Static and Dynamic Coupling Attribute Evaluators

Gilles Roussel, Didier Parigot, Martin Jourdan

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Static and Dynamic Coupling Attribute Evaluators

Gilles ROUSSEL , Didier PARIGOT , Martin JOURDAN

N° 2670

Octobre 1995

PROGRAMME 2

ISSN 0249-6399

INRIA
ROCQUENCOURT

Static and Dynamic Coupling Attribute Evaluators

Gilles Roussel , Didier Parigot , Martin Jourdan

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet charme

Rapport de recherche n° 2670 — Octobre 1995 — 13 pages

Abstract: Several years ago, the notion of attribute coupled grammars was introduced by Ganzinger and Giegerich [5], together with their descriptional composition. The latter works essentially at the specification level, i.e., it produces an attribute grammar which specifies the composition of two attribute coupled grammars.

We introduce a new approach to this composition of attribute coupled grammars. It no longer works at the specification level but rather at the evaluator level. It produces a special kind of attribute evaluator, called *coupling evaluator*. We present both a static version and a dynamic version of coupling evaluators. Both versions retain the good property of descriptional composition that intermediate trees are not physically constructed.

In addition—and this is the main advantage of our approach, compared with descriptional composition—, it is possible to build separately the dynamic coupling evaluator of each attribute coupled grammar; in other words we achieve real *separate compilation* of AG modules.

Key-words: Attribute grammars, Composition, Separate Compilation

(Résumé : tsvp)

Ce travail a été partiellement financé par le projet ESPRIT #5399 "Compare".

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Evaluateur d'Attributs de Couplage Statiques et Dynamiques

Résumé :

Les grammaires couplées par attributs ont été introduites par Ganzinger et Giegerich [5], avec la notion de composition descriptionnelle. Cette composition s'effectue essentielement au niveau de la spécification, c'est-à-dire qu'elle produit une nouvelle grammaire attribuée qui spécifie la composition des deux grammaires couplées.

Nous introduisons une nouvelle approche de cette composition de grammaire couplées par attributs. Elle ne s'effectue pas au niveau de la spécification mais plutôt au niveau des évaluateurs d'attributs. Elle produit un autre type d'évaluateur d'attributs, appelé évaluateur de couplage. Nous présentons à la fois une version statique et version dynamique de ces évaluateurs de couplage. Ces deux versions conservent la bonne propriété de la composition descriptionnelle, qui est que l'arbre intermédaire n'est pas construit physiquement.

De plus — c'est l'un des principaux avantages de notre approche, comparée à la composition descriptionnelle —, il est possible de construire séparément les évaluateurs de couplage. Autremant dit, nous aboutisons réellement à une compilation séparée des modules de grammaires attribuées.

Mots-clé : Grammaires Attribuées, Composition, Compilation Séparée

1 Introduction

1.1 Attribute couplings and descriptional composition

Since Knuth's seminal paper introducing attribute grammars (AGs) [12], it has been widely recognized that this method is quite attractive for specifying most kinds of syntax-directed computations, the most obvious application being compiler construction. Apart from pure specification-level features—declarativeness, strong 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 heavily used to develop practical, production-quality, applications.

Unfortunately, it appears that 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 [2] (e.g., FNC-2 [10], which is the base of the present work `http://www-rocq.inria.fr/charme/FNC-2/`). In our opinion [9], the main reason for this is that AGs still cruelly lack the same support for modularity as the one which is offered by most programming languages, even the oldest ones.

This is the reason why *attribute coupled grammars* (which we call *attribute couplings*[5](ACs) in the remainder of this paper) were introduced. They allow modularity in AG specifications. An application can be decomposed into several attribute couplings, each of which transforms an input syntax tree into a new output syntax tree. This has been widely recognized as a very important concept for attribute grammar modularization.

As separate compilation of each AC into a classical evaluator leads to a loss of efficiency compared to non-modular specifications (because of explicit constructions of the intermediate trees), descriptional composition has been introduced to create, from a sequence of attribute couplings, a single, large attribute coupling which performs the same translation but avoids any intermediate tree construction.

1.2 Towards separate compilation of ACs

We have implemented descriptional composition in our system based on Strongly Non-Circular (SNC) attribute grammars [10]. However, it is not quite satisfactory. In particular, the SNC class is not closed under descriptional composition [7]: the attribute grammar resulting from the composition of SNC modules is not necessarily SNC, which would make it unusable with FNC-2. Furthermore, in the context of attribute grammar reuse [4], descriptional composition is hard to use since it requires complete reconstruction of the resulting AG before any evaluator construction: separate compilation is impossible.

These observations and our personal work on attribute grammar reusability [8,13] have led us to introduce a new technique allowing separate compilation of modules, as in [4], but with no intermediate tree construction. This technique is based on the simple but powerful idea of descriptional composition, i.e., attaching computations of an AC to the tree construction actions of the AC which precedes it in a sequence. The main difference between descriptional composition and our technique is that we do not work at the specification level but at the evaluator level. To do so, we introduce new sorts of evaluators, called *coupling evaluators* and *parameterizable evaluators*.

Given a sequence of ACs, we construct a parameterizable evaluator for the last AC. This evaluator is very similar to a classical evaluator, except that it only sees one special production instance of its input tree at each time. When placed at the end of a sequence, when it wants to move in its input tree (change to a neighboring node, i.e. another production instance), it queries the "visit selector" mechanism which performs the move in the input tree of the first grammar in the sequence. The result is a new production instance which is passed through the coupling evaluators attached to the different ACs of the sequence. Each of them transforms its input production instance into a new production instance for its output grammar. The last created production instance is the production instance that the parameterizable evaluator requires.

Some will see similarities with the deforestation algorithm for functional languages. Duris [3] shows that with some restrictions we can compare the deforestation algorithm and the descriptional composition algorithm, but the objects (attribute grammars and functions) on which these algorithms apply can not be directly use one for the other. This is not the purpose of this paper so will not discuss this further.

1.3 Paper overview

To make our approach of ACs composition understandable, we explain it in a very informal way using examples. Our running example will be the list reversal AG [6], which we have rephrased using our notations in Fig. 1; this AC produces a reversed copy of its input list. Using this example we describe the different views of AC composition: descriptional composition (section 2.2), static coupling evaluators (section 3.1) and dynamic coupling

$$
\begin{array}{lllll}
root: & Z \rightarrow L & L.h := nil(); & & Z.z := root(L.s). \\
cons: & L_0 \rightarrow D\ L_1 & L_1.h := cons(D, L_0.h); & & L_0.s := Id(L_1.s). \\
nil: & L \rightarrow & L.s := Id(L.h). & &
\end{array}
$$

Added production for copy rules:

$$Id: \quad L_0 \rightarrow L_1 \quad L_0.s := Id(L_1.s); \quad L_1.h := Id(L_0.h).$$

Figure 1: Attribute coupling specifying list reversal

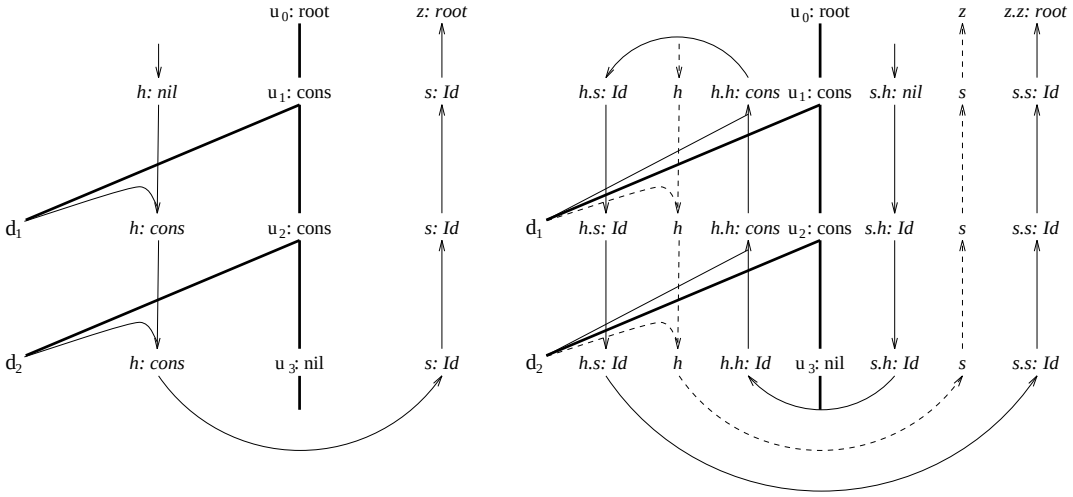

Figure 2: List reversal and double list reversal in action

evaluators (section 3.2). In addition, section 2.3 discusses related work and further motivates our approach, while section 4 highlights various properties of our technique.

2 Attribute Couplings and Descriptional Composition

2.1 The list reversal AG and its evaluator

In the list reversal example (Fig. 1), the input and the output grammars are the same list grammar. In this AC, the construction of the reversed list is performed using an inherited attribute h, the synthesized attribute s being here only to bring back the result to the *root* production using copy rules (noted Id). D is a terminal of a not-further-defined type.

We have completed the list reversal AG with a new production Id. It is added for composition purposes; indeed, with this new production, copy rules can be handled just as tree construction rules. Of course, this added production is viewed in the evaluator as a copy rule, i.e., $L_0.s := Id(L_1.s)$ really means $L_0.s := L_1.s$.

Fig. 2 illustrates the dependencies and operation of the list reversal AG on a 2-element list; node names (u_i and d_j) will be used in subsequent examples.

To complete this introduction, we give in Fig. 3 an attribute evaluator for the list reversal AG, based on visit sequences [11]. Informally, the *visit* function means that the evaluator is recursively called on an other node. The visit name is indexed by the corresponding non-terminal and the production name; when called on some node, the evaluator fetches which production is applied at it to select the corresponding visit sequence. The *eval_a_X* function calls the semantic rule defining the attribute occurrence $X.a$ on the current production instance. It gives a value to the instance of $X.a$ in the current production.

Now, to implement the composition of this AG with itself, the most naive method is to run the evaluator twice, first on the input list, yielding a reversed copy, then on this reversed copy, yielding a copy of the original list. The rest of this paper is devoted to more efficient methods to achieve the same result.

$root:\quad visit\_Z\_root(u):\quad eval\_h\_L(son(1,u)); visit\_L(son(1,u)); eval\_z\_Z(u).$
$cons:\quad visit\_L\_cons(u):\quad eval\_h\_L_1(son(2,u)); visit\_L(son(2,u)); eval\_s\_L_0(u).$
$nil:\quad visit\_L\_nil(u):\quad eval\_s\_L(u).$

Added visit for copy rules:

$Id:\quad visit\_L\_Id(u):\quad eval\_h\_L_1(son(1,u)); visit\_L(son(1,u)); eval\_s\_L_0(u).$

Figure 3: Attribute evaluator performing list reversal

For the *cons* production, the semantic rule $L_1.h := cons(D, L_0.h)$ induces
the projection of the following semantic rules:
$L_1.h := cons(D, L_0.h)$ projected, yields $L_0.h.h := cons(D, L_1.h.h)$
$L_0.s := Id(L_1.s)$ projected, yields $L_1.h.s := Id(L_0.h.s)$

Figure 4: An example of Descriptional Composition

$root:$	$Z \rightarrow L$	$L.h.s := Id(L.h.h);$ $L.s.h := nil().$	$Z.z.z := root(L.s.s);$
$cons:$	$L_0 \rightarrow D\ L_1$	$L_0.h.h := cons(D, L_1.h.h);$ $L_0.s.s := Id(L_1.s.s);$	$L_1.s.h := Id(L_0.s.h);$ $L_1.h.s := Id(L_0.h.s).$
$nil:$	$L \rightarrow$	$L.s.s := Id(L.h.s);$	$L.h.h := Id(L.s.h).$

Figure 5: Descriptional composition of the list reversal AC with itself

2.2 Descriptional composition of attribute couplings

Our notion of coupling evaluator is strongly related to descriptional composition. We give a brief idea of the latter through the composition of the list reversal AG with itself.

Descriptional composition takes two ACs and mechanically constructs a new AC, the semantics of which is equivalent to the functional composition of the given ACs. It is performed statically at the source level (on ACs) and eliminates the intermediate tree construction.

The basic idea of descriptional composition is to project on the productions of the first AC, the set of semantic rules of a given production of the second AC each time it appears as a tree construction operation in a semantic rule of the first AC. In the resulting AC, new attributes are created. These attributes are named using an attribute of the first AC and an attribute of the second; in Fig. 4 for instance, the attribute $s.h$ is built from s in the first AC and h in the second. A more detailed explanation and complete example can be found in [5]. Note that the semantic rules in the result AC are exactly those of the second given AC, where attributes have been renamed.

The AC resulting from the descriptional composition of the list reversal AC of Fig. 1 with itself is presented in Fig. 5. Its dependencies and operation are illustrated in Fig. 2.

In this figure, the dashed arrows represent the intermediate construction (the fisrt AG). The solid arrows represent the projection of the semantic rules (of the second AG) according to the dashed construction. With elimination of the dashed arrows, we obtain the descriptional composition resulting AG.

As expected, it constructs a copy of the input list, albeit in a rather complicated manner.

For this resulting AC, the attribute evaluator is given in the Fig. 6. This evaluator is composed of two visits, the first one (*visit_L_1*) performs the list construction (in the same order as the input list) and the second one (*visit_L_2*) merely carries this list around the tree through copy rules.

Descriptional composition has been implemented in the FNC-2 framework. We have devised extensions to the original algorithm [14] regarding conditional expressions and nested tree construction. In addition, the previous example shows that some optimizations can and must be made on the AC resulting from descriptional composition. Indeed, the second visit can be removed. In FNC-2, an optimizer performing these optimizations has been implemented [14]. It statically analyses the copy rules chains between attributes in a manner similar to the SNC dependency construction. Then the AC is transformed to shortcut some of these chains. For instance, in the previous resulting AC, all the attribute dependencies which induce the second visit are removed.

$$
\begin{array}{lll}
root: & visit\_Z\_root(u): & eval\_s.h\_L(son(1,u)); visit\_L\_1(son(1,u)); eval\_h.s\_L(son(1,u)); \\
 & & visit\_L\_2(son(1,u)); eval\_z.z\_Z(u). \\
cons: & visit\_L\_cons\_1(u): & eval\_s.h\_L_1(son(2,u)); visit\_L\_1(son(2,u)); eval\_h.h\_L_0(u). \\
 & visit\_L\_cons\_2(u): & eval\_h.s\_L_1(son(2,u)); visit\_L\_2(son(2,u)); eval\_s.s\_L_0(u). \\
nil: & visit\_L\_nil\_1(u): & eval\_h.h\_L(u). \\
 & visit\_L\_nil\_2(u): & eval\_s.s\_L(u).
\end{array}
$$

Figure 6: Attribute evaluator performing descriptional composition of the list reversal AC with itself

2.3 Related work and motivation

In [4], the authors present the notion of separable AGs. They want to have separate compilation of these AGs, hence they do not use descriptional composition. They propose an adaptation of the classical evaluation scheme which allows separate compilation, but the construction of (some) intermediate trees is necessary. Boyland and Graham [1] have attempted to follow up on this track.

In contrast, we wanted to have *both* the separate compilation of evaluators and the good property of descriptional composition that it avoids the construction of any intermediate tree. This was motivated by our personal work on attribute grammars reusability [8,13], which makes heavy use of the notion of attribute coupling. To make it practical we needed to have both properties at the same time. We also needed to overcome the non-closure problems of descriptional composition [7].

Our technique is based on the simple but powerful idea of descriptional composition, i.e. to attach the computations of an attribute grammar to the tree construction actions of the attribute coupling which precedes it in a sequence. The main difference between descriptional composition and our technique is that we do not work at the specification level but rather at the evaluator level. As we will show, this allows to reach our goal.

In [7], the author explains how to directly construct, given all the evaluators of an AC sequence, an evaluator associated to the AC resulting from the descriptional composition of the sequence. More precisely, "directly" means that there is no need to compute the attribute dependencies for the composed AC. However the need to completely construct the evaluator still exists; furthermore this approach only applies to purely synthesized syntactic ACs.[1] Our approach has some similarities with this construction, except that we work directly on the evaluators and we accept non-purely-synthesized ACs.

To come back to [4], then, from the external point of view, both their work and ours appear to have similar capabilities (separate compilation). However our technique is more efficient, since we do not construct any intermediate tree, and more powerful, since we have no limitation regarding the AG class.

3 Coupling Evaluators

We will decompose our presentation into two steps: first we present the static view of coupling evaluators and then the evolution of this approach, the dynamic view.

3.1 Static coupling evaluators

The basic idea of static coupling evaluators is very similar to that of descriptional composition, except that it works on evaluators rather than on ACs and that it projects visit sequences instead of semantic rules. Thus this new method no longer constructs an AC specification but a new attribute evaluator. This attribute evaluator is an alternate evaluator for the (virtual) AC resulting from the descriptional composition. The construction of this attribute evaluator requires neither the construction of that AC nor the call to the classical evaluator generator.

The evaluators resulting from this construction use the same kind of visit sequences as the incremental DNC attribute evaluators which can be generated by the FNC-2 system [10]. Top-down visits move to a son of the current node, whereas bottom-up visits move to the parent.

The static coupling evaluator for a sequence of two ACs is built by projecting on the visit sequences of the evaluator of the first AC the set of visits of a given production of the evaluator of the second AC each time the production appears as a tree construction operation in a semantic rule of the first AC. The visit renaming process is similar to the attribute renaming process of descriptional composition.

[1] See proposition 8 in [7] for more details.

On the *cons* production, by the *cons* construction, the projection of the *visit_L_cons* visit induces the following visit:

$$cons: \quad visit\_\uparrow\_L\_h\_cons\_2(u): \quad eval\_h.h\_L_0(father(u)); visit\_\uparrow\_L\_h(father(u)); eval\_h.s\_L_1(u).$$

On the *id* construction, the projection of the *visit_L_id* visit induces the following visit:

$$cons: \quad visit\_\downarrow\_L\_s\_cons\_0(u): \quad eval\_s.h\_L_1(son(2,u)); visit\_\downarrow\_L\_s(son(2,u)); eval\_s.s\_L_0(u).$$

Figure 7: An example of visit projection

$$\begin{array}{lll}
root: & visit_\downarrow_Z_z_root_0(u): & eval_s.h_L(son(1,u)); visit_\downarrow_L_s(son(1,u)); eval_z.z_Z(u). \\
 & visit_\uparrow_L_h_root_1(u): & eval_h.s_L(u). \\
cons: & visit_\downarrow_L_s_cons_0(u): & eval_s.h_L_1(son(2,u)); visit_\downarrow_L_s(son(2,u)); eval_s.s_L_0(u). \\
 & visit_\uparrow_L_h_cons_2(u): & eval_h.h_L_0(father(u)); visit_\uparrow_L_h(father(u)); eval_h.s_L_1(u). \\
nil: & visit_\downarrow_L_h_nil_0(u): & eval_h.h_L(u); visit_\uparrow_L_h(u); eval_s.s_L(u).
\end{array}$$

Figure 8: Static coupling evaluator for the composition of the list reversal AC with itself

Once again we take as example the projection for list reversal AC on the *cons* production, see Fig. 7. The name of a visit contains the name of the attribute of the first AC on which the construction has been performed. If this attribute was an inherited attribute, then the projected visit is a bottom-up visit (indicated by $\uparrow$), else it is top-down ($\downarrow$). For a bottom-up visit, as it could come from any of the right-hand-side nodes of a production, the name of the visit also contains the position of this node in the production. As with descriptional composition, the semantic rules used in the evaluator are the same as those of the second AC, with attributes renamed; in consequence, the *eval* operations in Fig. 7 and Fig. 8 are the same as those in Fig. 6.

The complete resulting evaluator (see Fig. 8) is very different from the classical evaluator associated with the AC resulting from the descriptional composition (see Fig. 6). However, they have the same semantics. The latter makes two top-down visits while the former makes a top-down visit followed by a bottom-up visit.

3.2 Dynamic Coupling Evaluators

To achieve separate compilation we extend the static version of coupling evaluators into a dynamic version. With this new version it is possible to compose any (statically unknown) sequence of ACs in one operation. More precisely, for each AC we statically construct a dynamic coupling evaluator which can be used in any sequence. In our example we limit ourselves to the composition of two ACs, but it is only for convenience.

The evaluator for the whole sequence is composed of a *filter* for each AC in the sequence but the last, and a *parametrizable evaluator* for the last AC. A parametrizable evaluator is the same as the corresponding classical evaluator except that it works on *virtual nodes*, *virtual productions* and *virtual attributes* instead of real ones. All these notions are explained below; their relationship to one another is depicted in Fig. 9, which illustrates our new evaluation model and compares it to the classical one.

It is important to note here that both filters and parametrizable evaluators can be constructed separately for each AC and used in any AC sequence; thus, we *do* have separate compilation.

Instead of statically performing the projection of the visits on construction rules, we do it dynamically through filters. So, for the first AC we construct a filter which translates any real production of the first grammar into some virtual production of the second. On this virtual production the evaluator of the second AC can work. A virtual production is an object that an evaluator can consider as a production of a real tree of its input grammar. The filter translates a real production into a virtual production of the second grammar according to a given attribute occurrence. There is one possible translation in the filter for each attribute occurrence in the production.

As filters can be called in a sequence we need to use a more complex object than a simple attribute occurrence. This object, called virtual attribute, is a sequence of attributes of the form $a.b.c$ when the sequence is composed of three filters. Thanks to this virtual attribute, it is possible to find the attribute occurrences which guide each filter. More precisely, to find one of these guiding attribute occurrences, we need the special "node" in the production which represent the calling node of the visit sequence. We call it the virtual node. This one is deduced from a real node in the input tree and the virtual attribute.

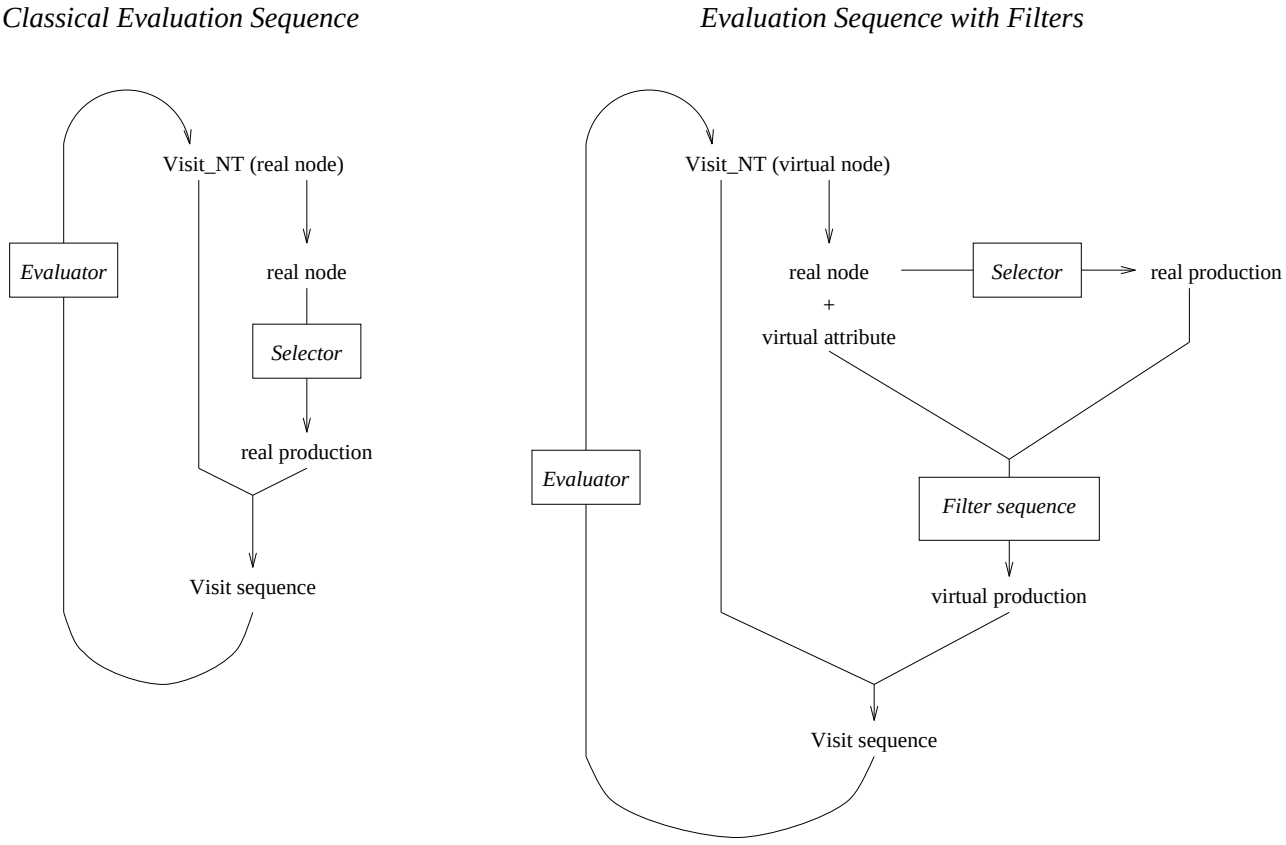

Figure 9: Operation of a classical and a dynamic coupling evaluator

For example, we explain here the translation of the *cons* production filter for the attribute occurrence $L_1.h$. The input node is denoted v_1 and the virtual attribute is of the form $h.w$ (in a sequence with only one filter, i.e., two ACs, w is always ϵ, i.e., empty); this corresponds to the case v_1, h in the filter. The semantic rule corresponding to this attribute occurrence is $v_1.h := cons(D, v_0.h)$ (see Fig. 1). So the translation gives the virtual production $cons : v_1.h \rightarrow D\ v_0.h$. The new virtual node in this production is $v_1.h$. This corresponds to the following line of the *cons* production filter:

$$v_1, h : \langle v_1.h, w, cons : v_1.h \rightarrow D\ v_0.h \rangle$$

On this example we can easily see that each filter eliminates the first attribute of the input virtual attribute to give the resulting virtual attribute. The eliminated attribute has been used to guide the filter.

The virtual production keeps track of all the attribute occurrences it originates from. These attributes will be used to obtain further virtual attributes.

The formal presentation of these notions is presented in [15]. Fig. 10 gives the complete set of filters for the list reversal AC.

The size of the data structures needed to run the filters is exactly the same as used in the AC resulting from the descriptional composition. We insist on the fact that the virtual productions do not exist physically, except through a constant-space coding.

To implement the dynamic composition of a sequence of ACs, we must call the sequence of coupling evaluators ended by the real evaluator of the last AC. The process is initialized with the root node and the virtual attribute which is composed of the sequence of attributes which carry the results of the different ACs in the sequence (of the form $z.z.z$). In Fig. 11 we give a complete execution trace of such a call on the same two-element list as in Fig. 2.

We detail here the interpretation of the first four lines of this trace. The dynamic coupling evaluator for list reversal is called "from above" (see below) on the root node of the input tree with virtual attribute z. The evaluator and selector mechanism (**E**) translates this virtual node $u_0.z$ into the real node u_0 and virtual

For a given tuple of form $\langle v_j, a.w, p\rangle$ the coupling evaluator has the following form:

$$
\begin{array}{ll}
\textit{case } p \textit{ is} & \\
\quad root : Z \rightarrow L & \langle v_j, a.w, root : v_0 \rightarrow v_1\rangle \\
 & \textit{case } v_j, a \textit{ is} \\
 & v_0, z : \langle v_0.z, w, root : v_0.z \rightarrow v_1.s\rangle \\
 & v_1, s : \langle v_1.s, w, root : v_0.z \rightarrow v_1.s\rangle \\
 & v_1, h : \langle v_1.h, w, nil : v_1.h \rightarrow\rangle \\
 & \\
\quad cons : L_0 \rightarrow D\ L_1 & \langle v_j, a.w, cons : v_0 \rightarrow D\ v_1\rangle \\
 & \textit{case } v_j, a \textit{ is} \\
 & v_0, s : \langle v_0.s, w, Id : v_0.s \rightarrow v_1.s\rangle \\
 & v_1, s : \langle v_1.s, w, Id : v_0.s \rightarrow v_1.s\rangle \\
 & v_1, h : \langle v_1.h, w, cons : v_1.h \rightarrow D\ v_0.h\rangle \\
 & v_0.h : \langle v_0.h, w, cons : v_1.h \rightarrow D\ v_0.h\rangle \\
 & \\
\quad nil : L \rightarrow & \langle v_j, a.w, nil : v \rightarrow\rangle \\
 & \textit{case } v_j, a \textit{ is} \\
 & v, s : \langle v.s, w, Id : v.s \rightarrow v.h\rangle \\
 & v, h : \langle v.h, w, Id : v.s \rightarrow v.h\rangle \\
 & \\
\quad Id : L_0 \longrightarrow L_1 & \langle v_j, a.w, Id : v_0 \rightarrow v_1\rangle \\
 & \textit{case } v_j, a \textit{ is} \\
 & v_0, s : \langle v_0.s, w, Id : v_0.s \rightarrow v_1.s\rangle \\
 & v_1, s : \langle v_1.s, w, Id : v_0.s \rightarrow v_1.s\rangle \\
 & v_0, h : \langle v_0.h, w, Id : v_1.h \rightarrow v_0.h\rangle \\
 & v_1, h : \langle v_1.h, w, Id : v_1.h \rightarrow v_0.h\rangle
\end{array}
$$

Figure 10: Dynamic Coupling Evaluator for the List reversal AC

attribute z; since we have the real node, we also have the corresponding real production *root*:

$$(\mathbf{E}) \quad visit\_Z(u_0.z) \rightarrow \langle u_0, z, root : u_0 \rightarrow u_1\rangle$$

The filter (**F**) translates this tuple as in Fig. 10:

$$(\mathbf{F}) \quad \langle u_0, z, root : u_0 \rightarrow u_1\rangle \rightarrow \langle u_0.z, \epsilon, root : u_0.z \rightarrow u_1.s\rangle$$

In this new virtual production, the virtual attribute is empty, so the real evaluator can use it just as any real production. Here it instantiates the visit sequence (**VS**) of the *root* production found in the tuple (the visit sequence appears in Fig. 3) with the virtual nodes which form the virtual production:

$$(\mathbf{VS}) \quad visit\_Z\_root(u_0.z) : eval\_h\_L(u_1.s); visit\_L(u_1.s); eval\_z\_Z(u_0.z).$$

This sequence gives rise to two (virtual) attribute computations (instantiations of the semantic rules (**SR**) in Fig. 1):

$$(\mathbf{SR}) \quad u_1.s.h := nil(); u_0.z.z := root(u_1.s.s)$$

and to a further visit "from above", $visit\_L(u_1.s)$, which is carried out in the same way.

The only other remark to make on this figure is when the evaluator reaches $visit\_L(u_2.h)$ while it is on the real production $nil : u_2 \rightarrow$. The real node to visit, u_2, is the LHS of this real production, so it will be visited "from below"; hence, the associated tuple returned by the evaluator/selector mechanism, $\langle u_2, h, cons : u_1 \rightarrow D\ u_2\rangle$, contains the production above u_2.

4 Discussion and Extensions

One could argue that this dynamic coupling evaluator is quite expensive, because each visit to a node provokes a call to the filter sequence, where a normal evaluator would simply look up the production from the visited node. That's true, but that's the price to pay for dynamic coupling, just as virtual method lookup in object-oriented languages is expensive. However, this can be alleviated by the same techniques: a form of caching/memoization

The first call is made on the root node u_0 with the virtual attribute z:

$$
\begin{array}{c}
(\mathbf{E}) \quad visit\_Z(u_0.z) \rightarrow \langle u_0, z, root : u_0 \rightarrow u_1 \rangle \\
(\mathbf{F}) \quad \langle u_0, z, root : u_0 \rightarrow u_1 \rangle \rightarrow \langle u_0.z, \epsilon, root : u_0.z \rightarrow u_1.s \rangle \\
(\mathbf{VS}) \quad visit\_Z\_root(u_0.z) : eval\_h\_L(u_1.s); visit\_L(u_1.s); eval\_z\_Z(u_0.z). \\
(\mathbf{SR}) \quad u_1.s.h := nil(); u_0.z.z := root(u_1.s.s) \\
- * - \\
(\mathbf{E}) \quad visit\_L(u_1.s) \rightarrow \langle u_1, s, cons : u_1 \rightarrow d_1 \ u_2 \rangle \\
(\mathbf{F}) \quad \langle u_1, s, cons : u_1 \rightarrow d_1 \ u_2 \rangle \rightarrow \langle u_1.s, \epsilon, Id : u_1.s \rightarrow u_2.s \rangle \\
(\mathbf{VS}) \quad visit\_L\_Id(u_1.s) : eval\_h\_L_1(u_2.s); visit\_L(u_2.s); eval\_s\_L_0(u_1.s). \\
(\mathbf{SR}) \quad u_2.s.h := u_1.s.h; u_1.s.s := u_2.s.s \\
- * - \\
(\mathbf{E}) \quad visit\_L(u_2.s) \rightarrow \langle u_2, s, cons : u_2 \rightarrow d_2 \ u_3 \rangle \\
(\mathbf{F}) \quad \langle u_2, s, cons : u_2 \rightarrow d_2 \ u_3 \rangle \rightarrow \langle u_2.s, \epsilon, Id : u_2.s \rightarrow u_3.s \rangle \\
(\mathbf{VS}) \quad visit\_L\_Id(u_2.s) : eval\_h\_L_1(u_3.s); visit\_L(u_3.s); eval\_s\_L_0(u_2.s). \\
(\mathbf{SR}) \quad u_3.s.h := u_2.s.h; u_2.s.s := u_3.s.s \\
- * - \\
(\mathbf{E}) \quad visit\_L(u_3.s) \rightarrow \langle u_3, s, nil : u_3 \rightarrow \rangle \\
(\mathbf{F}) \quad \langle u_3, s, nil : u_3 \rightarrow \rangle \rightarrow \langle u_3.s, \epsilon, Id : u_3.s \rightarrow u_3.h \rangle \\
(\mathbf{VS}) \quad visit\_L\_Id(u_3.s) : eval\_h\_L_1(u_3.h); visit\_L(u_3.h); eval\_s\_L_0(u_3.s). \\
(\mathbf{SR}) \quad u_3.h.h := u_3.s.h; u_3.s.s := u_3.h.s \\
- * - \\
(\mathbf{E}) \quad visit\_L(u_3.h) \rightarrow \langle u_3, h, cons : u_2 \rightarrow d_2 \ u_3 \rangle \\
(\mathbf{F}) \quad \langle u_3, h, cons : u_2 \rightarrow d_2 \ u_3 \rangle \rightarrow \langle u_3.h, \epsilon, cons : u_3.h \rightarrow d_2 \ u_2.h \rangle \\
(\mathbf{VS}) \quad visit\_L\_cons(u_3.h) : eval\_h\_L_1(u_2.h); visit\_L(u_2.h); eval\_s\_L_0(u_3.h). \\
(\mathbf{SR}) \quad u_2.h.h := cons(d_2, u_3.h.h); u_3.h.s := u_2.h.s \\
- * - \\
(\mathbf{E}) \quad visit\_L(u_2.h) \rightarrow \langle u_2, h, cons : u_1 \rightarrow d_1 \ u_2 \rangle \\
(\mathbf{F}) \quad \langle u_2, h, cons : u_1 \rightarrow d_1 \ u_2 \rangle \rightarrow \langle u_2.h, \epsilon, cons : u_2.h \rightarrow d_1 \ u_1.h \rangle \\
(\mathbf{VS}) \quad visit\_L\_cons(u_2.h) : eval\_h\_L_1(u_1.h); visit\_L(u_1.h); eval\_s\_L_0(u_2.h). \\
(\mathbf{SR}) \quad u_1.h.h := cons(d_1, u_2.h.h); u_2.h.s := u_1.h.s \\
- * - \\
(\mathbf{E}) \quad visit\_L(u_1.h) \rightarrow \langle u_1, h, root : u_0 \rightarrow u_1 \rangle \\
(\mathbf{F}) \quad \langle u_1, h, root : u_0 \rightarrow u_1 \rangle \rightarrow \langle u_1.h, \epsilon, nil : u_1.h \rightarrow \rangle \\
(\mathbf{VS}) \quad visit\_L\_nil(u_1.h) : eval\_s\_L(u_1.h). \\
(\mathbf{SR}) \quad u_1.h.s := u_1.h.h
\end{array}
$$

The semantic rules add up to the following equation system, sorted in evaluation order:

$$
\begin{array}{lcllcl}
u_1.s.h & := & nil() & u_2.h.s & := & u_1.h.s \\
u_2.s.h & := & u_1.s.h & u_3.h.s & := & u_2.h.s \\
u_3.s.h & := & u_2.s.h & u_3.s.s & := & u_3.h.s \\
u_3.h.h & := & u_3.s.h & u_2.s.s & := & u_3.s.s \\
u_2.h.h & := & cons(d_2, u_3.h.h) & u_1.s.s & := & u_2.s.s \\
u_1.h.h & := & cons(d_1, u_2.h.h) & u_0.z.z & := & root(u_1.s.s) \\
u_1.h.s & := & u_1.h.h & & &
\end{array}
$$

which shows that the final result is indeed a copy of the original list.

Figure 11: Example of evaluation trace

if we know nothing about the AC sequence, or a form of partial evaluation if we want to specialize the evaluators for a given sequence. The idea for the latter case is to "inline" the last filter into the final evaluator, then inline the previous to last filter into the resulting evaluator, etc. Thus, most of the filters clauses can be eliminated and a sequence of several filters clauses can often be reduced to a unique clause. Not surprisingly, the resulting evaluator is quite similar to the static coupling evaluator. We now give an example. The filter line:

$$\langle u_0, z, root : u_0 \rightarrow u_1 \rangle \rightarrow \langle u_0.z, \epsilon, root : u_0.z \rightarrow u_1.s \rangle$$

inlined into the evaluator visit:

$$root : visit\_Z\_root(u) : eval\_h\_L(son(1,u)); visit\_L(son(1,u)); eval\_z\_Z(u).$$

gives the new visit:

$$root : visit\_Z\_root(u_0.z) : eval\_h\_L(u_1.s); visit\_L(u_1.s); eval\_z\_Z(u_0.z).$$

which is very similar to the following visit of the static coupling evaluator:

$$root : visit\_ \downarrow \_Z\_z\_root\_0(u) : eval\_s.h\_L(son(1,u)); visit\_ \downarrow \_L\_s(son(1,u)); eval\_z.z\_L(u).$$

The main achievement of our dynamic coupling evaluator technique is of course separate compilation of attribute couplings, with nearly no limitation on the expressive power (see below). This breakthrough will undoubtedly make attribute grammars much more attractive as a programming tool for large applications, since it will strongly speedup the edit–compile–run cycle. In addition, when used in the various AG reusability frameworks [4,8,13], it allows to have libraries of pre-compiled, executable modules, just as in any other programming environment.

The space efficiency of dynamic coupling is good since it avoids any intermediate tree construction. We stress again that, if we restrict the ACs in a sequence to comply with the original descriptional composition algorithm (every syntactic attribute occurrence is used exactly once, and semantic attributes do not influence tree construction), pushing a tuple through the filter pipe to get a virtual production for the last evaluator is a constant-space process. There is strictly no need to store additional information at the tree nodes. This should be clear from Fig. 11.

Regarding time efficiency, dynamic coupling appears somewhat expensive, as pointed out at the end of section 3.2. Only practical experience with a real implementation—which we don't have yet—will show how big a problem this is. However we think that even a naive implementation would have a honorable speed, and a careful implementation with caching and elimination of copy rules (see below) would be quite usable. In addition, as explained above, it is possible to trade flexibility for speed through inlining.

The second advantage of the notion of coupling evaluator is certainly that there is no class constraint on the coupling evaluators. The class of the resulting (complete) evaluator is the class of the parameterizable evaluator of the last AC of a given sequence. Thus, to construct an evaluator based on visit sequences, for a given AC sequence, only the last AC needs to have such an evaluator. In fact, the resolution of class constraints in descriptional composition was the original motivation of this work. This brings an interesting result for attribute grammar evaluators: we can now construct an evaluator based on visit sequences for a non-SNC attribute grammar, provided that it is correctly modularized, i.e., decomposed in a sequence of AC where the last one is SNC [7]. Note that the static coupling technique solves the class closure problem but not that of separate compilation.

From a theoretical point of view, it is very difficult to formally compare the naive, sequential composition with our new technique, beside the fact that intermediate trees are not constructed. The only good (but partly false) view which we could give is a comparison with factoring in mathematics: with coupling evaluators we factor out moves into one tree rather than walking on several trees.

In this paper we presented a first approach to the construction of coupling evaluators, in which the main goal was not efficiency, but rather usability and simplicity. To improve the dynamic coupling evaluator we can introduce various extensions to the basic construction.

The first one regards a specific treatment of copy rules. Indeed, in a sequence of coupling evaluators (filters), when an identity virtual production is encountered, then all the following virtual productions are also identities. So, it is easily possible to eliminate all the subsequent transformations, since they are identities. Furthermore, by using non-local virtual productions, i.e., productions which relate nodes which are not immediate neighbors to each other in the tree, it is possible to eliminate many attribute copies. You can find all the details of

these optimizations in [14]. With these optimizations most copy rules eliminated by our static optimizer are dynamically eliminated. However, moves in the tree still remain.

In this paper, in order to make the presentation easier, we had implicitly restricted ourselves to purely syntactic attribute couplings. However it is easy to accommodate semantic attributes, by decomposing each AC into an evaluator for the semantic part and a filter for the syntactic part. Then, you compute first the semantic attributes of the first AC, storing in the tree those which are needed in the sequel of the sequence, then you compute the semantic attributes of the second AC using the corresponding evaluator and the filter for the first AC, and so on. If the value of a semantic attribute can influence the syntactic part of its AC, this can be handled in the corresponding filter, provided the attribute is stored in the tree.

In this paper, we had also assumed that we had the same restrictions regarding the input ACs as for descriptional composition. These restrictions are essentially that every syntactic attribute occurrence must be used exactly once, to avoid the construction of a dag or a forest. With dynamic coupling evaluators we can eliminate the dag restriction. Indeed, with one additional argument it is possible to distinguish the various parents of a given node in a dag, and hence their attributes. This was not possible with descriptional composition, as it needs to know statically the number of attributes on every node. More details appear in [14].

5 Conclusion

In this paper we have presented a new, dynamic approach to the composition of attribute grammars. Like descriptional composition [5], it avoids the construction of intermediate trees. Like separable AGs [4], it allows the separate compilation of the AGs in the sequence. Hence, it should prove valuable for developing large applications with AGs, since it makes both modularity and reusability in AGs more practical.

Of course, this approach needs to be implemented and practical experience with it must be gained to decide whether it is indeed effective. This is part of our future work. The implementation will be carried out as part of the Fnc-2 system, which already provides the basic bricks such as evaluators able to walk upwards in the tree.

We will also work to make the basic technique more powerful and more efficient.

References

1. Boyland, J. and Graham, S. *Composing Tree Attributions.* In *21st POPL.* Jan. 1994, 375–388.
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. Duris, É. Transformation de grammaires attribuées pour la mise à jour destructive. Dépt. d'Informatique, Univ. d'Orléans, Rapport de DEA, Sept. 1994.
4. Farrow, R., Marlowe, T. J. and Yellin, D. M. *Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation.* In *19th POPL.* Jan. 1992, 223–234.
5. Ganzinger, H. and Giegerich, R. Attribute Coupled Grammars. *SGPLN 196* (June 1984), 157–170.
6. Ganzinger, H., Giegerich, R. and Vach, M. MARVIN: a Tool for Applicative and Modular Compiler Specifications. Fachbereich Informatik, Univ. Dortmund, Forschungsbericht 220, July 1986.
7. Giegerich, R. Composition and Evaluation of Attribute Coupled Grammars. *Acta Inform. 25* (1988), 355–423.
8. Jourdan, M., Le Bellec, C., Parigot, D. and Roussel, G. *Specification and Implementation of Grammar Couplings using Attribute Grammars.* In *Programming Languages Implementation and Logic Programming*, M. Bruynooghe and J. Penjam, Eds. Lect. Notes in Comp. Sci., vol. 714, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1993, 123–136.
9. Jourdan, M. and Parigot, D. *Application Development with the FNC-2 Attribute Grammar System.* In *Compiler Compilers '90*, D. Hammer, Ed. Lect. Notes in Comp. Sci., vol. 477, Springer-Verlag, New York–Heidelberg–Berlin, Oct. 1990, 11–25.
10. 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.* SGPLN, vol. 25, no. 6, July 1990, 209–222.
11. Kastens, U. *Implementation of Visit-Oriented Attribute Evaluators.* In *Attribute Grammars, Applications and Systems*, H. Alblas and B. Melichar, Eds. Lect. Notes in Comp. Sci., vol. 545, Springer-Verlag, New York–Heidelberg–Berlin, June 1991, 114–139.

12. Knuth, D. E. Semantics of Context-free Languages. *Math. Systems Theory 2*2 (June 1968), 127–145, Correction: *Math. Systems Theory 5*, 1, pp. 95-96 (Mar. 1971)..

13. Le Bellec, C. La généricité et les grammaires attribuées. Dépt. d'Informatique, Univ. d'Orléans, Thèse de doctorat, June 1993.

14. Roussel, G. Différentes transformations de grammaires attribuées. Dépt. d'Informatique, Univ. de Paris VI, Thèse de doctorat, Mar. 1994.

15. Roussel, G., Parigot, D. and Jourdan, M. *Coupling Evaluators for Attribute Coupled Grammars.* In *5th Int. Conf. on Compiler Construction (CC '94)*, P. A. Fritzson, Ed. Lect. Notes in Comp. Sci., vol. 786, Springer-Verlag, New York–Heidelberg–Berlin, Apr. 1994, 52–67.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399