World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:4, No:1, 2010

Dynamic Attribute Dependencies in Relational Attribute Grammars

K. Barbar*, M. Dehayni†, A. Awada‡, and M. Smaili§

***Abstract*—** Considering the theory of attribute grammars, we use logical formulas instead of traditional functional semantic rules. Following the decoration of a derivation tree, a suitable algorithm should maintain the consistency of the formulas together with the evaluation of the attributes. This may be a Prolog-like resolution, but this paper examines a somewhat different strategy, based on production specialization, local consistency and propagation: given a derivation tree, it is interactively decorated, i.e. incrementally checked and evaluated. The non-directed dependencies are dynamically directed during attribute evaluation.

***Keywords*—**Input/Output attribute grammars, local consistency, logical programming, propagation, relational attribute grammars.

Open Science Index, Computer and Information Engineering Vol:4, No:1, 2010 publications.waset.org/1245.pdf

I. Introduction

ATTRIBUTE grammars (abbreviated AGs) were first introduced by Knuth to describe syntactic-based translations [1]. This approach is a purely declarative programming paradigm syntax directed. They have been widely studied and used, especially as a compilation technique in the field of programming languages [2], or as formal specifications for more general tree transductions and semantics [3]–[6].

An AG is a context-free grammar (CFG) which non-terminal symbols are decorated with inherited and synthesized attributes, and productions are enriched with semantic rules defining assignments for the attributes. The goal is to give some “meaning” to the terms obtained from the grammar. The semantic rules show dependencies between attributes, revealing the order to compute their values. In short, some attributes should be computed before other ones because the formers are parameters of the latters.

When using AG, one common task is to avoid circular dependencies for any derivation tree of the grammar. Knuth presented an exponential-space algorithm for the circularity problem [1]. The intrinsically exponential complexity of this problem was first proved by Jazayeri, Ogden, and Rounds [7], who reduced the acceptance problem of writing pushdown acceptors to the circularity problem. Jazayeri [8] (and the correction by Dill [9]) tried to provide a simpler construction of AGs by reducing the acceptance problem of space-bounded alternating Turing machines.

Another task is to determine efficient methods to compute all the attributes in a given derivation tree. In the classical way, the AG is statically analyzed to anticipate the whole dependencies of any derivation tree. The attributes in the semantic rules are defined as input (or output) only in order to find convenient properties of the dependencies [10]. Yet, these restrictions reduce the expressiveness of AGs.

Classical AGs lack of expressiveness has resulted in limited use outside the domain of static language processing. This leads to extend the classical formalism into the notion of Dynamic Attribute Grammars (DAG) [11] to enhance the expressiveness and to allow describing computations on structures that are not just trees. This results in a language that is comparable in power to most functional languages, with a distinctive declarative character. Kikuchi and Katayama define the semantics of general AGs by using semantic functions whose inputs are structures derived from the underlying grammar and whose outputs are attributed structures [12]. Then they provide classifications of general AGs based on the abstract properties of semantic functions. In [13], Neven introduces extension of AGs that work over extended CFG, allowing arbitrary regular expressions on the right-hand side of productions. Viewed as a query language, extended AGs are particularly relevant as they can take into account the inherent order of the children of a node in a structured document.

In this paper we will rather emphasize the fact that using non-directed semantic rules (i.e. relations or constraints) greatly enhances the declarative power of AGs, because it is no more presumed which attribute should be evaluated first. At “execution time” dependencies are dynamically built when some attribute is declared as input attribute.

In section3, we recall basic formal definitions of concepts like CFG and productions. In Section4, we adapt the definition of relational AGs [14] to fit our needs. Section5 reveals input/output productions as our main tool to describe static and dynamic evaluation. Such productions are connected together in Section6 to build partially evaluated derivation trees. They are compared together in Section7 to show possible transformations from one into another. Then, in Section8 we incrementally evaluate the attributes of a derivation tree using step by step transformations of the production occurrences.

* K. Barbar is with the Faculty of Sciences (section 2), Lebanese University, Fanar, Lebanon (e-mail: kbarbar@ul.edu.lb).
† M. Dehayni is with the Faculty of Sciences (section 1), Lebanese University, Hadath, Lebanon (e-mail: maydehayni@ul.edu.lb).
‡ A. Awada is with the Faculty of Sciences (section 1), Lebanese University, Hadath, Lebanon (phone: 961-3-660924; fax: 961-5-465562; e-mail: al_awada@ul.edu.lb).
§ M. Smaili is with the Faculty of Sciences (section 1), Lebanese University, Hadath, Lebanon (e-mail: mosmaili@ul.edu.lb).

Open Science Index, Computer and Information Engineering Vol:4, No:1, 2010 publications.waset.org/1245.pdf

II. RELATED WORKS

A Relational AG is a formal tool to define such non-directed relational semantic rules separately from an abstract CFG [14]. It does not give any operational clue to satisfy the relations in a decorated tree. Yet, if an attribute in a production occurrence is declared being an input attribute, some other attributes that depend on may be evaluated. As any attribute can become an input attribute in a production occurrence, the search for dependencies is parameterized with the set of current input attributes.

This study is related to previous works that brings together AGs and logic [15], [16] or constraint satisfaction [17]–[19]. In [20], a unified view of AGs and logic programs is presented. The author compares both formalisms and shows that AGs have some features that are not present in logic programs. He proposes some extensions in the field of logic programming in order to enrich logic programs with extra features. Batory describes the interpretation of grammar representations in terms of propositional logic formulae [21]. Isakowitz introduces Abstract Attribute Grammars (AAG) in order to study the transformation from Logic Programs into AAG and vice versa [22]. He provides a construction that transforms any logic program into an equivalent AAG. The motivation for much of this work comes from the need for verifying the correctness of feature model selections that represent individual products. Ruffolo and Manna define semantic models in order to exploit domain knowledge for managing both structured and unstructured information [23]. These semantic models are executable, flexible and agile representation of domain knowledge. They are expressed by means of the Codex Language obtained combining Disjunctive Logic Programming and AGs.

Dynamic and incremental use of AGs is also concerned [11], [24], [25]. Reps, Teitelbaum, and Demers develop the Cornell Synthesizer Generator that is an incremental evaluator generation tool. It offers the possibility of replacing a sub-tree of a syntactic tree by another sub-tree: the propagation of new attribute values on the whole tree is then automatically processed [26]. By combining dynamic and incremental aspects, attribute dependencies can be dynamically directed, and that waiting for the values of all the input attributes is not necessary in order to evaluate some parts of the decorated tree. Unlike Prolog, enumeration of all the solutions is not considered here because there is usually infinitely many, unless some extra information. On the contrary, partial solutions are managed. Thus, each time an attribute gets a value in a production occurrence, the current set of input attributes is enriched, potentially leading to new dependencies and incremental propagations in the derivation tree.

III. ABSTRACT CONTEXT-FREE GRAMMARS

In this section, we recall basic formal definitions of concepts that will be used further on, like CFG, productions and derivation trees.

Definition1 (CFG) *A context-free grammar is a tuple (N, T, Z, P) where:*

- *N is the alphabet of non-terminal symbols;*
- *T is the alphabet of terminal symbols;* $N \cap T = \varnothing$
- *Z is the axiom of the grammar,* $Z \in N$*; Z must be the root of any derivation tree of the grammar;*
- *P is a set of context-free productions (see Definition2).*

Definition2 (CFP) *A context-free production p in a CFG (N, T, Z, P) is a tuple* $X_0 \rightarrow X_1 \ldots X_n$ *where:*

- X_0 *is an occurrence of an element of N;*
- $X_1 \ldots X_n$ *are occurrences of elements of* $N \cup T$.

In the production *p*: X_0 is the left-hand side of p, while $X_1 \ldots X_n$ is the right-hand side. The elements in a production p, even if they are occurrences of identical terminal or non-terminal symbols, are characterized by their positions in p, in Dewey notation (the symbol 0 is the empty word ε). Thus, there is a straightforward tree representation of a production, where the root is X_0 and leaves are $X_1 \ldots X_n$.

Definition3 (ACFG) *An abstract CFG is a tuple (N, P) where N and P appear in a CFG tuple (N, T, Z, P). If a production p in P contains some occurrences of terminals in T, they are ignored. Consequently a production p is essentially considered as a tuple of one or more non-terminals.*

ACFG are the essence of context free grammars. We can also note that the axiom Z disappears: any non-terminal in the left-hand side of a production can be the root of a derivation tree of the grammar.

Example1 shows an ACFG that will be used with attributes in section3 to illustrate the factorial function. Example2 shows an ACFG for strictly binary trees.

Example1 (FacACFG) An abstract CFG representing a free monoid:

N = {Fac}

P = {p_1: Fac → Fac , p_2 : Fac → ε}

Example2 (BinACFG) An abstract CFG for strictly binary trees:

N = {Bin}

P = (p_1: Bin → Bin Bin, p_2: Bin → ε}

Let G be an ACFG. A production $X_0 \rightarrow X_1 \ldots X_n$ in G gives rise to a set of trees which roots represent occurrences of X_0, and direct sub trees derived from productions with $X_1 \ldots X_n$ as left-hand sides. In a derivation tree, the occurrences of (the roots of) the productions and the occurrences of the non-terminals are characterized by their positions in Dewey notation, obtained by concatenating their position in the production to the global position of their parent in the tree.

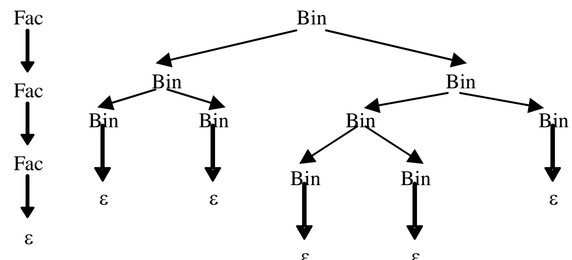


Fig. 1 Abstract Derivation Trees

Fig.1 shows two derivation trees for Fac_{ACFG} and Bin_{ACFG}. The Fac occurrences of the production p_1 appear at positions 0, 1 and an occurrence of p_2 is at position 1.1; while the Bin occurrences of the production p_1 appear at positions 0, 1, 2 and 2.1.

The way a derivation tree is constructed is an important point for us, as it partly determines why and how the attributes are computed. Look again at the trees in Fig.1, and imagine that they are interactively built. Each time, we have to choose which non-terminal to develop, using the productions of the grammar as construction rules. Clearly, different strategies exist, whether bottom-up, top-down, or in either direction, to finally obtain a derivation tree. In the next sections, we show how these strategies determine orientations between attributes. But in most cases, we will consider a tree that already exists before any orientations or evaluations.

Open Science Index, Computer and Information Engineering Vol:4, No:1, 2010 publications.waset.org/1245.pdf

IV. Relational Attribute Grammars

In this section, we recall the definition of relational attribute grammars (RAG). Definitions of "classical" AGs can be found in [4], [10], but we do not use them here. It is probably possible to transform AGs into RAGs, e.g. by considering directed semantic rules as non-directed, thus breaking the distinction between inherited (evaluated during a top-down tree traversal) and synthesized attributes (evaluated during a bottom-up traversal) in the specification of the grammar.

RAGs were introduced to prove the validity of a "classical" AG with respect to a specification [14], [27]. A specification consists in a collection of logical formulas, each formula being associated to a production and establishing the relationships between the attributes of this production.

From a different point of view, logical formulas can replace the semantic rules to directly specify which relations the attributes of a production should respect. In the sequel, we adopt this point of view. Such formulas enhance the declarative power of classical AGs, while it becomes more difficult to prove their correctness and to compute attributes, because of the use of a possibly too general logical language.

We use slightly different notations than [14], [27]. We don't define the sorts of the attributes, nor give precise definition for the logical parts of the grammars. These restrictions are not significant for the purpose of this paper because we just want to outline a specification level independent from the underlying logical language.

Definition4 (RAG) *A Relational Attribute Grammar is a tuple (N, P, A, φ, I) where:*

- *(N,P) is an ACFG;*
- *A is the alphabet of attribute names;*
- *φ is a finite set of formulas from a logical language L;*
- *I is the interpretation of L.*

Each element X in N is decorated with a subset of A, denoted A_x. An attribute symbol a appearing in A_x is called the occurrence of the attribute a in X, or the attribute a of X, and is denoted a(X) or X.a. For a production p in P, the attributes a, b... of a non-terminal X appearing at position i are denoted a_i, b_i, ... so they form a set denoted A_p, X_i. The set A_p of the attributes appearing in p is equal to $\bigcup_{X_i \in p} A_p, X_i$. *Each production in P is associated to a unique formula in φ as explained in Definition5. N and P form the grammatical part (syntax) of the relational attribute grammar, while A, φ, and I form the labeling formalism part (semantics).*

Definition5 (RAP) *A relational Attribute Production in a relational attribute grammar (N, P, A, φ, I) is a tuple (p, φ_p) where:*

- *p is a context-free production from P;*
- *φ_p is the formula of φ associated to p.*

The set of variables in φ_p should be a subset of A_p. If not specified, these variables are existentially quantified. For each context-free production p in P, there is a unique relational attribute production (p, φ_p) (thus, we may ambiguously use p to denote the context-free production p or the relational attribute production (p, φ_p)).

Example3 shows a RAG for the factorial function. The attributes n and r of the non-terminal Fac respectively denote the argument of the recursive call and the corresponding result n!.

Example3 (Fac_{RAG}) A RAG for the factorial function:

$N = \{Fac\}$
$P = \{p_1: Fac \rightarrow Fac,\ p_2: Fac \rightarrow \varepsilon\}$
$A_{Fac} = \{n,r\}$
$A_{p1} = \{ n_0, r_0, n_1, r_1\}$
$A_{p2} = \{ n_0, r_0\}$
$\varphi_{p1} = (n_0 = n_1 + 1 \wedge r_0 = r_1 \times n_0)$
$\varphi_{p2} = (n_0 = 0 \wedge r_0 = 1)$
I : $L \rightarrow$ Bool $\cup$ Nat; Nat denotes natural numbers with the usual operations.

A more general function is shown in Example4. The formulas associated to the productions specify the relations between the occurrences of the attributes. In these formulas, the symbol '=' denotes an equivalence predicate, not an assignment function, so it is not explicitly specified how the attributes should be computed. It strongly depends on the way a derivation tree is constructed, and on extra information like the values of some unknown attributes.

Example4 ($ParamFac_{RAG}$) A RAG for a kind of factorial function with unspecified start conditions:

$N = \{ParamFac\}$
$P = \{p_1: ParamFac \rightarrow ParamFac,\ p_2: ParamFac \rightarrow \varepsilon\}$
$A_{ParamFac} = \{n, r\}$
$A_{p1} = \{ n_0, r_0, n_1, r_1\}$
$A_{p2} = \{ n_0, r_0\}$
$\varphi_{p1} = (n_0 = n_1 + 1 \wedge r_0 = r_1 \times n_0)$
φ_{p2} = true, the formula is always true
I = L $\rightarrow$ Bool $\cup$ Nat≥0; Nat≥0 is the domain of positive natural numbers with the usual operations.

Let G be a RAG. The set of relational derivation trees determined by G is the same as the set of derivation trees determined by the ACFG in G, the occurrences of the non-

terminals being decorated with additional vertices representing their attributes. An attribute in the tree has the same position as its related non-terminal occurrence. The relations between the attributes are represented by edges or hyper edges. Consequently, for a derivation tree t, the attributes (vertices) together with the relations (edges) form a graph called the attribute graph of t and denoted *Graph(t)*.

In Section2, some strategies to generate derivation trees were outlined. The attribute graph of a derivation tree t is generated while t is, using the same strategy.

Fig.2 shows two derivation trees, one from Fac_{RAG}, and one from $ParamFac_{RAG}$. In the tree from Fac_{RAG}, the edges representing relations could be directed, as it is possible to evaluate all the attributes starting at the values 0 and 1 in the occurrence of production p_2. This is not the case in the tree from $ParamFac_{RAG}$. These points are investigated in the next section.

V. Input/output Productions

The logical formulas associated with relational attribute productions specify the relationships between the attributes, but they are not supposed to be straight operational. The attributes in a production are not characterized as input or output. This constitutes the (declarative) strength and the (operational) weakness of RAG.

The way the attributes of a derivation tree should be computed depends strongly on the strategy to generate the tree together with its attribute graph and on the values of some input attributes from which to start. In the left tree Fig.2, a top-down strategy would wait an occurrence of p_2 before evaluating, while a bottom-up strategy, starting at the occurrence of p_2, could evaluate the attributes during the construction of the tree. Each strategy would consider the values 0 and 1 in φ_{p2} as input. In the tree at the right on Fig.2, the situation is clearly different: some extra information is needed to compute even a part of the attributes; else there are infinitely many solutions.

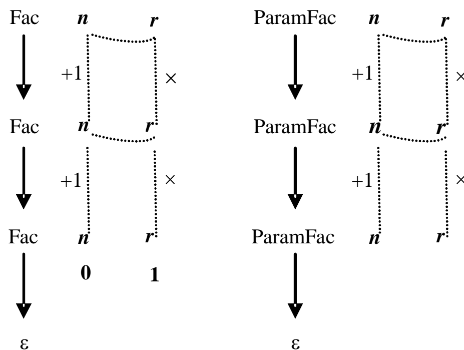


Fig. 2 Abstract Derivation Trees

In this section, input and output attributes are distinguished in a production, but some attributes may be neither input nor output. Moreover, different values for each input attribute are considered. A relational attribute production p gives rise to several productions of a new type, each production syntactically identical to p but with different input attributes and different values for these attributes. For each of these productions, (the values of) output attributes result in functional applications of (the values of) input attributes. We call such productions *input/output relational attribute productions,* or *I/O productions* in short.

Definition6 (I/O RAP) An Input/Output Relational Attribute Production is a tuple (p, In, v, Out, v'):

- *p is a relational production from a relational grammar (N, P, A, φ, I);*
- *In is a subset of A_p, called the input set, which elements are called the input assignment attributes;*
- *v is a mapping from In to the domain of interpretation, called the input assignment;*
- *Out is a subset of A_p - In, called the output set, which elements are called the output attributes;*
- *v' is a mapping from Out to a domain of functional terms which variables are the attributes of In. These terms have to be interpreted with I and v. v′ is called the output assignment.*

Let a be an output attribute. We have $v'(a)= f(a_1,...,a_k)$, with f a functional term and $a_1,...,a_k$ some input attributes (the indexes do not correspond to positions in p). The output assignment v' should be a "logical consequence" of φ_p with the assignment v in the interpretation I, so we write: $(I,v) \models [\varphi_p \wedge a=f(a_1,...,a_k)]$.

The most important problem, not addressed here, is to automatically determine proper functional terms f in v', just by considering φ_p and v, i.e. information relative to the production, and not from a global knowledge about the grammar. These terms should precisely reflect functional properties of φ_p.

The set A_p – *In* - *Out* is the set of attributes which are neither input nor output attributes. They are called the unknown attributes. It is possible to tell properties of these attributes just by looking at φ_p and v. One of these properties is that the set of their possible values is not reduced to a singleton, so they don't appear in *Out*.

Because we give no restrictions on the formulas in φ, we have to distinguish the values of the input attributes: in general, different input values for the same attribute may induce different output sets, e.g. the formula $((a=0 \wedge b=1) \vee (a\geq 1 \wedge c=a-1))$ gives different output sets for different values of a.

Eventually, φ_p can be invalidated by v. This situation is called a *conflict*. Depending on the logical language and its interpretation, verifying the validity of a formula under a particular assignment may be "hard" or, worse, undecidable, but it is not the subject of this paper. An assignment v is said *valid* if it does not invalidate φ_p. In the sequel, we will only consider productions for which there is no conflict, *i.e. we consider only valid assignments*. This restriction will be relaxed in future papers.

Open Science Index, Computer and Information Engineering Vol:4, No:1, 2010 publications.waset.org/1245.pdf

Example5 Two I/0 productions obtained from a production p_1:

p_1 = <ParamFac → ParamFac, $n_0 = n_1 + 1 \wedge r_0 = r_1 \times n_0$>

$p'_1 = <p_1, In_1, v_1, Out_1, v'_1>$	$p'_2 = <p_1, In_2, v_2, Out_2, v'_2>$
$In_1 = \{n_0, r_1\}$	$In_2 = \{n_1\}$
$v_1(n_0) = 3$	$v_2(n_1) = 3$
$v_1(r_1) = 8$	$Out_2 = \{n_0\}$
$Out_1 = \{n_1, r_0\}$	$v'_2(n_0) = n_1 + 1$
$v'_1(n_1) = n_0 - 1$	
$v'_1(r_0) = r_1 \times n_0$	

A relational attribute production *p* gives rise to a set of I/O productions, denoted IO_p. Given a valid assignment *v* for a subset of A_p, there exists at least one I/O production $<p,In,v,Out,v'>$ in IO_p. Hence, IO_p is infinite if the set of valid assignments is infinite, limiting the practical use of I/O productions. It would be preferable to partition IO_p into finitely many classes of I/O productions, as it will be shown in Section6.

Now, we have the basic elements to specify static and dynamic properties of attribute dependencies in derivation trees. In Section5, occurrences of I/O productions are connected together to give a derivation tree *t* and to produce directed dependencies between the attributes in the attribute graph of *t*. In Section6, we modify input sets to show dynamic transformations of I/O productions. In Section7, these transformations are used in derivation trees to locally propagate the evaluation of the attributes.

VI. Input/output Attribute Grammars

As explained in section2, given two productions p_1 and p_2 of an ACFG, an occurrence of p_1 can be connected to an occurrence of p_2 if the left-hand side of p_1 appears in the right-hand side of p_2, or inversely, if the left-hand side of p_2 appears in the right-hand side of p_1. The same principle applies for RAPs and their attribute graphs.

Yet, considering I/O productions, we need to define more precisely how they can be connected together in terms of assignments of attributes. Doing this we straight define the derivation trees of a grammar containing I/O productions.

Definition7 (IOAG) *An Input/Output Attribute Grammar is a tuple (G, P′) where:*

- *G is a RAG (N, P, A, φ, I)*
- *P′ is the set of I/O productions obtained from p and φ;*

$$P' = \bigcup_{p \in P} IO_p$$

Each production in a RAG implies several I/O productions. If the occurrences of two RAPs p_1, p_2 can be connected together, so can the occurrences of two I/O productions derived from p_1 and p_2, with additional constraints on the input and output sets and on the assignments.

Let p_1, p_2 be two RAPs, and p'_1, p'_2 two I/O productions respectively derived from p_1 and p_2. Suppose that the left-hand side of p_2 appears in the right-hand side of p_1 at position *k*.

$$\left.\begin{array}{l} p_1 = < X_0 \rightarrow X_1 \ldots X_k \ldots X_{n1}, \varphi_{p1}> \\ p_2 = < Y_0 \rightarrow Y_1 \ldots Y_{n2}, \varphi_{p2}> \end{array}\right\} \quad Y_0 = X_k$$

$$p'_1 = <p_1,In_1,v_1,Out_1,v'_1>$$
$$p'_2 = <p_1,In_2,v_2,Out_2,v'_2>$$

Y_0 and X_k are the same non-terminal symbol. When an occurrence of p_2 is connected at position *k* to an occurrence of p_1, the attributes of Y_0 and X_k become the same vertices in the attribute graph of the resulting tree. In notations, it is important that the indexes of the attributes in both occurrences correspond to their positions in the tree, such that further on, there is no ambiguity in the input and output sets and in the assignments.

When we connect occurrences of p'_1 and p'_2 the same way as occurrences of p_1 and p_2, the following constraints apply on input and output sets:

$$In_1 \cap A_{p_1,X_k} \subset (In_2 \cup Out_2) \cap A_{p_2,Y_0} \quad (1)$$

$$In_2 \cap A_{p_2,Y_0} \subset (In_1 \cup Out_1) \cap A_{p_1,X_k} \quad (2)$$

$$Out_1 \cap A_{p_1,X_k} \subset In_2 \cap A_{p_2,Y_0} \quad (3)$$

$$Out_2 \cap A_{p_2,Y_0} \subset In_1 \cap A_{p_1,X_k} \quad (4)$$

Formula (1) (resp. (2)) indicates that an input attribute of X_k (resp. Y_0) should be an input or output attribute of Y_0 (resp. X_k). Formula (3) (resp. (4)) indicates that an output attribute of X_k (resp. Y_o) should be an input attribute of Y_0 (resp X_k). Moreover, still considering the same connection between occurrences of p'_1 and p'_2, the following constraints apply on the assignments:

$$\forall a \in In_1 \cap A_{p_1,X_k}, [a \in Out_2 \wedge v_1(a) = I(v'_2(a))] \vee [a \in In_2 \wedge v_1(a) = v_2(a)] \quad (5)$$

$$\forall a \in In_2 \cap A_{p_2,Y_0}, [a \in Out_1 \wedge v_2(a) = I(v'_1(a))] \vee [a \in In_1 \wedge v_2(a) = v_1(a)] \quad (6)$$

$$\forall a \in Out_1 \cap A_{p_1,X_k}, [a \in In_2 \wedge I(v'_1(a)) = v_2(a)] \quad (7)$$

$$\forall a \in Out_2 \cap A_{p_2,Y_0}, [a \in In_1 \wedge I(v'_2(a)) = v_1(a)] \quad (8)$$

Formula (5) (resp. (6)) indicates that the value of an input attribute of X_k (resp. Y_0) should be equal to the interpretation of the value of the same attribute considered as an output attribute of Y_0 (resp. X_k), or to the value of the same attribute considered as an input attribute of Y_0 (resp. X_k). Formula (7) (resp. (8)) indicates that the interpretation of the value of an output attribute of X_k (resp. Y_0) should be equal to the value of the same attribute considered as an input attribute of Y_0 (resp. X_k). In short, a common attribute of both occurrences is either an input attribute for each, or an output attribute for one occurrence and an input attribute for the other, or finally an unknown attribute for each.

Example 6 Connecting occurrences of I/O productions: p_1 is a RAP; p'_1, p'_2, and p'_3 are I/O productions obtained from p_1. An occurrence of p'_2 can be connected at the position 1 in an occurrence of p'_1, because $v_1(n_1) = v_2(n_0)$. This is not the case for any occurrence of p'_3, because $v_1(n_1) \neq v_3(n_o)$. However, an occurrence of p'_3 can be connected at position 1 in an occurrence of p'_2, because $I(v'_2(n_1)) = v_3(n_o)$.

p_1 = <ParamFac → ParamFac, $n_0 = n_1 + 1 \wedge r_0 = r_1 \times n_0$>

p'_1 =	p'_2 =	p'_3=
$<p_1,\{n_1\},v_1,\{n_0\},v'_1>$	$<p_1,\{n_0\},v_2,\{n_1\},v'_2>$	$<p_1,\{n_0\},v_3,\{n_1\},v'_3>$
$v_1(n_1) = 3$	$v_2(n_0) = 3$	$v_3(n_0) = 2$
$v'_1(n_0) = n_1 + 1$	$v'_2(n_1) = n_0 - 1$	$v'_3(n_1) = n_0 - 1$

Open Science Index, Computer and Information Engineering Vol:4, No:1, 2010 publications.waset.org/1245.pdf

Open Science Index, Computer and Information Engineering Vol:4, No:1, 2010 publications.waset.org/1245.pdf

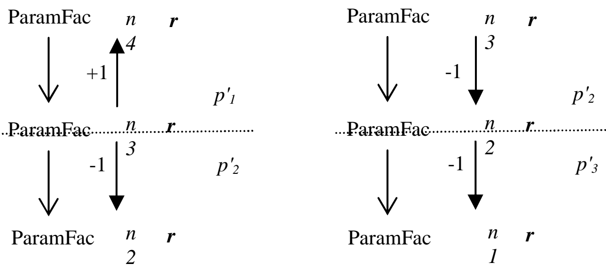


Fig. 3 Input/Output Derivation Trees for Example6

Let G be an IOAG. The set of *input/output derivation trees* determined by G is the set of trees in which the occurrences of the I/O productions are correctly connected together. The attribute graph associated with a derivation tree contains directed edges between input and output attributes. An edge links an input attribute a to an output attribute b if a is a parameter of b in the output assignment.

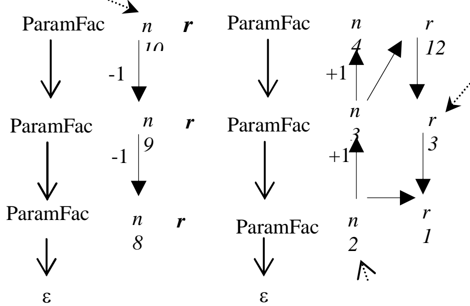


Fig. 4 Input/Output Derivation Trees

Fig.4 shows two such derivation trees and trees and their attribute graphs. Some attributes are spotted out (dashed lines with arrows) because they are global (or external) input attributes, *i.e.* attributes that are input in every production occurrence they appear in. The other attributes are input in one production occurrence and output in another, if not just output. The directed edges give the direction of the evaluation, i.e. the interpretation of the functional expressions from input to output attributes.

VII. Comparing Input/Output Productions

In this section, we study the growth of the input set of a single production and the induced consequences on its output. In an I/O production, turning some unknown attributes into input attributes accrues to change this production into another, potentially with new output. It gives a means to compare I/O productions in terms of input attributes and assignments.

Let p be a RAP. An I/O production p'_1 derived from p can be transformed into another I/O production p'_2 also derived from p, by extending the input set of p'_1. The input set of p'_2 should be the union of the input set of p'_1 together with this extension. Informally, as the input set grows from p'_1 to p'_2, the output set grows and the set of unknown attributes diminishes. When the set of unknown attributes is empty, the I/O production is said to be *saturated*, and cannot be more transformed.

Definition8 (Specialization) *Let p be a RAP, and $p'_1 = <p_1,In_1,v_1,Out_1,v'_1>$, $p'_2 = <p_1,In_2,v_2,Out_2,v'_2>$ two I/O productions derived from p. p'_2 is said to be a specialization of p'_1, which is denoted $p'_2 \prec p'_1$, if:*

$$In_1 \subseteq In_2 \wedge \forall a \in In_1,\ v_1(a) = v_2(a)$$

We also say that p'_2 is more saturated than p'_1. The consequences of these constraints are the followings: $Out_1 \subseteq Out_2$ and $\forall a \in Out_1$, $v'_1(a) = v'_2(a)$

The relation $\prec$ is a partial and well-founded order for IO_p. A unique upper bound, called the empty I/O production, exists; it corresponds to p with an empty input set. Every I/O production in IO_p is a specialization of this one. On the contrary, and in non trivial cases, there are infinitely many incomparable lower bounds which are the *saturated* I/O productions of IO_p (productions without unknown attributes).

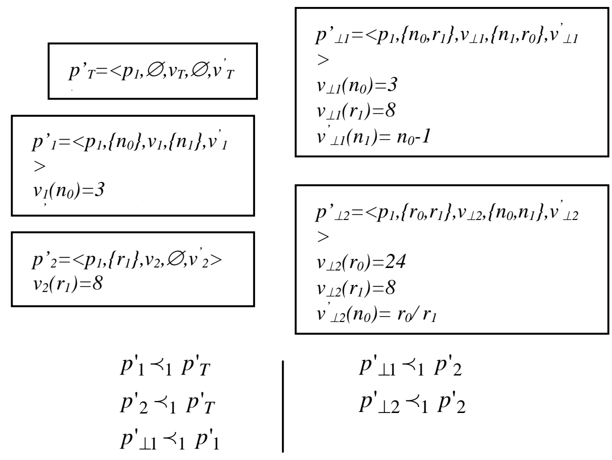
The set In_2 - In_1 is the set of input attributes that appear in p'_2 but not in p'_1. Let n be the cardinal of In_2 - In_1. If n = 0, then $p'_2 = p'_1$. If n = 1, p'_2 is said to be a one-attribute specialization of p'_1, which is denoted $p'_2 \prec_1 p'_1$. If $n \geq 1$, there exists at least one chain $p'_2 \prec_1 p'_{1,n-1} \prec_1 \dots\dots\dots \prec_1 p'_{1,1} \prec_1 p'_1$ of n one-attribute specializations from p'_1 to p'_2 (each specialization $\prec_1$ adds one new input attribute from In_2 - In_1).

A specialization $p'_2 \prec p'_1$ reveals several parameters to describe p'_2 using p'_1. As every I/O production is the result of at least one chain of one-attribute specializations starting from the empty I/O production, it is straightforward to use a chain to define an I/O production. This would be a word in a language SL_p of specializations. Instead of fully specifying every I/O production derived from p, SL_p would describe how they share specification parts together.

The language SL_p can be represented with an automaton A_p which states represent I/O productions, and transitions represent extensions for input and output sets and assignments. The initial state is the empty production, and the final states are the saturated productions. Yet, A_p is infinite since IO_p is.

Example7 Specializing I/O productions: p'_T, p'_1, p'_2, $p'_{\perp 1}$, $p'_{\perp 2}$, are I/O productions derived from a RAP p_1. p'_T is the empty I/O production. $p'_{\perp 1}$ and $p'_{\perp 2}$ are incomparable saturated I/O productions.

p_1 = <ParamFac → ParamFac, $n_0 = n_1 + 1 \wedge r_0 = r_1 \times n_0$>



Open Science Index, Computer and Information Engineering Vol:4, No:1, 2010 publications.waset.org/1245.pdf

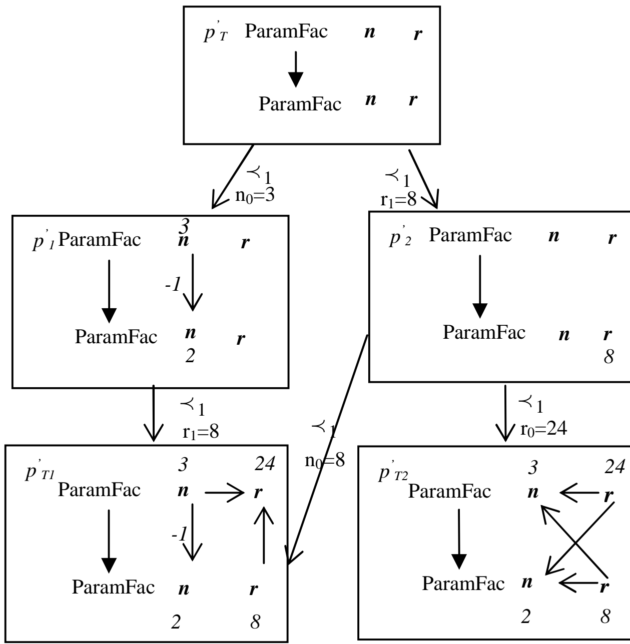


Fig. 5 Part of the automaton A_{p1}

The fact that A_p is infinite is not a good point in practice, so we propose now having a finite automaton. The objective is to partition IO_p into finitely many classes. Each class should verify the equivalence of the input and output sets of the I/O productions, and the equivalence of the output assignments in terms of functional expressions. The abstraction relies in grouping together I/O productions with the same "behavior" but different input assignments.

There exist algorithms that compute an abstract interpretation of Horn clauses by using an abstract domain *{ground, nonground}* for the values of the variables [28]–[30]. In the case of I/O productions, we will respectively use *known* instead of *ground* and *unknown* instead of *nonground*, and produce finite automata with abstract interpretations of the logical formulas in the productions.

Considering a relational production p, we present an algorithm that starts at the (singleton) class containing the empty I/O production derived from p. Finding a new class is considering a one-attribute specialization concerning a previously unknown attribute a. The abstract interpretation of ($\varphi_p \wedge a = known$) gives the set of (output) attributes which values are *known*, indicating the set of I/O productions contained in the new class. The algorithm terminates when no new class can be produced from the classes already found. A class is represented by a pair (*In, Out*) where *In* is the set of input attributes which values are *known* and *out* is the set of output attributes which values are *known*. The procedure *abstract-interpretation* is like the one developed in [28] for the computation of abstract interpretation of Prolog programs.

```
S= {(∅,∅)}
S'=∅
While S'≠S
  For each (In,Out) ∈ S - S'
  {
    S'=S' ∪ {(In, Out)}
    For each a ∉ In ∪ Out
    {
      Out'=abstract-interpretation(φp,In ∪ {a})
      S=S ∪ {(In ∪ {a}, Out'))}
    }
  }
```

This algorithm gives an idea of the construction of the (abstract) automaton: the initial state represents the class (∅,∅), and each time a pair [(In, Out), (In ∪ {a}, Out')] is found, a transition labeled a links the state representing the class (In, Out) to the state representing the class (In ∪ {a}, Out').

Example8 A finite automaton corresponding to the classification of I/O productions derived from a production p_1.

$p_1 = \langle ParamFac \rightarrow ParamFac,\ n_0 = n_1 + 1 \wedge r_0 = r_1 \times n_0 \rangle$

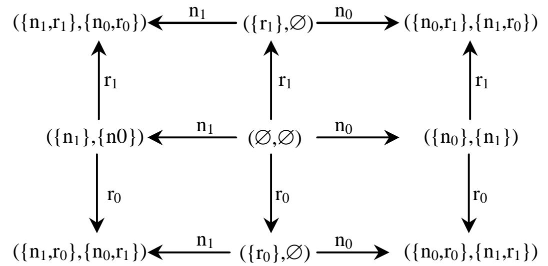


Fig. 6 Finite automaton of I/O productions from p1

VIII. Incremental Propagation in Derivation Trees

In this section, we show how to obtain a specialization series of partially evaluated derivation trees. The goal is to define transformations of the attribute graph of a tree, by means of local specializations of the production occurrences. Thus, attributes in the tree become progressively evaluated, until all attributes are.

The relation $\prec$ extends naturally from productions to trees. Consider two I/O derivation trees t_1 and t_2 determined by an IOAG and representing the same abstract tree but with different attribute graphs. The tree t_2 is a *specialization* of t_1, which is (ambiguously) denoted $t_2 \prec t_1$, if $p_2 \prec p_1$ for all pairs p_1, p_2 of production occurrences at the same position respectively in t_1 and t_2.

This general definition does not show to dynamically obtain t_2 from t_1, so we may introduce an incremental strategy for tree specializations.

Consider an I/O derivation tree t_T which is only built with occurrences of empty I/O productions. First, we choose an unevaluated attribute in T_t. Generally, any attribute belongs to two production occurrences; otherwise it is an attribute of a leaf or of the root of t_T. Giving a value for this attribute is specializing the two production occurrences.

Consequently, new output values are locally produced in each occurrence (see Section6). At this point, the tree is no

Open Science Index, Computer and Information Engineering Vol:4, No:1, 2010 publications.waset.org/1245.pdf

more a valid I/O derivation tree, but now the connected production occurrences receive the output values as input, so they are specialized in their turn.

Such local specializations propagate in the tree from occurrences to occurrences, and stop where it gives no output. The tree is globally specialized through that propagation of local specializations, and the result is an I/O derivation tree t' verifying $t' \prec t_T$. In fact, we can write $t' \prec_1 t_T$ because we did choose only one new input attribute in the whole tree.

We may repeat the preceding steps, until all the production occurrences in the tree are saturated, resulting in a saturated tree.

Example 9 Tree Specialization: starting from an empty I/O tree, the value of an attribute is given, leading to the propagation (dotted arrows) of specializations.

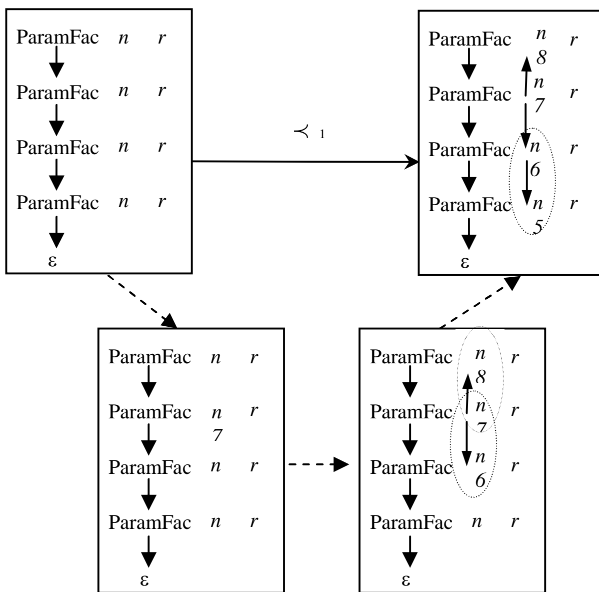


Fig. 7 Tree specialization

We present an algorithm for the incremental propagation in a derivation tree. This algorithm uses abstract values for the attributes, as shown in Section6. It starts with a new input attribute *input-a* for the tree. A pair (*p, a*) denotes a production occurrence and a new input attribute for this production occurrence. The algorithm uses such pairs to specify what attributes remain to be propagated in some productions.

The procedure *contains* takes an attribute in the tree and returns the set of production occurrences to which it belongs. The procedure *state* takes a production occurrence in the tree and returns the pair (*In, Out*) of current input and output attributes of this occurrence. The procedure *specialize* takes a production occurrence and a new input attribute and realizes the specializations of the production occurrence, as described in Section6. The symbol × denotes the Cartesian product of sets.

P= *contains*(input-a)
S=P × {input-a}

While S≠∅
{
S=S-{(p,a)}
(In, Out)= *state*(p)
specialize(p, In∪{a})
(In', Out')= state(p)
For each a' ∈ Out'- Out
{
P= *contains*(a')
S=S ∪(p × {a'})
}
}

IX. Conclusion

In this paper, we released the inherited and synthesized natures of attributes in AGs, by considering them more like variables in logical formulas, than variables in static functional expressions. Yet, we used functional expressions to reflect attribute evaluation, giving configurations in terms of input and output attributes. For each formulae, we studied the whole set of valid configurations, instead of considering just one configuration.

Using RAGs seems a trivial task: one gives an abstract context-free grammar, attributes for non-terminals and relations that bind these attributes together. Then, one supposes that the relations are satisfied all along the interactive edition of a derivation tree, including tree growing and attribute evaluation.

In fact, the operational point strongly depends on the logical part of the grammar. What if we cannot construct functional expressions from logical formulas? And even if these expressions are constructed for productions, we are faced with evaluation problems relating to non-local attribute dependencies in derivation trees. This cannot be done just by separately considering each production.

This problem is one of a wide class of evaluation problems that all depend on one point: circular attribute dependencies. In general, knowing to locally use some formulas is not sufficient for conjunctions of these formulas, because of strong connections between the variables. We hope to give some results concerning these problems in future studies.

However, we believe that local specializations and propagations can improve the expressiveness of semantic rules and the comprehensibility of the mechanism of attributes evaluation. In the case that one accepts the induced limitations, RAGs together with specialized productions (denoted by $\prec$) represent an interesting formal tool to specify dynamic attribute dependencies. Incremental evaluation is a useful consequence of dependency.

References

[1] D.E. Knuth, “Semantics of context-free languages,” *Math. Systems Theory,* vol. 2, no. 2, pp. 127-145, 1968. Correction in *Math. Systems Theory,* vol. 5, no. 1, pp. 95-96, 1971.
[2] A. V. Aho, M. Lam, R. Sethi & J.D. Ullman, “Compilers: principles, techniques, and tools,” Addison Wesley, 2007.

[3] H. Alblas, and B. Melichar (eds), *Attribute Grammars, Applications and Systems.* Lecture Notes in Computer Science 545, Prague, Springer-Verlag, June 1991.

[4] P. Deransart, M. Jourdan, and B. Lorho, *Attribute Grammars: Definitions, Systems and Bibliography.* Lecture Notes in Computer Science, vol. 323, New York: Spinger-Verlag, 1988.

[5] P. Deransart, and M. Jourdan, *Attribute Grammars and their Applications.* Lecture Notes in Computer Science, vol. 461, Paris: Springer-Verlag, 1990.

[6] J. Engelfriet, *Attribute grammars, Attribute evaluation methods.* Methods and Tools for Complier Construction, Ed. B. Lorho, New York: Cambridge University Press, 1984, pp. 103-138.

[7] M. Jazayeri, W.F. Ogden, and W.C. Rounds, "The intrinsically exponential complexity of the circularity problem for attribute grammars," *Communications of the ACM,* vol. 18, no. 12, pp. 697-706, Dec. 1975.

[8] M. Jazayeri, "A Simpler Construction for Showing the Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars," *Journal of the ACM,* vol. 28, no. 4, pp. 715-720, Oct. 1981.

[9] J.M. Dill, "A Counterexample for A Simpler Construction for Showing the Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars," *Journal of the ACM,* vol. 36, no. 1, pp. 92-96, Jan. 1989.

[10] B. Courcelle, *Attribute grammars: Definitions, analysis of dependencies, proof methods*. Methods and Tools for Compiler Construction, New York: Cambridge University Press, 1984, pp. 81-102.

[11] D. Parigot, G. Roussel, M. Jourdan, and E. Duris, *Dynamic attribute grammars*. Tech. Rep. 2881, INRIA, Rocquencourt, France, 1996.

[12] Y. Kikuchi, and T. Katayama, "On generalization of attribute grammars," *Systems and computers in Japan,* vol. 27, no 9, pp. 33-42, 1996.

[13] F. Neven, "Attribute grammars for unranked trees as a query language for structured documents," *Journal of Computer and System Sciences*, vol. 70, no. 2, pp. 221-257, March 2005.

[14] B. Courcelle, and P. Deransart, "Proofs of partial correctness for attribute grammars with application to recursive procedures and logic programming," *Information and computation,* vol. 78, no. 1, pp. 1-55, July 1988.

[15] P. Deransart, and J.A. Maluszynski, *A grammatical View of Logic Programming.* MIT Press, Nov. 1993.

[16] J. Paakki, "Attribute grammar paradigms-a high-level methodology in language implementation," *ACM Computing Surveys (CSUR),* vol. 27, no. 2, pp. 196-255, June 1995.

[17] L. Barford, and B.T. Vander Zanden, *Attribute grammars in constraint-based graphics systems*. Tech. Rep. TR87- 838: Department of Computer Science, Cornell University, Ithaca, New York, 1987.

[18] J. Steele, *The Definition and Implementation of a programming Language Based on Constraint.* PhD thesis: MIT Artificial Intelligence Laboratory, 1980.

[19] B.T. Vander Zanden, *Incremental Constraint Satisfaction and its Application to Graphical Interfaces*, Tech. Rep. TR88-941: Department of Computer Science, Cornell University, Ithaca, New York, October 1988.

[20] J. Maluszynski, *Attribute Grammars and Logic Programs: A Comparison of Concepts*. Eds. H. Albas and B. Melichar, Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science 545, Springer-Verlag, 1991, pp. 330-357.

[21] D. Batory, "Feature Models, Grammars and Propositional Formulae," *in Proc. of the 9th Software Product line Conference (SPLC 2005)*, Springer LNCS 3714, 2005, pp. 7-20.

[22] T. Isakowitz, *Can we transform logic programs into attribute grammars?* Stern School of Business, New York University, 1991, http://archive.nyu.edu/bitstream/2451/14362/1/IS-91-06.pdf.

[23] M. Ruffolo, and M. Manna, "A Logic-Based Approach to Semantic Information Extraction," i*n Proc. of the 8th International Conference on Enterprise Information Systems (ICEIS'06)*, 2007, pp. 70-84.

[24] R. Hoover, *Incremental Graph Evaluation.* PhD thesis: Department of Computer Science, Cornell University, Ithaca, New York, 1987.

[25] S.E. Hudson, "Incremental attribute evaluation: A flexible algorithm for lazy update," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 13, no. 3, pp. 315-341, July 1991.

[26] T. Reps, T. Teitelbaum, and A. Demers, "Incremental context-dependent analysis for language based editors," *ACM Transactions on Programming Languages and Systems,* vol. 5, no.3, pp. 449-477, July 1983.

[27] P. Deransart, *Validation des grammaires d'attributs*. PhD thesis: Université de Bordeaux I, 1984.

[28] K. Barbar, and K. Musumbu, "Implementation of interpretation algorithm by means of attribute grammars," *in Proc. of the 26th South-eastern Symposium on system Theory, IEEE Computer Society*, 1994, pp. 87-93.

[29] M.M. Corsini, K. Musumbu, A. Rauzy, and B. Le Charlier, *Efficient bottom-up abstract interpretation of logic programs by means of constraint solving over symbolic finite domains*. Tech. Rep: Institute of Computer Science, University of Namur, Belgium, 1993.

[30] K. Marriott, H. Sondergaard, and N.D. Jones, "Denotational abstract interpretation of logic programs," *ACM Transactions on Programming Languages and Systems,* vol. 16, no. 3, pp. 607-648, May 1994.

Open Science Index, Computer and Information Engineering Vol:4, No:1, 2010 publications.waset.org/1245.pdf

Kablan Barbar was born in Chettaha, Lebanon on February 22nd, 1955. He received a Master degree in computer science from the University of Bordeaux I in 1979. He holds Ph.D in Computer Sciences from the University of Bordeaux I since 1980.

He taught at the University of Bordeaux I between 1980 and 1996. He is currently a full professor at the Faculty of Sciences of the Lebanese University and director of the Lebanese University's law center. His current research interests include attributed grammars, compiling of markup languages and automatic generation of web applications.

May Dehayni was born in Gebaa, Lebanon on March 25th, 1977. She received a Master degree in computer science from the Lebanese University in 2000. She received a Ph.D in computer science from the University of Toulouse III (UPS), France in 2004.

She is an Assistant Professor in computer science at the Faculty of Sciences of the Lebanese University, Hadath, Lebanon. Her previous research and publications have been in the field of meta-modeling (MOF). Her Ph.D. thesis proposes an approach of model transformation based on attribute grammars. She is currently working in the domains of Information Research Systems, and combining Attribute Grammars with logic.

Ali Awada was born in Baalbeck, Lebanon on October 30th , 1964. He received a Master degree in computer science from the University of Toulouse III (UPS), France in 1988. He received a Ph.D in computer science from the "Institut National Polytechnique" of Toulouse, France in 1993.

He is a Professor in computer science at the Faculty of Sciences of the Lebanese University, Hadath, Lebanon. He previously worked on human/machine communication in natural language. His current research interests include semantic distance, using ontology in the process of request expansion in Information Research Systems, and combining Attribute Grammars with logic.

Mohamad Smaili was born in Aïta-Fakhar, Lebanon on December 2nd, 1961. He received a Master degree in computer science from the University of Montpellier II (USTL), France in 1990. He received a Ph.D in computer science from the University of Montpellier II in 1993.

He is a Professor in computer science at the Faculty of Sciences of the Lebanese University, Hadath, Lebanon. His research interests are fuzzy logic, digital circuits, and simulation.