **Sprinkle J., Mernik M., Tolvanen J.–P., Spinellis D.** Guest Editors' Introduction: What Kinds of Nails Need a Domain–Specific Hammer? // IEEE Software. – IEEE, 2009. – No. 4(26). – P. 15–18.

Received 2010 10 12
Accepted after revision 2011 04 27

M. Mernik, D. Rebernak. Aspect-Oriented Attribute Grammars // Electronics and Electrical Engineering. – Kaunas: Technologija, 2011. – No. 10(116). – P. 99–104.

Despite the efforts of several researchers modularization, reusability and extensibility remain a problem within the area of language specification. Attribute Grammars (AGs) present a well-known formal approach for defining programming languages. This paper presents a new approach to language specification which increases the level of attribute grammars modularity and reusability and decreases developers' effort for specifying a new language. The paper introduces Aspect-Oriented Attribute Grammars (AOAGs) which extend the original notion of attribute grammars with features known from Aspect-Oriented Programming (AOP). Ill. 4, bibl. 16 (in English; abstracts in English and Lithuanian).

M. Mernik, D. Rebernak. Aspektų atžvilgiu orientuota būdingoji gramatika // Elektronika ir elektrotechnika. – Kaunas: Technologija, 2011. – Nr. 10(116). – P. 99–104.

Nepaisant kelių mokslininkų pastangų, skirtų kalbų modulumui tirti, vis tiek išlieka kalbos specifikos problema. Pateikiamas formalus požiūris į gerai žinomas programavimo kalbas. Pristatomas naujas kalbos specifikos tyrimas, padidinantis modulumą, pritaikymą, o kartu ir mažinantis kūrėjų pastangas sukurti naują kalbą. Il. 4, bibl. 16 (anglų kalba; santraukos anglų ir lietuvių k.).