This paper appears in the proceedings of *Programming Language Implementation and Logic Programming (PLILP '93)*, ed. by M. Bruynooghe and J. Penjam, Tallinn (ES), August 25–27, 1993, pp. 123–136, vol. 714 of *Lect. Notes in Comp. Sci.*, Springer-Verlag.

Specification and Implementation of Grammar Couplings using Attribute Grammars*

Martin Jourdan, Carole Le Bellec, Didier Parigot & Gilles Roussel

INRIA
Projet ChLoÉ, Bât. 13, Domaine de Voluceau, Rocquencourt, BP 105
F-78153 Le Chesnay Cedex, France
E-mail: {Martin.Jourdan,Carole.Le_Bellec,Didier.Parigot,
Gilles.Roussel}@inria.fr.

Abstract. This paper introduces the notion of a *coupling* of two grammars, defined by associations between their non-terminals and terminals. We present an algorithm for automatically producing, from these associations, an attribute grammar which specifies the translation from one grammar to the other.
The motivation for, and context of, this algorithm is our work aiming at improving modularity and reusability of attribute grammars. When it is combined with descriptional composition, we obtain what we consider to be the most declarative framework for this to date.

1 Motivation and Outline

1.1 On modularity and reusability for attribute grammars

Since Knuth's seminal paper introducing attribute grammars (AGs) [8], it has been widely recognized that this method is quite attractive for specifying every kind of syntax-directed computation, the most obvious application being compiler construction. Apart from pure specification-level features—declarativeness, structure, locality of reference—, an important advantage of AGs is that they are executable, i.e., it is possible to automatically construct, from an AG specifying some computation, a program which implements it. One could thus expect that they are used heavily to develop practical, production-quality, applications.

Unfortunately, it appears this is not yet the case, although AGs have been around for quite a long time and although powerful AG-processing systems are now available (e.g. Fnc-2 [6], which is the base of the present work, but see also [2] for a good list of other systems). In our opinion, the main reason for this is that AGs still cruelly lack the same support for modularity and reusability as the one which is offered by most programming languages, even the oldest ones.

Classical AGs are apparently a modular formalism: each production is a block in which only local computations (semantic rules) are specified, and which

* This work was supported in part by ESPRIT Project #5399 "Compare"

communicates with the rest of the AG through well-defined interfaces (non-terminals with their attributes). However this modularity is strongly dependent on the syntactic structure of the input data (underlying grammar). In addition, all aspects of an application, even those which are not directly related to one another, must be specified together on the same single grammar. This leads to huge, monolithic AGs which are hard to understand. So we need another base for modularity in AGs, less syntactic and more semantic. Indeed, most uses of AGs, e.g. in compiler construction, show that the same computations ("semantics") are used again and again in different applications, although on different grammars. So, in addition to modularity, we need support for reusability.

The basic observation which leads us to our proposal [9] for such an extension to AGs is the following one. An AG describes some computation (algorithm) on some grammar (data type). In most practical AGs however, several more or less independent algorithms are interleaved, and, for each algorithm, only a small part of the grammar is really of interest, and the way this algorithm is expressed depends on this grammar (names, etc.). It is thus very difficult to reuse whole parts of a given AG into another one for a slightly different problem or with a slightly different grammar.

In the last few years, different approaches [1,3,4,7] have been proposed to alleviate this problem. To motivate the work described in the present paper, we now sketch our own approach, which is similar to the one of [4]. It is based on the notion of Generic Attribute Grammars.

1.2 Generic Attribute Grammars

A *generic attribute grammar* is an AG which describes a well-defined, broad-purpose algorithm which it would be valuable to reuse easily; such a generic AG concentrates an expert's knowledge of the corresponding problem and of its solutions. To make it as general as possible, the syntax underlying the generic AG is reduced to the entities which have a role in the algorithm. Its attributes and semantic rules are reduced to those necessary to describe the computation.

For instance, the grammar of a generic AG for name analysis would only include such non-terminals as name definition, name use and block (and ordered lists of these), its attributes would include little more than a symbol table and its semantic rules would be derived from some specific but reusable set of visibility rules.

To instantiate such a generic AG on some given "real" grammar, the user identifies those of its syntactic entities which have a role in the algorithm described by the generic AG (key non-terminals and terminals). With each such key entity he associates the entity in the generic AG which play the same role (instantiation mapping). Under certain conditions, this mapping defines a function which "abstracts" each tree of the input grammar into the corresponding tree of the output grammar (the generic one). The result of the instantiation is defined as the result of the execution of the generic AG on the abstracted tree: the attributes are *transported back* to the corresponding nodes in the input tree.

There are several ways to implement this instantiation process. The one we have developed, although not the most general, has the advantages to be completely static and to rely on the well-known notions of coupling AGs and their descriptional composition,[2] introduced by Ganzinger and Giegerich [5]. In contrast, in [4] the detailed implementation of the above abstraction function is left to the user; this is more versatile but less attractive.

In this paper (section 2), we formalise the above notion of association between two grammars and name it *grammar coupling*; we study some conditions for this coupling to be well-defined (a function). In addition (section 3), we propose an algorithm which, given an output grammar (grammar of the generic AG), an input grammar (real grammar) and a valid coupling between them, constructs under certain feasibility conditions a *coupling attribute grammar* which implements the coupling function.

Then, the process of instantiating a generic AG reduces to meta-composing this coupling AG with the generic AG, which results in an AG defined on the "real" grammar with the same semantics as the generic one.

In this paper we focus on the technical aspects of grammar couplings and their implementation by AGs. Lack of space does not allow us to discuss their practical usefulness as a base for the notion of genericity in attribute grammars, in particular whether their expressive power is sufficient. Note however that our view of genericity in AGs is quite close to the one described and justified in [4]; the present work only aims at bringing more abstraction and automation to their instantiation process.

Lack of space also forbids illustrating the definitions and algorithms with an example. Please see [9,10] for more details.

2 Grammar Couplings

2.1 Informal outline

Let G_i be an input grammar and G_o an output grammar. In this section we want to define associations between the non-terminals and terminals of G_i and G_o (*grammar coupling*) and check their validity, i.e. that, for each tree of G_i, there exists a unique corresponding tree of G_o (in other words, they define a function). In the next section we will describe an algorithm for implementing (some of) these functions by *coupling attribute grammars*.

This correspondence between two grammars has some similarity with the notion of Covering Grammars [12], except that the latter is based on an association between productions.

We consider that, to be as declarative as possible, the specification of an association between entities of the two grammars should be done at the level of terminals and non-terminals. Association is easier with non-terminals which represent syntactic *notions* than with productions which represent syntactic *structures*. We hope that our algorithm will automatically solve the mapping between

[2] In the sequel we'll use the phrase *meta-composition*.

structures. In our opinion, this approach to the definition of grammar couplings puts minimal burden onto the user while keeping a reasonable expressive power.

2.2 Preliminaries

Definition 1. A *context-free grammar* is a tuple $G = (N, T, Z, P)$ in which:

- N is a set of non-terminals;
- T is a set of terminals, $N \cap T = \emptyset$;
- Z is the root non-terminal (start symbol), $Z \in N$;
- P is a set of productions,
  $p : X_0 \rightarrow X_1 \ldots X_n$ with $X_0 \in N$ and $X_i \in (T \cup N)$.

Throughout this paper we'll assume that every grammar is reduced.

For a grammar G, we will use the following notations:

- t denotes an (incomplete) syntax tree of G.
- u denotes a node of t.
- *label(u)* denotes the non-terminal at u.
- *prod(u)* denotes the production at u.

Other notations will be defined as needed, but most of them are hopefully quite well-known or self-explanatory.

2.3 Correspondence Application

A grammar coupling will be defined by an *application* specifying the *correspondence* between the input and output grammars. For such an application to define a function on trees of the input grammar, and to allow computations on output trees to be transported back to input trees, it must comply with some rules.

First, each non-terminal (resp. terminal) of the output grammar must be associated at least with one non-terminal (resp. terminal) of the input grammar. Secondly, we forbid that several non-terminals of output grammar be associated with the same non-terminal of input grammar. Finally we have a constraint on roots: the root non-terminal of the output grammar must be associated only with the root of the input grammar.

Definition 2 Correspondence application. A *correspondence application* Cor between the input grammar $G_i = (N_i, T_i, Z_i, P_i)$ and the output grammar $G_o = (N_o, T_o, Z_o, P_o)$ is defined by two applications Cor_T and Cor_N

$$Cor_N : N_o \rightarrow \bar{\mathcal{P}}(N_i),^{3}$$
$$Cor_T : T_o \rightarrow \bar{\mathcal{P}}(T_i)$$

with

$$Cor : N_o \cup T_o \rightarrow \bar{\mathcal{P}}(N_i) \cup \bar{\mathcal{P}}(T_i)$$
$$Cor(x) = \begin{cases} Cor_N(x) \text{ if } x \in N_o \\ Cor_T(x) \text{ if } x \in T_o \end{cases}$$

[3] $\bar{\mathcal{P}}(N)$ denotes the set of *not empty* subsets of N

so that

- $\forall x, y \in (N_o \cup T_o), x \neq y \Rightarrow Cor(x) \cap Cor(y) = \emptyset$;
- $Cor(Z_o) = \{Z_i\}$.

These restrictions are imposed in a first approach to make the validity test and the construction of the coupling AG simpler. At the end of this paper we'll discuss how some of them can be relaxed.

With the Cor application and the classical "derivation" relation $\Longrightarrow$ on G_i, different sets of terminals or non-terminals can be defined.

Definition 3 Key, neutral and dead non-terminals and terminals. Given G_i, G_o and Cor as above, we have:

- $N_k = \{X \in N_i \mid \exists Y \in N_o, X \in Cor(Y)\}$ is the set of *key non-terminals*;
- $T_k = \{X \in T_i \mid \exists Y \in T_o, X \in Cor(Y)\}$ is the set of *key terminals*;
- $N_n = \{X \in N_i - N_k \mid \exists Y \in N_k, X \stackrel{*}{\Longrightarrow} Y\}$ is the set of *neutral non-terminals*;
- $N_d = N_i - (N_k \cup N_n)$ is the set of *dead non-terminals*;
- $T_d = T_i - T_k$ is the set of *dead terminals*.

There is no neutral terminal.

2.4 Reduction to closed trees

From the association between grammar symbols given by Cor, we want to define an association between trees. The basic idea is that only key symbols are to be taken into account, so we split an input tree into *closed (sub-)trees*, i.e. subtrees delimited by key symbols.

To compute the set of closed trees, we transform the input grammar. The first step is to forget about dead symbols (those of T_d and N_d), which play no role in the coupling. For convenience, in the sequel we also call the resulting grammar G_i, with $T_i = T_k$ and $N_i = N_k \cup N_n$.

Definition 4 Closed tree. Given G_i, G_o and Cor as above, a closed tree is a tree of G_i such that:

- the root is a key non-terminal of N_k;
- the leaves are labeled by key symbols in $N_k \cup T_k$;
- the internal nodes are labeled by neutral non-terminals in N_n.

We assume that each of these closed trees is associated with a corresponding subtree of the output grammar; "corresponding" means that it has the same frontier, up to the mapping through Cor. Then, these output subtrees are pasted together according to the structure of closed trees in the input tree, and this gives the corresponding output tree, together with a link between every key node in the input tree and the corresponding node in the output tree.

To make sure that this process defines a function, and to make sure that we can construct this function statically, we impose that the number of different closed trees of G_i, and their size, be finite.

The second transformation of G_i is to eliminate the recursive neutral non-terminals, to make sure that we only have finite closed trees. More precisely, we detect the neutral non-terminals which derive into themselves without any key non-terminal in between. Indeed, assume that $X \in N_k$, $Y \in N_n$ and P_i contains the productions $X \rightarrow Y$, $Y \rightarrow Y\ Y$ and $Y \rightarrow X$. Then the closed trees constructed from X and Y, with X's on the frontier, are unbounded in number and size. So, one additional condition for a grammar coupling to be valid is that this kind of situation does not occur. We now give an algorithm to check this condition while transforming G_i so that the construction of the set of closed trees is easier.

For this purpose, we define a new derivation relation, noted $\underset{Cor}{\Longrightarrow}$. It is the restriction of $\Longrightarrow$, such that $X \underset{Cor}{\Longrightarrow} Y$ iff $X \Longrightarrow Y$ and $X \notin N_k$. The recursion elimination algorithm is then as follows:

Recursion Elimination Algorithm

1. Using the definition of derivation, we generate the set $N_r = \{X \in N_n \mid X \overset{*}{\underset{Cor}{\Longrightarrow}} X\}$.
2. On this set N_r, we build equivalence classes for the relation Γ defined by:

$$X\ \Gamma\ Y \Longleftrightarrow (X \overset{*}{\underset{Cor}{\Longrightarrow}} Y \text{and} Y \overset{*}{\underset{Cor}{\Longrightarrow}} X)$$

   We rename each non-terminal of N_r by the name of its equivalence class.[4].
3. In this new grammar, the neutral non-terminals should only derive into themselves through unit productions of the form $X \rightarrow X$ with $X \in N_r$, otherwise the coupling is deemed invalid.
4. Then, we eliminate all such productions from G_i.

In this new grammar G_i there is no recursion between neutral non-terminals any more. Hence we can start the construction of closed trees. For each $X \in N_k$, we want to generate all the possible closed trees of G_i with root X. To do so we use a fixed point algorithm, called the *tree closure algorithm*, which manipulates a set of (incomplete) trees of G_i. On each tree of this set we apply a transformation which generates a new set of trees, until this set solely contains closed trees. The transformation is composed of elementary steps, noted $t \xrightarrow{u,p} t'$, which, given a tree t, a leaf node u of t and a production p of G_i such that $label(u) = LHS(p)$[5] $\in N_n$, replaces u by the tree $p(u_1, \ldots, u_n)$. The starting set is the set of productions with LHS in N_k. The algorithm is presented in figure 1. As we have eliminated the recursive neutral non-terminals, we are sure that this algorithm terminates.

[4] This renaming is also applied on P_i

[5] Left Hand Side of p

$R' \leftarrow \{t \mid \exists p \in P_i, LHS(p) \in N_k, t = p(u_1, \ldots, u_n)\}$;
repeat
  $R \leftarrow R'$
  $R' \leftarrow \emptyset$;
  for each tree t in R
    if t is a closed tree then
      $R' \leftarrow \{t\} \cup R'$
    else
      for each u leaf of t with $label(u) = X \in N_n$
        for each production p with $X = LHS(p)$
          $t \xrightarrow{u,p} t'$;
          $R' \leftarrow \{t'\} \cup R'$;
until $R = R'$.

Fig. 1. Tree Closure Algorithm

2.5 Instantiation

Now, our goal is to determine, for each closed tree, if there exists a subtree of the output grammar which matches it. To make things simpler in a first step, we restrict here these output trees to be of height one, i.e. (plain) productions. At the end of this paper we'll extend the construction to more complex trees.

With each closed tree t we associate its *input virtual production* p; it is of the form $X_0 \rightarrow Y_1 \ldots Y_n$, where X_0 is the label of the root of t and the Y_i ($1 \leq i \leq n$) are from left to right the labels of the leaves of t (hence $Y_i \in (N_k \cup T_k)$). With this input virtual production we associate the corresponding *output virtual production*, which is its image through Cor, i.e. each symbol $X \in N_k \cup T_k$ is replaced by Y such that $X \in Cor(Y)$.

In the simple approach, we test if this output virtual production is a real production in G_o. If this is not the case, we don't know which output tree to associate with this closed tree, and we consider the grammar coupling Cor to be invalid. This is of course quite restrictive, but even so we believe that it covers many practical cases.

The result is a set of pairs (p_o, t) where p_o is in P_o. At this point it can be said that the correspondence between the two grammars is valid. It is obvious that a valid coupling defines a function from the trees of G_i to (a subset of) those of G_o, and it is easy to see how to implement it in an abstract way:

1. split an input tree into closed trees at key symbols;
2. for each such closed tree, insert the corresponding output production in the output tree, according to the location of the closed tree in the input tree, with links from each node in this output production to the corresponding key symbol in the input tree.

Then, any computation on the output tree can be transported back through these links to the nodes of the input tree, which is exactly what is needed for the instantiation of generic AGs.

3 Automatic Generation of the Coupling Attribute Grammar

In this section we want to implement a given valid grammar coupling by an attribute grammar which will perform the translation of G_i trees into G_o ones as defined above. This kind of attribute grammar is called *coupling attribute grammar* and we will note it: $\alpha : G_i \rightarrow G_o$. This notion was introduced by Ganzinger and Giegerich [5], together with their meta-composition.

Using the latter, it is then possible to apply on G_i trees some algorithm specified on G_o by an attribute grammar. More precisely, meta-composition produces, from a coupling AG from G_i to G_o and an AG on G_o, a new attribute grammar which computes on G_i trees the same result as the composition of these two AGs but without constructing the intermediate G_o tree. This is exactly what we meant above by "transporting back" the results of the computation on the output tree to the input one. In fact, the existence of meta-composition is a very good reason to try to implement grammar couplings with AGs.

3.1 Preliminaries

To introduce our algorithm, we first recall the classical definition of attribute grammars.

Definition 5 Attribute Grammar. An *Attribute Grammar* is a tuple (G, A, F) where :

- $G = (N, T, Z, P)$ is a context-free grammar as in definition 1.
- $A = \bigcup A(X)$ is a set of attributes;
- $F = \bigcup F(p)$ is a set of semantic rules where $f_{p,a,i}$ designates the semantic rule defining the attribute a of non-terminal X_i in production p.

We introduce a simple definition of Coupling Attribute Grammar, derived from the one by Ganzinger and Giegerich [5].

Definition 6 Coupling Attribute Grammar. We call *Coupling Attribute Grammar* of G_1 and G_2, noted $\alpha : G_1 \rightarrow G_2$, an attribute grammar $\alpha = (G_1, A_\alpha, F_\alpha)$ such that, for each $a \in A_\alpha$ of type S, $S \in (N_2 \cup T_2)$ and for each $f_{p,a,i} \in F_\alpha$, $f_{p,a,i}$ is either a copy rule or a tree-construction function associated with some production of P_2.

A coupling AG $\alpha : G_1 \rightarrow G_2$ takes as input a tree of G_1 and gives as output a tree of G_2.[6]

[6] This definition is a restriction of Ganzinger & Giegerich's one: it forces t_2 to be (isomorphic to) a subset of t_1, it disallows any semantic computation and forbids $f \in F_\alpha$ to be a complex tree construction.

3.2 Generation of the coupling AG

The construction of the coupling AG starts where the validation test ended, i.e. with the set of closed trees and associated output productions. We want to build a purely synthesised[7] coupling AG $\alpha = (G_i, A_\alpha, F_\alpha)$ which performs on each such pair (p_o, t) the translation of t into p_o.

To clarify the construction of α, we decompose our algorithm into two steps. First we construct pseudo-AGs on each pair (p_o, t). Then we cross-check the resulting pseudo-AGs to produce the coupling AG α. Any of these steps may fail, even if the coupling is valid.

Initial constraint First of all, there exists a strong constraint on each key non-terminal X, induced by the Cor application, and which stems from the above notion of link between a key node and the corresponding output node: X must carry exactly one attribute, of type $Cor^{-1}(X)$. For convenience, the name of this attribute will be the name of its type.

Construction of a pseudo attribute grammar on a pair (p_o, t) The final coupling AG must be such that, at some node u of the tree t of each pair (p_o, t), the construction of p_o is performed. Above u, copy rules transport the resulting tree to the root of t; below u, copy rules transport the arguments of this construction, available at the leaves of t. In general there are several choices for u.

To determine them, we apply a bottom-up algorithm on t. The elementary step on each node u is to decide if the construction of p_o is possible on the production $p = prod(u)$. Otherwise all attribute occurrences in the RHS of p should be transferred to the LHS.

In some cases this transfer operation induces on the LHS non-terminal the declaration of more than one attribute with the same name (type) . Therefore we introduce a notation of indexed attribute, noted (a, j), which replaces classical attributes. The index j permits to distinguish them without losing their relative order.

For each t and each $X \in N_i$ we have a set $A_t(X)$ of indexed attributes (a, j), the type of which is $a \in (N_o \cup T_o)$. For each t and each $p \in P_i$ we have a set $F_t(p)$ of semantic rules between indexed attributes, noted $f_{p,(a,j)}$.[8]

In the closed tree t there is no difference between key non-terminals and key terminals. So we translate the initial constraint as follows:

$$\forall X \in N_k \cup T_k, A_t(X) = \{(a, 1)\} \text{ where } X \in Cor(a).$$

[7] This restriction aims at reducing the complexity of the construction and also helps the meta-composition process by making the result AG of the same class as the generic AG.

[8] We forget the non-terminal position which is always zero because α is purely synthesised.

To simplify our algorithm, we don't make explicit checks of uniqueness of elements of $F_t(p)$; it is clear that non-uniqueness implies failure of the construction of α.

Now we present the elementary step on the node u with $p = prod(u)$. In this version we do the construction only on the root production. The elementary step uses two functions, $AP(p)$ constructing a set AP of indexed attributes for $LHS(p)$ and $FP(p)$ constructing a set FP of semantic rules for p, and a predicate *Construction(p)* testing if the p_o construction is possible on p. AP, FP and *Construction* are defined in Fig. 2. From the definition of (p_o, t) and

function $AP(p)$
  $p : X_0 \rightarrow X_1 \ldots X_n$
  $AP \leftarrow \emptyset$
  for each $X_i \in RHS(p)$
    for each $(x,k) \in A_t(X_i)$
      $AP \leftarrow AP \cup \{(x, Card(x, AP) + 1)\}$
where $Card(x, AP)$ is the number of indexed attributes (x, j) in AP with the same type x

function $FP(p)$
  $p : X_0 \rightarrow X_1 \ldots X_n$
  $FP \leftarrow \emptyset$
  for each x, such that $\exists (x,k) \in A_t(X_0)$
    for each k in increasing order
      $FP \leftarrow FP \cup \{f_{p,(x,k)} = Copy(X_i.(x,j))\}$
where the occurrence attribute $X_i.(x,j)$ is the "leftmost"[9]
which is not used in the LHS of any copy rule already in $FP(p)$.

predicate *Construction(p)*
  Cardinal of *AP(p)* is equal to the length of $RHS(p_o)$.

(i,j) is the smallest couple using lexicographic order.

Fig. 2. The AP and FP functions and the *Construction* predicate.

the initial constraint, the total number of indexed attributes at the leaves of t is equal to the length of $RHS(p_o)$; as soon as we have this number of indexed attribute occurrences, we know that the construction is possible.

Elementary step on a node u of t We note r the root of t and $\{(a,1)\}$ the associated attribute singleton $A_t(label(r))$. Then,

1. If u is the root of t:
   (a) If *Construction(prod(u))* is true:

$$F_t \leftarrow F_t \cup \{f_{prod(u),(a,1)} = p_o(\ldots)\}$$

(b) Otherwise construction of α fails.

2. If $A_t(label(u)) = \emptyset$:

$$A_t(label(u)) \leftarrow AP(prod(u));\ F_t \leftarrow F_t \cup FP(prod(u))$$

3. If $A_t(label(u)) \neq \emptyset$:
   (a) If $AP(prod(u)) = A_t(label(u))$:

$$F_t \leftarrow F_t \cup \{FP(prod(u)\}$$

   (b) Otherwise construction of α fails.

Recall that the construction of α also fails if any addition to F_t results in several distinct semantic rules for the same attribute occurrence in the same production.

Our algorithm gives the unique solution in which construction of p_o is performed on the root production. In fact, it's easy to modify this elementary step to produce a set of solutions. Each solution is attached to a position in tree where the construction of p_o is possible. This is discussed in section 4.1.

Final instantiation The final step is to regroup all the A_t, F_t in a correct way, so as to produce the A_α and F_α which define a correct coupling attribute grammar.

For each non-terminal, we check if it has the same indexed attributes declarations in each A_t. For each production, we check if it has the same semantic rules in each F_t.[10] If so, the construction of the coupling AG is feasible and is not much more than the union of the A_t and F_t. The only difference lies with recursive neutral non-terminals eliminated during the second step of recursion elimination in the validation algorithm. These non-terminals had been reduced to their equivalence classes. For each of these classes, there is an unique set of declared attributes. For each non-terminal of a same class we declare the indexed attributes set of the class. For each production eliminated in step 4 of the recursion elimination algorithm, we generate the copy rules from all the attributes in the RHS to the LHS. Indeed, it is the only acceptable semantics for these eliminated productions.

If this cross-checking succeeds, we have consistent declarations of indexed attributes on each element of $N_i \cup T_i$ and a set of semantic rules on each production of G_i. Now we must transform these sets to specify a classical attribute grammar with classical attributes. In addition, the base grammar must be the real input grammar G_i, including dead non-terminals and terminals. For non-terminals, we rename the indexed attribute (a, i) to $a.i$ in declarations and in semantic rules. For the terminals T_i, we replace $(a, 1)$ in the semantic rules by the access to the value of T_i.

At the end of this construction, the generated coupling AG has the following properties:

1. it generates one tree of G_o, not a forest;

[10] Here we do not take into account empty sets.

2. each attribute occurrence is used only once in a production, which implies that real trees are built, not dags;
3. it is a purely synthesised AG.

These properties are the least constraining to apply meta-composition.

4 Extensions to the Basic Algorithm

4.1 Multiple solutions

In the basic algorithm we present for each tree the computation of the unique possible solution with construction of p_o on the root production. In fact it's sometimes possible to perform the construction lower in the tree. The condition for p_o construction to be possible on some production is that all attributes representing the RHS of p_o are available on it. The attribute representing LHS is then transferred to the root.

For a given tree t we have now a set $S(t)$ of possible solutions. We try to find a solution in each $S(t)$ which is compatible with the other $S(t')$. In fact we try each possible combination.

4.2 Permutation mapping

In the basic algorithm we require each output virtual production to be exactly a real production in P_o; this means in particular that the order of the symbols in the RHS is kept unchanged.

We could have chosen a more permissive strategy, requiring for instance that for each output virtual production p there exists in P_o a real production p', the RHS of which is a permutation of the RHS of p. Possible ambiguities could be resolved by choosing the real production with the "closest" RHS.

4.3 Root constraint

Now we want to relax the constraint $Cor(Z_o) = \{Z_i\}$. In fact, for a given tree t of G_i, we would like to instantiate the gene on different subtrees of t. So we accept that the generated coupling attribute grammar α constructs a forest, provided that, for each tree of the forest, the type of the root is Z_o. So we must check the following condition:

$$\forall X \in N_k \ \textit{such that}\ Z_i \underset{Cor}{\overset{*}{\Longrightarrow}} X, X \in Cor(Z_o)$$

This new approach mandates a new definition of neutral non-terminals:

$$N_n = \{X \in N_i - N_k \mid \exists Y \in N_k - Cor(Z_o), X \overset{*}{\Longrightarrow} Y\}$$

Moreover we add to the dead symbols elimination step the elimination of non-terminals in $Cor(Z_o)$ appearing in the RHS of productions; this ensures that

output trees are disconnected. Apart from that nothing is changed in the construction of the coupling grammar.

We need to adapt meta-composition such that it accepts this kind of forest construction. As each tree of the forest has a Z_o root and Z_o, being the start symbol of G_o, is assumed to have no inherited attribute, this adaptation does not cause any difficulty [10].

4.4 Extension of instantiation

We have described a instantiation associating a closed tree to a single production, we now want to extend this to an association between closed trees and output subtrees of height greater than one.

Starting with a closed tree, the first step is to construct the corresponding *output virtual production* as above. Then, its RHS is seen as a sentential form of G_o, whose derivation tree will be associated with the input closed tree. Constructing this derivation tree is easy with a slightly transformed parser.

The second step is to construct the coupling AG. It is easy to modify the algorithm in section 3.2 to build the whole output tree at a single node of the input closed tree. We are currently investigating whether it is feasible to spread this construction over the closed tree.

The third step of merging the various pseudo-AGs is unchanged.

5 Future Work and Conclusion

Recall that our main goal was to free the user from the dull and error-prone task of specifying by hand the details of the translation between two grammars. Our notion of grammar coupling clearly fulfills this goal. It remains to experiment with it in order to assess its practical value as support for modularity and reusability in AGs. We have the strong intuition that, in spite of its feasibility constraints, the power of our association mechanism remains realistic for users. It is essentially on this point that we have a more attractive approach than the one described in [4], in which the specification of this translation is completely left to the user.

The algorithm presented above is a first contribution to our plan to implement a tool supporting modularity and genericity in the framework of the FNC-2 AG system [6]. Already, meta-composition is implemented in a coupling module [11]. We are starting the implementation of our instantiation algorithm in the FNC-2 context. This mainly entails extending it to abstract syntaxes.

Beforehand, we had studied whether our approach of genericity for AGs was realistic (see [9]), and our experience of AG specification with FNC-2 (more than 50,000 lines of AG descriptions) confirms that this is indeed the case.

We hope that, in the near future and with this new notion of modularity in AGs, users will be able to specify large applications from a library of generic AGs. Then attribute grammars will become still more usable for real-world applications.

References

1. Baum, B. *Another Kind of Modular Attribute Grammars.* In *4th Int. Conf. on Compiler Construction*, U. Kastens and P. Pfahler, Eds. Lect. Notes in Comp. Sci., vol. 641, Springer-Verlag, New York–Heidelberg–Berlin, Oct. 1992, 44–50.
2. Deransart, P., Jourdan, M. and Lorho, B. *Attribute Grammars: Definitions, Systems and Bibliography*. Lect. Notes in Comp. Sci., vol. 323, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.
3. Dueck, G. D. P. and Cormack, G. V. Modular Attribute Grammars. *Comput. J. 33* (1990), 164–172, See also: research report CS-88-19, Univ. of Waterloo (May 1988).
4. Farrow, R., Marlowe, T. J. and Yellin, D. M. *Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation.* In *19th ACM Symp. on Principles of Progr. Languages.* Jan. 1992, 223–234.
5. Ganzinger, H. and Giegerich, R. Attribute Coupled Grammars. *ACM SIGPLAN Notices 196* (June 1984), 157–170.
6. Jourdan, M., Parigot, D., Julié, C., Le Bellec, C. and Durin, O. *Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System.* In *ACM SIGPLAN '90 Conf. on Progr. Languages Design and Implementation.* ACM SIGPLAN Notices, vol. 25, no. 6, July 1990, 209–222.
7. Kleinhans, C. Intégration de la modularité dans les grammaires attribuées. Dépt. d'Informatique, Univ. de Nice, Thèse, Feb. 1991.
8. Knuth, D. E. Semantics of Context-free Languages. *Math. Systems Theory 22* (June 1968), 127–145, Correction: *Math. Systems Theory 5*, 1, pp. 95-96 (Mar. 1971)..
9. Le Bellec, C. La généricité et les Grammaires Attribuées. Dépt. d'Informatique, Univ. d'Orléans, Thèse non encore soutenue.
10. Roussel, G. Différentes Transformations de Grammaires Attribuées. Dépt. d'Informatique, Univ. de Paris VI, Thèse non encore soutenue.
11. Roussel, G. A Transformation of Attribute Grammars for Eliminating Useless Copy Rules. INRIA, Research report to come.
12. Soisalon-Soininen, E. and Wood, D. On a Covering Relation for Context-Free Grammars. McMaster University, report 80-CS-21, Hamilton, Ontario, Canada, Sept. 1980.

This article was processed using the LaTeX macro package with LLNCS style