Implementation of Multiple Attribute Grammar Inheritance in the tool LISA

Marjan Mernik, Viljem Žumer, Mitja Lenič, Enis Avdičaušević
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova 17, 2000 Maribor, Slovenia
marjan.mernik@uni-mb.si

Abstract

Multiple attribute grammar inheritance is a structural organization of attribute grammars where the attribute grammar inherits the specifications from ancestor attribute grammars, may add new specifications or may override some specifications from ancestor specifications. In the paper the implementation of multiple attribute grammar inheritance is described. The proposed approach is successfully implemented in the compiler/interpreter generator tool LISA ver. 2.0.

Keywords: programming language specifications, object-oriented attribute grammars, inheritance, compiler generators

1. Introduction

We have developed a compiler/interpreter generator tool LISA which automatically produces a compiler or an interpreter from the ordinary attribute grammar specifications [Mernik 1995, Žumer 1997]. But in this version of the tool the incremental language development was not supported, so the language designer had to design new languages from scratch or by scavenging old specifications. The language design process should be supported by modularity and abstraction in a manner that allows incremental changes as easily as possible. This is one of the strategic directions of further research on programming languages. When introducing new concept designer has difficulties integrating it into the language in an easy way. Therefore 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 inherits the properties of its ancestors, and may introduce new properties that extend, modify or defeat its inherited properties. We have applied inheritance, a characteristic feature of object-oriented programming, in attribute grammars [Knuth 1968, Deransart 1990, Alblas 1991, Paakki 1995, Parigot 1999]. In [Mernik 1998a] the multiple attribute grammar is introduced only in the informal manner through examples of a simple calculator language, and in [Mernik 1999] a formal definition of multiple attribute grammar inheritance is presented. The multiple attribute grammar inheritance introduces some intricacies and therefore standard algorithms must be modified. In this paper the implementation details of multiple attribute grammar inheritance are described. The multiple attribute grammar inheritance approach is successfully implemented in the newly developed version of the tool LISA ver. 2.0. We have incrementally designed various small programming languages, such as PLM [Mernik 1998b] with multiple attribute grammar inheritance. Our experience with these non-trivial examples shows that multiple inheritance in attribute grammars is useful in managing the complexity, reusability and extensibility of attribute grammars. Huge specifications become much shorter and are easier to read and maintain. One of the benefits of this approach is also that for each language increment a compiler can be generated and the language tested.

2. Example

Multiple attribute grammar inheritance is a structural organization of attribute grammars where the attribute grammar inherits the specifications from ancestor attribute grammars, may add new specifications, and may override some specifications from ancestor specifications. With inheritance we can extend the lexical, syntax and semantic parts of programming language specification. In object-oriented languages the properties that consist of instance variables and methods are subject to modification. Since in attribute grammars semantic rules are tightly coupled with particular production rules, properties in multiple attribute grammar inheritance consist of lexical regular definitions, attribute definitions, rules which are generalized syntax rules that encapsulate semantic rules and methods on semantic domains. In our opinion the main weakness of multiple attribute grammar inheritance approach is that it does not help the designer in the case when languages have similar semantics and a very different syntax. For this reasons *templates* are also introduced. A template in attribute grammars is a polymorphic abstraction of a semantic rule parameterized with attribute occurrences which can be associated with many production rules with different nonterminal and terminal symbols. Since a nonterminal symbol can be considered as a class in object-oriented attribute grammars [Koskimies 1991], a template in attribute grammars is a kind of polymorphism. Further, at template instantiation appropriate semantic rules are generated at compiler generation time which is similar to templates in object-oriented languages where the code is generated at compile time. As an example, a value distribution pattern is described as:

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

A template describing the value distribution pattern is:

```
template <attributes Y_in, X_in*>
 compute valueDistribution {
    { X_in* = Y_in; }
}
```

The formal argument `X_in*` in the template `valueDistribution` is a variable list of arguments. Such arguments are denoted with an asterisk after the name. At the template instantiation a part of semantic rules enclosed with braces is generated for each argument in the variable list. Together with a variable list of arguments some functions are defined which can be used for variable list manipulation (first, last, succ, pred, empty). The usage of the above functions is presented in the next example. A pattern bucket brigade left is described as:

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

A template describing the pattern bucket brigade left is:

```
template <attributes Y_in, Y_out, X_in*, X_out*>
 compute bucketBrigadeLeft {
     if (empty(X_in*) && empty(X_out*))
         Y_out = Y_in;
     else
        first(X_in*) = Y_in;
        { X_in* = pred(X_out*);}
        Y_out = last(X_out*);
     endif
 }
```

Example 1:

Let us look at the example of a simple language with assignment statements, where expressions have no side effects. The meaning of the program:

a := 5
b := a + 1 + a + a

are the following values a=5, b=16. This language is specified in the tool LISA in a following manner.

```
language Expr {
   lexicon {
            Number     [0-9]+
            Identifier [a-z]+
            Operator   \+ | :=
            ignore      [\0x09\0x0A\0x0D\ ]+
   }
   attributes Hashtable *.inEnv, *.outEnv;
              int *.val;
```

```
rule Start {
  START ::= STMTS compute {
              STMTS.inEnv = new Hashtable();
              START.outEnv = STMTS.outEnv;
  };
}
rule Statements {
  STMTS ::= STMT STMTS compute {
          bucketBrigadeLeft<STMTS[0].inEnv,
                 STMTS[0].outEnv,
                 [STMT.inEnv,STMTS[1].inEnv],
                 [STMT.outEnv,STMTS[1].outEnv]>
   }
  | compute {  /* epsilon */
           bucketBrigadeLeft<STMTS.inEnv,STMTS.outEnv,[],[]>
   };
}
rule Statement {
        STMT ::= #Identifier \:= EXPR compute {
              EXPR.inEnv = STMT.inEnv;
              STMT.outEnv = put(STMT.inEnv, #Identifier.value(), EXPR.val);
          };
}
rule Expression1 {
        EXPR ::= EXPR + TERM compute {
              valueDistribution<EXPR[0].inEnv, [TERM.inEnv, EXPR[1].inEnv]>
              EXPR[0].val = EXPR[1].val + TERM.val;
          };
}
rule Expression2 {
        EXPR ::= TERM compute {
              valueDistribution<EXPR.inEnv, [TERM.inEnv]>
              EXPR.val = TERM.val;
          };
}
rule Term1 {
        TERM ::= #Number compute {
              TERM.val = Integer.valueOf(#Number.value()).intValue();
          };
}
rule Term2 {
        TERM ::= #Identifier compute  {
            TERM.val = ((Integer)TERM.inEnv.get(#Identifier.value())).intValue();
          };
}
method Environment {
        import java.util.*;
        public Hashtable put(Hashtable env, String name, int val)
          {
              env = (Hashtable)env.clone();
              env.put(name, new Integer(val));
              return env;
          }
   }
} // language Expr
```

If later designer need expressions with side effects he/she must change only those parts which differ from ancestor specifications. In our example we have to use bucket brigade left pattern instead of distribution pattern. Also a new rule `Term3` which produce side effect with following expression construct [id := EXPR] is introduced. For example next program:

a := 5
b := a + 1 + [a := 8] + a

produce the following values a = 8 and b = 22. The language `ExprSideEffect` is an extension of the language `Expr` where regular definitions `Number`, `Identifier` and `ignore`, attributes `inEnv`, `outEnv` and `val`, rules `Start`, `Statements` and method `Environment` are inherited and reused. The regular definition `operator`, rules `Statement`, `Expression1`, `Expression2`, `Term1` and `Term2` are extended and rule `Term3` is added. Note that computations for attribute `val` in rules `Expression1`, `Expression2`, `Term1` and `Term2` are inherited and hence not specified in `ExprSideEffect`.

```
language ExprSideEffect extends Expr  {
  lexicon {
            extends Operator  \[ | \]
  }
   rule extends Start {
        compute {} // first rule is starting production
   }
   rule extends Statement  {
        compute {
            bucketBrigadeLeft<STMT.inEnv,STMT.outEnv,
              [EXPR.inEnv],
              [put(EXPR.outEnv, #Identifier.value(), EXPR.val)]>
        }
   }
   rule extends Expression1 {
         compute {
              bucketBrigadeLeft<EXPR[0].inEnv, EXPR[0].outEnv,
                                [EXPR[1].inEnv, TERM.inEnv],
                                [EXPR[1].outEnv, TERM.outEnv]>
         }
   }
   rule extends Expression2 {
        compute {
             bucketBrigadeLeft<EXPR.inEnv,EXPR.outEnv,[TERM.inEnv],[TERM.outEnv]>
            }
   }
   rule extends Term1 {
        compute {
          bucketBrigadeLeft<TERM.inEnv, TERM.outEnv, [], []>
        }
   }
   rule extends Term2 {
        compute {
          bucketBrigadeLeft<TERM.inEnv, TERM.outEnv, [], []>
        }
   }
   rule Term3 {
     TERM ::= [ #Identifier \:= EXPR ] compute {
        TERM.val = EXPR.val;
        bucketBrigadeLeft<TERM.inEnv,TERM.outEnv,
              [EXPR.inEnv],
              [put(EXPR.outEnv,#Identifier.value(), EXPR.val)]>
     }
   }
} // language ExprSideEffect
```

3. Implementation

The concrete realisation of inheritance depends on many low level implementation issues such as the use of pseudo-variables self and super, delegation or concatenation, direction and completion of message lookup and ordered or unordered inheritance [Taivalsaari 1996]. In our case the subject of inheritance is a specification which is further used in compiler generation time to automatically generate the compiler. In its basic form, inheritance can be viewed as record combination (in our case as specification combination) which can be implemented either by delegation or concatenation. A problematic issue of multiple inheritance are name collisions. An unordered inheritance approach is used where name collisions are considered as errors. Such a case arises when two or more ancestor specifications use the same property name. In this case all ambiguities must be resolved explicitly. The name resolution process is described with the algorithm 1.

Algorithm 1:

Input: Id - name of the property to be resolved
    L - current language
    overriddenIdentifiers(L, E) - set of overridden identifiers in language L from language E
    supers(E) - set of ancestor languages for language E

method resolve(Id, L)
    ToVisit $\leftarrow$ {L}
    Result $\leftarrow \varnothing$
    repeat
        LookIn $\leftarrow$ ToVisit
        ToVisit $\leftarrow \varnothing$
        for $\forall$ E $\in$ LookIn **do**
            if Id $\in$ E **then**
                if Id $\notin$ overriddenIdentifiers(L, E) **then** Result $\leftarrow$ Result $\cup$ { Id }
            else ToVisit $\leftarrow$ ToVisit $\cup$ supers(E)
            end if
        end for
    until ToVisit = $\varnothing$
    if |Result| = 1 **then return** Result
    else if |Result|>1 **then return** horizontal overlap
    else return unknown property
    end if

Productions are inherited through the rules and can be extended or overridden in the new language. Productions are not directly the elements of inheritance, neither are the nonterminals and references to them which brings up some problems with nonterminal resolution (algorithm 2). The same nonterminal could be defined in more than one language. In the compilation phase all nonterminals are assigned to the their language, where similar techniques are used as in early and late binding.

Algorithm 2:

method resolveNonterminal(Id, L)
    ToVisit $\leftarrow$ {L}
    Result $\leftarrow \varnothing$
    repeat
        LookIn $\leftarrow$ ToVisit
        ToVisit $\leftarrow \varnothing$
        for $\forall$ E $\in$ LookIn **do**
            for each rule R in language E **do**
                if R $\notin$ overriddenIdentifiers(L, E) **then** ToVisit $\leftarrow$ ToVisit $\cup$ supers(E)
                for each nonterminal N in rule R **do**
                    if Id = name(N) $\wedge$ accessible(N, L) **then** Result $\leftarrow$ Result $\cup$ { N }
                end for
            end for
            if Result $\neq \varnothing$ **then**
                if |Result| = 1 **then return** Result
                else if |Result|>1 **then return** horizontal overlap
                end if
            end if
        end for
    until ToVisit = $\varnothing$
    return unknown nonterminal

When modifying references to nonterminals, identifiers in the semantics functions must also be modified. That action invalidates indexes to the nonterminals, so that all references must be reconsidered and indexes re-computed.

Example 2:
```
language A {
  lexicon {
    term   a | b
  }
  attributes int *.a;
  rule X {
     P ::= a A.P P
     compute { P.a = 1 + P[1].a + P[2].a; } ;  // A.P - early bind
  }
  rule Y {
     P ::= b    compute { P.a = 1; };
 }
}
language B extends A {
  rule START {
    G ::= P compute { G.a = P.a; };
  }
  rule extends Y {
     compute { P.a = 0; }
  }
}
```

From specifications above the following monolithic attribute grammar is generated:

```
B.G = B.P compute { B.G[0].a = B.P[0].a; };
B.P ::= b compute {B.P[a] = 0;}
// B.P ::= a A.P B.P compute { B.P[0].a = 1 + A.P[1].a + B.P[2].a;};
// error indexes must be re-computed
B.P ::= a A.P B.P compute { B.P[0].a = 1 + A.P[0].a + B.P[1].a;};  // O.K.
A.P ::= a A.P A.P compute { A.P[0].a = 1 + A.P[1].a + A.P[2].a;};
A.P ::= b compute {A.P[a] = 1;}
```

Another problem are lexical specifications. First, a nondeterministic finite-state automata (NFA) is generated from lexical specifications. Further in the process of conversion to deterministic finite automata (DFA) some final states are joined by using some rules for collision resolving. The commonly used rule is the first match rule, so the order of regular definitions becomes important. But this causes a problem when lexical specifications are inherited as presented in the following example:

```
AddSubCalc.digit    [0-9]
Dec.int             [0-9]+
```

For example, the input string “7” is recognised as `AddSubCalc.digit`. If the reference to the `Dec.int` was made in the specifications, the error would be reported, despite the correctness of specification. If the order of statements were different, the same problem would appear with reference to `AddSubCalc.digit`. Our solution to this problem is to find all matching regular definitions for the input string (Algorithm 3). In our example, for the input string “7” the result from lexical analyses would be a set {`AddSubCalc.digit`, `Dec.int`}. In that case the reference to both identifiers can be made and therefore the sequence of regular definitions becomes irrelevant. This solution has also impact on parser implementation.

Algorithm 3: Modified NFA to DFA conversion

$Q_h \leftarrow \{x \mid x \in Q \wedge \exists c \in \Sigma : |\delta(x,c)|>1\}$

while $Q_h \neq \emptyset$ **do**

  for $\forall c \in \Sigma : |\delta(S \in Q_h, c)|>1$ **do**

    $W \leftarrow \delta(S,c)$

    create new state T, $\forall d \in \Sigma : \delta(T,d) = \cup_{y \in W} \delta(y,d)$

    if $\exists x \in W \wedge \exists x \in F$ **then**

for all $x \in W \wedge x \in F$ **do**
type of token is set $R_T = R_T \cup R_x$
end for
$F \leftarrow F \cup \{T\}$
end if
if $\exists d \in \Sigma$: $|\delta(T,d)|>1$ **then**
$Q_h \leftarrow Q_h \cup \{T\}$
end if
$Q \leftarrow Q \cup \{T\}$
$Q_h \leftarrow Q_h \backslash \{S\}$
end for
end while
remove unreachable states and minimize automata

4. The tool LISA

Multiple attribute grammar inheritance is successfully implemented in the compiler/interpreter generator tool LISA ver. 2.0. The tool LISA is compiler generator with the following features:

- LISA is platform independent since it is written in Java
- the possibility to work in a textual or visual environment
- integrated development environment (fig 1.) where users can specify – generate – compile-on-the-fly – execute programs in a newly specified language
- lexical, syntax and semantic analysers can be of different types and can operate standalone
- visual presentation of different structures such as finite state automata, BNF, syntax tree, semantic tree, dependency graph
- animation of lexical, syntax and semantic analysers
- the specification language support multiple attribute grammar inheritance and templates which enable to design a language incrementally or reuse some fragments from other programming language specifications.

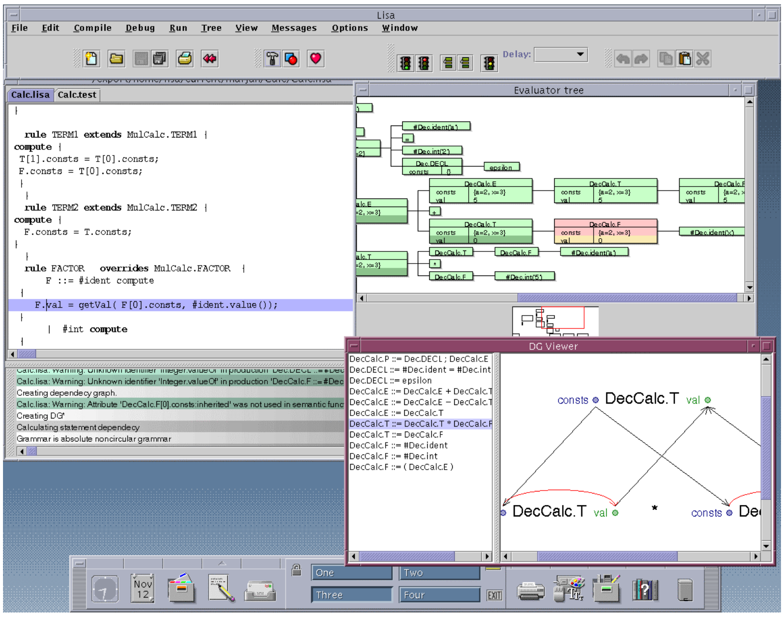

Fig 1.: LISA integrated development environment

5. Related Work

We borrowed the idea of grammar inheritance from [Aksit 1991] where only property is a production rule, and extended it to multiple attribute grammar inheritance. The difference between the approaches is also in the granularity of modification. In the approach of [Aksit 1991] modification is possible only for the whole production rule with

alternatives and semantic actions, since the only property is left hand nonterminal. In such cases the language designer must repeat all grammar productions when new semantic actions are added. In our approach the property is rule name and hence there is no need for repetition of productions.
In object-oriented attribute grammars [Koskimies 1991] the concepts of class and class hierarchies have been introduced where nonterminals act as classes and class hierarchies have been derived from context free grammar. Inheritance could be applied to attributes, attribute computations and syntactic patterns within one attribute grammar. It is also well known that inherited attributes and class hierarchies produce some conflicts on well-definedness of attribute grammars and hence multiple inheritance is not allowed, and also inherited attributes can not be used in dynamic classes. In our approach a different view is chosen where the whole attribute grammar is a class without the above mentioned conflicts.
With remote attribute access and inheritance [Kastens 1994], an attribution module is defined which can be reused in a variety of applications. But with this approach the attribution module can be only constructed for those attribute computations where the attribute depends only on remote attributes. In this case computation is associated to a symbol rather than to a production. With the inheritance described in [Kastens 1994] an attribute computation can be further independent from symbols used in particular language definitions.

6. Conclusion

With multiple attribute grammar inheritance, regular definitions, rules, attributes and operations on semantic domains can be inherited or overridden from ancestor specifications. In the paper an example and the implementation of multiple attribute grammar inheritance is described since multiple attribute grammar inheritance introduces some intricacies and therefore standard algorithms must be modified. The new version of the tool LISA, where the proposed approach is successfully implemented, is briefly described.

References

[Aksit 1991] Aksit M., Mostert R., Haverkort B. *Grammar Inheritance*. Technical Report, Department of Computer Science, University of Twente, 1991.
[Alblas 1991] Alblas H., Melichar B. (Eds.) *Attribute Grammars, Applications, and Systems*. Proceedings of SAGA, Lecture Notes in Computer Science Vol. 545, Springer - Verlag, 1991.
[Deransart 1990] Deransart P., Jourdan M. (Eds.) *Attribute Grammars and their Applications*. Proceedings of 1st WAGA, Lecture Notes in Computer Science Vol. 461, Springer - Verlag, 1990.
[Kastens 1994] Kastens U., Waite W.M. *Modularity and reusability in attribute grammars*. Acta Informatica Vol. 31, pp. 601 - 627, 1994.
[Knuth 1968] Knuth D. E. *Semantics of contex-free languages.* Math. Syst. Theory, Vol. 2, No. 2, pp. 127 - 145, 1968.
[Koskimies 1991] Koskimies K. *Object-Orientation in Attribute Grammars*, In Proceedings of SAGA, Alblas H., Melichar B. (Eds.), Lecture Notes in Computer Science Vol. 545, Springer - Verlag, pp. 297-329, 1991.
[Mernik 1995] Mernik M., Korbar N., Žumer V. *LISA: A Tool for Automatic Language Implementation*. ACM Sigplan Notices, Vol. 30, No. 4, pp. 71 - 79, 1995.
[Mernik 1998a] Mernik M., Žumer V. *Incremental Language Design*. IEE Proceedings Software, Vol. 145, No. 2-3, pp 85-91, 1998.
[Mernik 1998b] Mernik M., Lenič M., Avdičaušević E., Žumer V. *A Reusable Object-Oriented Approach to Formal Specifications of Programming Languages*. L'Objet Vol. 4, No. 3, pp. 273-306, 1998.
[Mernik 1999] Mernik M., Lenič M., Avdičaušević E., Žumer V. *Multiple Attribute Grammar Inheritance.* In Proceedings of 2nd WAGA, Parigot D., Mernik M. (Eds.), pp. 57-76, 1999.
[Paakki 1995] Paakki J. *Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation*. ACM Computing Surveys, Vol. 27, No. 2, pp. 196 - 255, 1995.
[Parigot 1999] Parigot D., Mernik M. (Eds.) *Attribute Grammars and their Applications*. Proceedings of 2nd WAGA, INRIA Publications, 1999.
[Taivalsaari 1996] Taivalsaari A. O*n the Notion of Inheritance.* ACM Computer Surveys, Vol. 28, No. 3, pp. 438 - 479, 1996.
[Žumer 1997] Žumer V., Korbar N., Mernik M. *Automatic Implementation of Programming Languages using Object Oriented Approach*. Journal of Systems Architecture, Vol. 43, No. 1-5, pp. 203 - 210, 1997.