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

\[ AG = (G, A, R). \]  

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

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, joint-points are static points where additional semantics might be applied. In attribute grammars semantics is specified within the scope of a single production. Therefore, joint-points are grammar productions. The aspect part of specifications is defined within the advice and 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

$$\text{AOAG} = (G, A, R, Pc, Ad, Aa).$$

(2)

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

$$pName : \text{LeftS} \rightarrow \text{RightS},$$

(3)

where $pName$ is an unique identifier, $\text{LeftS}$ is the matching rule for the left-hand side non-terminal of a grammar production, and $\text{RightS}$ is the matching rule for right-hand side of a grammar production. A pointcut production $pName : \text{LeftS} \rightarrow \text{RightS}$, selects a production $p : X_0 \rightarrow X_1 \ldots X_n$ ($n \geq 0$) if $X_0$ matches $\text{LeftS}$ and $X_1 \ldots X_n$ match $\text{RightS}$. Let $P_m$ denote the set of grammar productions selected by a pointcut production $pc_v$, $P_m = \{p | p \in P \land \text{matched}(p, pc_v)\}$. Matched productions $P_m$ (a set of join points) selected by pointcuts $Pc$ are then defined as $P_m = \bigcup_{i=1..m} P_{m_i}$. $P_{m_i} = P$.

Next component, $Ad$ is a set of advice specifications

$$Ad = \{ad_1, \ldots, ad_l\},$$

(4)

where $aName$ is an unique identifier, symbols $F_r \in V \cup A$, $r \geq 0$, are formal parameters in semantic rules ($R_s$) specified in advice $ad_k$, and $R_s$, $v \geq 0$ is a set of semantic rules with following form:
productions P. For each matched production \( p_i \) the following manner:

\[
R_{sv} = \{ X_j.a = f(y_1, \ldots, y_k) \mid a \in A(X_j), y_i \in A(X_j) \cup \ldots \cup A(X_n), 0 \leq i \leq k \},
\]

where \( X_0, \ldots, X_n \) are non-terminal symbols from grammar productions 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, \( Aa = \{ a_{a1}, \ldots, a_{at} \} \), where advice application statement \( a_{au} \), \( 1 \leq u \leq t \), has the following form

\[
\text{apply } aName < S_1, \ldots, S_q > \text{ on } pName,
\]

where apply is a reserved word, \( aName \) is the name of existing advice specification, \( pName \) is the name of existing pointcut specification, and \( S_q \in V \cup A, q \geq 0 \), 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 \( X_0, 0 \leq i \leq n \), defined in \( Ad \), are defined by semantic rules in \( R_{sv} \). Advice \( a_{ad} \) (aName) is weaved on pointcut \( p_c \) (pName), which match productions \( P_m \). For each matched production \( p_c \in P_m \), the actual set of semantic rules \( R_{ak} \) is obtained by replacing formal parameters \( F_j \) (specified in \( a_{ad} \)) by actual parameters \( S_i \) (specified in \( a_{au} \)) in \( R_{sv} \) (number of actual parameters of \( a_{au} \) and formal parameters of \( a_{ad} \) must be the same; \( q = r \)). The set of semantic rules \( R_{a} \) obtained from advice \( Ad \) and pointcuts \( P_c \) is defined as

\[
R_a = \bigcup_{k=1}^{t_1} R_{ak},
\]

and needs to be weaved with core semantic rules \( R_{p} \) to obtain well defined attribute grammar \( AG = (G, A, R') \) in the following manner:

\[
R'_c = R_{p} \bigcup \bigcup_{k=1}^{t_1} R_{ak},
\]

\[
R' = \bigcup_{t=1}^{t_2} R'_c,
\]

\[
(G, A, R') = (G, A, R, P_c, Ad, Aa).
\]

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

\[ \begin{align*}
p_0 & : A \rightarrow B \, C \{ A.x = B.x + C.x; B.y = 0; C.y = 1; \} \quad // Rp_0 \\
p_1 & : B \rightarrow a \, B \{ B_0.x = B_1.x; B_1.y = B_0.y + 1; \} \quad // Rp_1 \\
p_2 & : B \rightarrow b \, B \{ B_0.x = B_1.x + 1; B_1.y = B_0.y; \} \quad // Rp_2 \\
p_3 & : B \rightarrow \varepsilon \{ B.x = B.y; \} \quad // Rp_3 \\
p_4 & : C \rightarrow c \{ C.x = C.y + 1; \} \quad // Rp_4 \\
p_5 & : C \rightarrow d \{ C.x = C.y + 2; \} \quad // Rp_5 \\
\end{align*} \]

Pointcuts:

\[ \begin{align*}
PC_1 & : A \rightarrow B \, C \quad // matches p_0 \\
PC_2 & : B \rightarrow * \, B \quad // matches p_1 and p_2 \\
PC_3 & : B \rightarrow \varepsilon \quad // matches p_3 \\
PC_4 & : C \rightarrow .. \quad // matches p_4 and p_5 \\
PC_5 & : * \rightarrow .. \quad // matches all productions \\
\end{align*} \]

Advice:

\[ \begin{align*}
ad_1 & <X, Y, Z, val> \{ X.val = Y.val + Z.val; \} \\
ad_2 & <X, val> \{ X_0.val = X_1.val + 1; \} \\
ad_3 & <X, val, value> \{ X_0.val = value; \} \\
ad_4 & <value> \{ RHS.value = LHS.value; \} \\
\end{align*} \]

Advice application:

\[ \begin{align*}
apply ad_1 <A, B, C, cost> & \text{ on } PC_1 \quad // Ra_{10} = \{ A.cost = B.cost + C.cost; \} \\
apply ad_3 <B, cost> & \text{ on } PC_2 \quad // Ra_{21} = \{ B_0.cost = B_1.cost + 1; \} \\
apply ad_2 <B, cost> & \text{ on } PC_3 \quad // Ra_{22} = \{ B_0.cost = B_1.cost + 1; \} \\
apply ad_3 <B, cost, 0> & \text{ on } PC_4 \quad // Ra_{33} = \{ B.cost = 0; \} \\
apply ad_3 <C, cost, 2> & \text{ on } PC_5 \quad // Ra_{34} = \{ C.cost = 2; \} \\
apply ad_3 <C, cost, 2> & \text{ on } PC_4 \quad // Ra_{33} = \{ C.cost = 2; \} \\
\end{align*} \]

Final semantic rules (after weaving):

\[ \begin{align*}
Rp'_{0} = Rp_0 \cup Ra_{10} \cup Ra_{30} \cup Ra_{40} = \\
\{ A.x = B.x + C.x; B.y = 0; C.y = 1; \\
A.cost = B.cost + C.cost; A.val = 0; \\
B.val = A.val; C.val = A.val; \} \\
Rp'_{1} = Rp_1 \cup Ra_{21} \cup Ra_{41} = \\
\{ B_0.x = B_1.x; B_1.y = B_0.y + 1; \\
B_0.cost = B_1.cost + 1; B_1.val = B_0.val; \} \\
Rp'_{2} = Rp_2 \cup Ra_{22} \cup Ra_{42} = \\
\{ B_0.x = B_1.x + 1; B_1.y = B_0.y; \\
B_0.cost = B_1.cost + 1; B_1.val = B_0.val; \} \\
Rp'_{3} = Rp_3 \cup Ra_{33} = \\
\{ B.x = B.y; B.cost = 0; \} \\
Rp'_{4} = Rp_4 \cup Ra_{34} = \\
\{ C.x = C.y + 1; C.cost = 2; \} \\
Rp'_{5} = Rp_5 \cup Ra_{35} = \\
\{ C.x = C.y + 2; C.cost = 2; \} \\
\end{align*} \]

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].

Fig. 4. Tool LISA

References

15. Schmidt C. Guest Editor’s Introduction: Model–Driven

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).


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.).