Facilitating the Development of Parallel Implementations of Declarative Programming Languages Using Attribute Grammars

Aggelos M. Thanos and George Papakonstantinou

National Technical University of Athens
Department of Electrical and Computer Engineering
9 Iroon Polytechniou str. 15773 Zografou Campus, Hellas
Tel: +30 1 7721530, Fax: +30 1 7722496
thanos@cslab.ece.ntua.gr

Abstract

This paper presents how we can model different control operations on parallel implementations of declarative programming languages. We use a program analysis method based on Attribute Grammar dependency graphs. We present the PAGE[41, 37] framework, which facilitates the development of parallel implementation of declarative languages. As in compiler technology, we use AG as a specification language for the description of the programming paradigms under consideration. Each of the programming paradigms is outlined from a transformation table or a combination of them. These tables consist of transformation actions that have to be applied, under some conditions. The transformations are described in the form of AG semantic rules. With this analysis we can discern a large part of the control of such languages which we can specify in a programmable way. The remain control part is forming a non-programmable layer which is following the restrictions of the underlying hardware architecture of each implementation. In this way we build a layer between the program executed and the control. Using AG technology to specify this control layer is the semantic basis of PAGE. The method can help towards the automation of the development of modern declarative programming languages, such as Concurrent Constraint Logic Programming Languages. The system has been implemented and tested in a wide range of architectures, exhibiting encouraging results.

1. Introduction

Functional approaches to Knowledge Representation (KR) [24, 4] have led to the introduction of control primitives in the form of computation in some domains other than Herbrand Universe. One of the primary advantages of this approach is the ability of the system to use efficient non-deductive computational methods, suitable for each of these domains. Using such a method, the expressiveness of the language is increased in the sense that we no longer need to encode all the concepts and operations of interest in the form of Herbrand terms. At semantical level, computation can still be considered as deduction in some class of the pertinent models, creating this way a uniform terrain for programming with terms and operations ranging over different domains. Functional Logic Programming (FLP) [13, 14] and Constraint Logic Programming (CLP) [17] represent two widely known programming paradigms falling in the above framework.

The CLP paradigm replaces the computational heart of LP, i.e., unification, by constraint consistency checking leading to a *constrain and generate* schema, instead of the *generate and test* schema of LP. This prioritization of a deterministic process (constraint solving) over a non-deterministic process (search) is the heart of the CLP paradigm. *Acceptance, rejection*, and *entailment* are the fundamental control operations on constraints. Again not sufficiently instantiated constraints lead to delays on the outcome of these operations and sometimes lead to incomplete operational semantics. Based on this principle, some well known systems have been developed: CLP($\mathcal{R}$) [18], CHIP [10, 11], PrologIII [5], AKL [15]. An early approach towards CLP has been presented in [32], where a sequential method was utilized.

In both LP and Constraint Logic Programming (CLP) control conditions and operations are sensitive to the availability of certain information. In order to become applicable, a rule must assure that some information is already present in the head predicate. This leads to the introduction of a notion of *directional information flow* which is the principal control mechanism in the above two paradigms. To verify that a program is operationally complete, static methods have been proposed which off-line analyze the program at hand, and which are usually founded on the relationship between LP and AG [2, 1, 7, 8]. Indeed, the intimate relationship between logic and AG [30, 8, 7] has led many researchers to implement tools and methods for Logic Programming or Functional Logic Programming (FLP) based on techniques of grammar evaluation [31, 29, 27, 28, 3].

In order to incorporate the above considerations into a fully distributed model, a multi-agent approach is needed, where each agent corresponds to a particular process implementing a subgoal. Moreover, to provide operational completeness, shared variables must be synchronized between agents so as to ensure consistency. This, in turn, amounts to a parallel evaluation of the respective AG [19].

This paper describes a method for unifying several parallel programming paradigms with AGs. The method is based on PAGE [41], a general tool for parallel attribute grammar evaluation. As in compiler technology, we use AG as a specification language for the description of the programming paradigms under consideration. Each of the programming paradigms is outlined from a transformation table or a combination of them. These tables consist of transformation actions that have to be applied, under some conditions. The transformations are described in the form of AG semantic rules. In this way we built a layer between the program executed and the control. According to Kowalski[23], Program = Logic + Control. In this work we go one step further and we decompose control into *control rationale* and *control principles*. While the principles component of the control is a set of mechanisms depending on the underlying implementation of the system (computational model), control rationale component is a set of specifications described via proper semantic rules and reflects the special operational semantics of the programming paradigm under consideration. Using AG technology to specify the control rationale is the semantic basis of PAGE.

2. Semantic Foundations

2.1. Attribute Grammars

Attribute Grammars have been proposed by Knuth [21, 22] as an extension of context-free grammars. The original motivation has been to facilitate the compiler specification and development procedure. The formalism became a useful tool for modeling pass-wise compilation strategies.

While the compilers area was the initial research area, AGs can also be used in a very wide research spectrum, where relations and dependences among structured and interpreted data are very valuable.

Definition 2.1 A *context-free grammar* G is a quadruple such that $G = < N, T, P, D >$, where N is a finite set of *nonterminal symbols*, T is a set of *terminal symbols*, P is a finite set of *productions*, and $D \in N$ is the *start* symbol of G. □

An element in $V = N \cup T$ is called *grammar symbol*. The productions in P are pairs of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in V^*$, i.e., the left hand side symbol (LHSS) X is a nonterminal, and the right hand side symbol (RHSS) α is a string of grammar symbols. An empty RHSS (empty string) is denoted as ε.

Definition 2.2 An *Attribute Grammar* consists of three elements, a *context-free grammar* G, a finite set of *attributes* A and a finite set of *semantic rules* R. Thus $AG =< G, A, R >$. □

A finite set of attributes $A(X)$ is associated with each symbol $X \in V$. The set $A(X)$ is partitioned into two disjoint subsets, the *inherited attributes* $I(X)$ and the *synthesized attributes* $S(X)$. Thus $A = \cup A(X)$.

The production $p \in P$, $p : X_0 \rightarrow X_1 \cdots X_n$ (n ≥ 0), has an *attribute occurrence* $X_i.a$, if $a \in A(X_i), 0 \leq i \leq n$. A finite set of *semantic rules* R_p is associated with the production p with exactly one rule for each synthesized attribute occurrence $X_0.a$ and exactly one rule for each inherited attribute occurrence $X_i.a$, $1 \leq i \leq n$.

Thus R_p is a collection of semantic rules of the form $X_i.a = f(y_1, \cdots, y_k)$, $k \geq 0$, where

1. either $i = 0$ and $a \in S(X_i)$, or $1 \leq i \leq n$ and $a \in I(X_i)$
2. each $y_j, 1 \leq j \leq k$, is an attribute occurrence in p, and
3. f is a function, called *semantic function*, that maps the values of $y_1, \ldots, y_k$ to the value of $X_i.a$. In a semantic rule $X_i.a = f(y_1, \ldots, y_k)$, the occurrence $X_i.a$ *depends* on each occurrence $y_j, 1 \leq j \leq k$.

Thus $R = \cup R_p$. By definition, synthesized attributes are *output* to the LHSS of the productions, while inherited attributes are *output* to the RHSS. In other words, synthesized attributes move the data flow *upwards* and inherited attributes move the data flow *downwards* in the derivation tree during the attribute evaluation procedure.

Remark: Notice that each semantic rule of an attribute grammar can be seen as a definition of the relation between local attribute values of the neighboring nodes of the parse tree. This relation is defined for a production rule and should hold for every occurrence of this production rule in any parse tree. In the original definition of AG (definition 2.2) the definition of the relation has the form of the equation. It is possible to generalize the concept of semantic rules and allow them to be arbitrary formulae (not necessarily equalities) over a language $\mathcal{L}$.

Definition 2.3 A *Conditional Attribute Grammar* (CAG) is an attribute grammar with an extended concept of the semantic rules. Thus a CAG is a 5-tuple $< G, S, A, \Phi, \mathcal{I} >$, where:

- G is the underlying context free grammar
- S is a set of sorts (i.e. of domains in which the attributes take values)
- A is a finite set of attributes. Each attribute a has a sort $s(a)$ in S.
- Φ is a map function of a logic formula Φ_p written in terms of an S-sorted logic language $\mathcal{L}$ to each production rule $p \in P$. The variables of a formula Φ_p include all *output* attribute occurrences of $A(p) = \cup_{X \in P} A(X)$. The allowed form of the function Φ_p is either a function f or a relation c.
  - in case of function, f has the form
    $f : X_{p,k}.a = f(X_{p,k_1}.a_1, \ldots, X_{p,k_m}.a_m)$
    where $f : \mathcal{I}(X_{p,k_1}.a_1) \times \ldots \times \mathcal{I}(X_{p,k_m}.a_m) \rightarrow \mathcal{I}(X_{p,k}.a)$.
  - in case of relation, c has the form
    $c : c(X_{p,k_1}.a_1, \ldots, X_{p,k_m}.a_m)$
    where $c : \mathcal{I}(X_{p,k_1}.a_1) \times \ldots \times \mathcal{I}(X_{p,k_m}.a_m) \rightarrow \{true, false\}$.
- $\mathcal{I}$ is an *interpretation* of $\mathcal{L}$ in some S-sorted algebraic structure $\mathcal{A}$. □

- `the underlying CFG nonterminals are taken from the LP clause predicates`
- `every nonterminal` p `of the AG has an associated set of attributes` $\mathcal{A}$ `corresponding to the arguments of the LP predicate and which separates in two subsets` $I(p)$ `and` $S(p)$`, depending on which of them is input or output argument in the original LP clause.`
- `for each output attribute occurrence we associate a semantic rule which computes the most general unifier of a set of equations on terms between the output attribute occurrence and some input attribute instances of that attribute`

Figure 1: AG Construction Rules for LP Equivalence

Semantic rules induce dependences between attributes. These dependences can be presented by a *dependency graph*, from which partial ordering relations are implied. From these partial orderings the *evaluation order* of the attribute occurrences can be determined. A *decorated tree* is a derivation tree in which all the attribute occurrences have been evaluated, according to their associated semantic rules. The dependency graph characterizes all restrictions on the control of computations. The actual sequence of attribute evaluation must preserve this ordering which is called *attribute evaluation strategy*. Attribute grammars can be classified according to the respective attribute evaluation strategy. A special class, introduced by Knuth [21], are the $S-$ *attributed* grammars in which only synthesized attributes are allowed.

Due to the restrictive form of the S-attributed grammars, $L-$ *attributed* grammars are used in practice. In these grammars any inherited attribute associated with a symbol $X_{p,j}$ of a production p depends either on the synthesized attributes of the preceded RHSSs $X_{p,k}$ (e.g., $1 \leq k < j$) or on the inherited attributes of the LHSS $X_{p,0}$ of the production. The synthesized attributes of the LHSS of the production depend either on the inherited attributes of the same symbol or on the synthesized attributes of RHSSs.

Definition 2.4 An attribute grammar is said to be *L-attributed* if and only if each inherited attribute of $X_{p,j}$ in the production $p : X_{p,0} \rightarrow X_{p,1}, \ldots, X_{p,np}$ depends only on the set $\bigcup_{1 \leq k < j} S(X_{p,k}) \cup I(X_{p,0})$ for $j = 1, \ldots, np$. Any synthesized attribute of the symbol $X_{p,0}$ depends only on the set $\bigcup_{1 \leq k \leq np} S(X_{p,k}) \cup I(X_{p,0})$. □

2.2. AGs and Logic Programming

Operational semantics of Logic Programming can be modeled by AG derivation trees. This relationship has been explored in the seminal work of [7, 8]. This approach is based on the simulation of SLD resolution scheme with the attribute evaluation strategy of AGs. With the proper transformations, an AG can become equivalent to a Logic Program in the sense that a decorated tree can bare the computations led to a most general unifier. The basic transformations used are outlined in the AG construction rules given in Fig. 1.

Of course, semantic rules are some times undefined, i.e.,the most general unifier does not exist for some set of equations. For this, the constructed AG can be seen as a CAG. Whether or not the constructed AG can always lead to a most general unifier is subject to special conditions that have to hold in the constructed AG. One of the most well known conditions is that the generated AG have to be L-attributed and the root of every rule must have a synthesized attribute which is maximal, according to the L-ordering.

Based on the above rules we will see in the following sections that we can also model Parallel Logic Programming, Functional Logic Programming, Concurrent Constraint Logic Programming, and more general classes of modern declarative parallel programing paradigms. Hidden in this paper is the belief that with a proper definition of the Φ component of the constructed CAG in terms of an S-sorted logic language $\mathcal{L}$, we can form a layer between logic and control, that we call *control rationale*. In this layer we can specify the operational semantics of a programming paradigm [38]. The idea of expressing the control information by its own, carefully designed language is not new but is prevalent for a long time [16]. Constraint Logic Programming Paradigm is a major representative of languages following this perspective. Here, AGs give us the semantic framework to realize these ideas by giving a more clear description of how can we "control the control".

2.3. Functional Logic Programming

A Functional Logic Program[1] (FLP) is like an ordinary logic program except that the functors are divided into two disjoint sets: ordinary *constructors*, and *defined* symbols. A program may contain both *constructor terms* and *functional terms* whose principal functor is a defined symbol. Each defined symbol f with arity n is assumed to be associated with a function (external), taking n ground constructor terms as input and yielding one ground constructor term as output. Thus, a ground functional term $f(t_1, \cdots, t_n)$ represents a ground constructor term that we denote with $f(t_1, \cdots, t_n) \downarrow$. This reduced value is obtained by calling the external function for f with $t_1, \cdots, t_n$ as input.

2.4. Concurrent Constraint Programming and Basic Control Operations

A Concurrent Constraint Programming Language[2] (CCP) assumes the existence of a set of constraints. Simply put, constraints are first order formulas over some domain of discourse of some theory Σ (described by a signature $\mathcal{L}$). This set of constraints is assumed to be closed under conjunction, and the formulas must have free variables. Over a Concurrent CCP language two basic control operations apply: The *Tell* operation asserts a new constraint into the constraint store, providing that the former is consistent to all other constraints in the store. In other words in every model of Σ it is possible to find assignments of the free variables of the formula, under which the formula is consistent. The *Ask* operation asks whether a constraint is logically entailed from the constraint store. Applying an *Ask* operator might yield `true`, `false`, and `do not know yet`. In the latter case, more constraints have to be stored in the constraint store with *Tell* operations for getting a more specific answer (`true` or `false`). Consequently, *Ask* operation suspends the computation until some *Tell* operation add more information in the store. *Tell* and *Ask* operations have been introduced by Saraswat [34] after Milner [26] and Levesque [24]. By these two basic operations on constraints, i.e., Ask and Tell, we can formulate a framework of computation with synchronization on variables of constraints.

3. Modeling Parallel Paradigms with AGs

3.1. Parallel AG Evaluation

One of the most interesting properties of AGs is that they are tractable to parallel implementations. Recall that attribute dependences form dependency graphs, from which partial ordering relations are implied. The key idea is that attribute instances belonging to different orderings (instances that are not linked by the same dependency graph) may be computed concurrently. For example instances belonging to parse trees of two different clauses of the the same procedure (OR-parallelism) may be computed in parallel. Of course when fine grain parallelism is a requirement, then it is possible to have interleaved computation of attributes in the same dependency graph (AND-parallelism) by

[1] For strict definitions, the reader should refer to [9, 25, 13].
[2] For an introduction to Constraint Programming the reader should refer to [34, 33, 17]

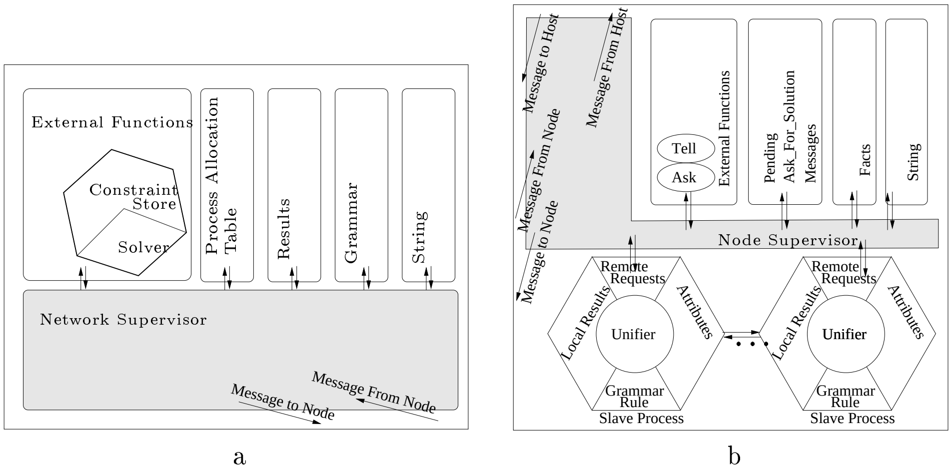


Figure 2: (a) Network Supervisor and (b) Node Supervisor structure.

proper message interchange. Many different approaches have been proposed for parallel AG evaluation [19], basically emanated from the compiler technology field and based on static analysis for useful parallelism detection.

In the context of Logic Programming, a parallel implementation of an AG evaluator would be a nice testbed for experimenting with various LP operational semantics. Towards this direction we have implemented PAGE [41, 37, 42], a Parallel AG Evaluator.

3.2. PAGE

Based on the inherent tractability of AG to parallel implementations we have developed PAGE; a Parallel AG Evaluator. Our main target was not just to give another Parallel AG Evaluator but to build a tool for experimenting with various LP operational semantics.

In PAGE, a *supervisor* process maintains a pool of messages, and it is responsible for supplying the processors with computational load (i.e., processes to execute). The Supervisor controls the initialization and termination of the evaluation. The grammar rules and the terminal symbols (or facts in the case of LP) are stored in the Supervisor. Additionally, the supervisor assigns the evaluation of new rules to *slave* processes and collects all the intermediate solutions. Each slave process, or agent, is distributed over a network of processors and handles a grammar production along with its semantic rules. It collects messages corresponding to the RHSS (body) of the production in which it is the head, and produces new messages, after proper unification, which correspond to the head of the production in which it is a RHSS.

We have to notice here that the evaluation order is preserved, in the sense that no slave is spawned if it is not assured that all the inherited attributes of the rule are bound and so the unifier can procceed. Recall that the evaluation order is implied by the partial ordering induced by the dependency graph corresponding to the AG under evaluation. Moreover, the unification procedure is built-in in the agent, instead of writing it specifically in the S-sorted language $\mathcal{L}$. We will see in the next sections that PAGE can support externally linked functions written in a convenient language (e.g., C), which we assume that is a subset of $\mathcal{L}$. This simple feature expands greatly the expressive power of PAGE. All the static information of the grammar (grammar productions, semantic rules, atomic productions) is broadcast and kept in the network processors for faster access. All the information generated from the slave processes (i.e., partial solutions) is stored locally in the network processors, in a caching hierarchy, inducing an *incremental attribute evaluation* and achieving a *controlled grained parallelism.*

For example, let us assume that we have to evaluate the following grammar:

Example 3.1

```
1.   S(x,y,z) :- A(x,y), E(y,z).
2.   A(x,y)   :- B(x), C(y).
3.   E(y,z)   :- B(y), D(z).
```

where S is the start symbol.

Supervisor assigns to a new slave (for example slave $s1$) the evaluation of rule `1`. Slave $s1$ asks the Supervisor for solutions for the RHSS `A` and `E` along with their corresponding inherited attributes. Supervisor checks if there exist already solutions for `A(x,y)` and `E(y,z)`. If not, it assigns to two new slaves (say $s2$ and $s3$) the evaluation of `A(x,y)` and `E(y,z)` respectively. □

In the above example we skipped the semantic rules of each production for convenience. Nevertheless, when a goal is assigned to an agent for evaluation, the semantic rules are performed once. Then, each time a solution of a subgoal arrives from the network the semantic rules are re-applied on the affected attributes to capture the new status established by the attribute bindings the new solution carries with. When more than one values are bound to a single attribute, we say that a *binding conflict* has occurred. Such cases are avoided by performing proper, post-evaluation, consistency checking semantic rules.

In PAGE we can use externally linked procedures or functions as semantic rules. There are three special purpose functions that are built-in with PAGE; `Tell(c)`, `Ask(c)`, and `Solve(c)`. `Tell` sends a constraint to a central constraint store provided that it is consistent with it. On success returns `true` along with potential bindings, otherwise returns `false`. `Ask` queries the store for the entailment of a constraint. On success returns `true`, otherwise returns `false`. Store is implemented through the function `Solve`, which is invoked each time `Tell` or `Ask` are executed. `Tell` and `Ask` cooperate with `Solve` in a client-server fashion, under a special message passing protocol for interchange constraints implemented in PAGE. These built-in functions can be overloaded by the user, giving the opportunity to implement and test PAGE with special purpose constraint solvers.

The described model achieves AND parallelism. If there already exist solutions for some or all RHSSs, then there is no need for new slaves for these RHSSs. Moreover, with proper transformation of the program into the corresponding AG, as we will see later, the respective evaluation order can impose waits or even prunings in the evolution of the goal subtrees. As a result, PAGE achieves *controlled grained parallelism*, because no unnecessary slaves are generated.

In the example 3.1 slave $s2$ asks the supervisor for solutions for the RHSS `B` and `C` along with their inherited attributes (i.e., `x` and `y` respectively). Supervisor checks if there exist already solutions for `B(x)` and `C(y)`. If not, it assigns to two new slaves (say $s4$ and $s5$) the evaluation of `B(x)` and `C(y)`, respectively. The procedure goes on in the same manner unfolding a proof tree over the network. If we add one more rule to our program

```
4.   S(x,y,z) :- A(y,x), E(z,y).
```

then another similar proof sub-tree will be generated, establishing OR parallelism.

Fig. 3 depicts the above example. The slaves may be located in different or the same processors.

In general, the data-flow style of execution of PAGE bares resemblance to the computational model of Conery's AND/OR process model [6], however it differs in the following important aspects: We use the aforementioned structure sharing approach so there is no need for stack copying which is a major source of overhead in the latter model. Solution caching for incremental attribute evaluation greatly improves our model. Moreover, our schema lies between the methods proposed by Fang [12] and Kaplan and Kaiser [20]. Indeed, we use a per rule-agent correspondence instead of attribute-agent of Fang's. Furthermore, the above per rule-agent matching achieves a data-splitting, though not in

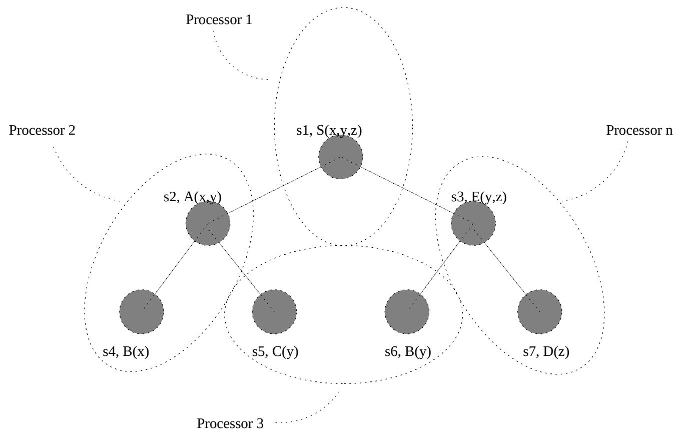


Figure 3: Slave processes generation for the example 3.1.

the size of the Kaplan and Kaiser method. Finally we use an interleaved unification schema in which more than one solutions of each AND parallel branch are unified at the same time [42].

We have implemented PAGE on top of ORCHID [39, 40], a parallel programming platform which encapsulates the machine dependences and provides a layer with parallel programming primitives. ORCHID masks the operating system dependences and provides some advanced facilities such as transparent asynchronous process-to-process communication with message passing, load balancing, broadcasting, virtual topologies, dynamic process allocation, global synchronization (global barriers and semaphores) and distributed shared memory. ORCHID supports the well-known distributed programming interfaces PVM and MPI, and extends them [36, 38] for multi-threaded programming. The system has been implemented and tested in a wide range of architectures, exhibiting encouraging results. The supported platforms include high-speed networks of Sparc and x86 workstations (running Solaris and/or Windows 95/NT), Parsytec Power Mouse, Power Xplorer, and GCEL512 Multiprocessors.

3.3. Modeling Parallel LP using AGs

As mentioned above, relating LP and AG is subject to performing the appropriate transformations of the original program and dynamically annotate the attribute occurrences and decorate the derivation tree during the execution [29]. In our approach we simulate parallel LP with parallel AG evaluation using PAGE. The general construction rules of the corresponding AG are outlined from transformation tables. These tables consist of transformation actions that have to be applied, under some conditions. Roughly speaking, these tables express the same rationale with the rules of Fig. 1, though somewhat more informally. According to the kind of parallelism we want, we use a corresponding class of transformation rules for transforming the logic program into an AG; in other words to produce a different set Φ of semantic rules. With these transformation rules we do nothing but the construction of different dependency graphs for the same logic program. Thus we can give different operational semantics to the same logic program. We have organized each class of these transformation rules in tables. Tables 1 (a) and (b) are used for transforming an LP into an equivalent AG for Dependent AND Parallelism(DAP) and Stream AND Parallelism (SAP), respectively.[3]

[3]There are two main forms of AND parallelism that overcome the problem of binding conflict: DAP removes binding conflicts by a consistency check operation after the (parallel) evaluation of the subgoals. SAP synchronize the evaluation of the goals in a way that conflicts can never occur (see transformation action $P4$ in Tables 1(a) and 1(b).)

(a)	CONDITION	ACTION
P0	t^e_{gi} is a constant c and $g == 0$	**if** $ia_i(R^e_g) \neq c$ and $ia_i(R^e_g) \neq nil$ **then** cancel this evaluation request **else** $sa_i(R^e_g) = c$
P1	t^e_{gi} is a constant c and $g \neq 0$	$ia_i(R^e_g) = c$
P2	t^e_{gi} is a variable and $g == 0$	**if** t^e_{gi} is the only occurrence of the variable in the production **then** $sa_i(R^e_g) = ia_i(R^e_g)$ **else if** t^e_{qj} is the last textual occurrence of the same variable in the production **then** $sa_i(R^e_g) = sa_j(R^e_q)$
P3	t^e_{gi} is a variable and t^e_{qj} is another textual occurrence of the same variable such that $q == 0$ and $g \neq 0$	$ia_i(R^e_g) = ia_j(R^e_q)$
P4	t^e_{gi} is a variable and t^e_{qj} is the nearest textual occurrence of the same variable to the left ($q < g$) such that $q \neq 0$ and $g \neq 0$	$sa_i(R^e_g) == sa_j(R^e_q)$
P5	there are many occurrences of the same variable in a rule R^e_g and $g \neq 0$	**if** all the variables have not the same value or those which are different are not nil **then** cancel this evaluation request **else** make those which are nil take the values of the others
(b)	**CONDITION**	**ACTION**
P0	t^e_{gi} is a constant c and $g == 0$	**if** $ia_i(R^e_g) \neq c$ and $ia_i(R^e_g) \neq nil$ **then** cancel this evaluation request **else** $sa_i(R^e_g) = c$
P1	t^e_{gi} is a constant c and $g \neq 0$	$ia_i(R^e_g) = c$
P2	t^e_{gi} is a variable and $g == 0$	**if** t^e_{gi} is the only occurrence of the variable in the production **then** $sa_i(R^e_g) = ia_i(R^e_g)$ **else if** t^e_{qj} is the last textual occurrence of the same variable in the production **then** $sa_i(R^e_g) = sa_j(R^e_q)$
P3	t^e_{gi} is a variable and t^e_{qj} is the nearest textual occurrence of the same variable to the left ($q < g$) such that $q == 0$ and $g \neq 0$	$ia_i(R^e_g) = ia_j(R^e_q)$
P4	t^e_{gi} is a variable and t^e_{qj} is the nearest textual occurrence of the same variable to the left ($q < g$) such that $q \neq 0$ and $g \neq 0$	$ia_i(R^e_g) = sa_j(R^e_q)$
P5	there are many occurrences of the same variable in a rule R^e_g and $g \neq 0$	**if** all the variables have not the same value or those which are different are not nil **then** cancel this evaluation request **else** make those which are nil take the values of the others

Table 1: Transformation Tables for (a) DAP and (b) SAP modeling.

Remark : In the transformation tables the notation t^e_{gi} is adopted when referring to either a constant term, a variable or a functional term, where e is the identification number of the program rule, g is the order in which the corresponding predicate symbol appears in the rule and i is the relative order in which the t^e_{gi} appears in the argument list of that predicate. For instance, the `X1` variable in the example 3.2 is denoted as $t^0_{3,0}$ (here *is* is considered as an ordinary predicate for the denotation of the index g). Additionally, if we order the variables of each predicate, the mappings $ia_i(R^e_g)$ and $sa_i(R^e_g)$ give the inherited and the synthesized attribute of the i^{th} variable corresponding to the predicate R^e_g. Similarly for the mappings $sa(v)$ and $ia(v)$, where v is a variable. Moreover, if within a clause R^e_g and R^e_q correspond to the same predicate, we can refer to them by their predicate name subscripted by their occurrence order. For example, fib_j means the j^{th} occurrence of the predicate fib in the rule (e.g., fib_0 denotes the head of the rule).

The produced AG is a form of L-attributed AG and consequently follows the rules for operational completeness described in [8]. Thus the transformations lead to a sound and operationally complete grammar.

Example 3.2 The Fibonacci series problem written in Prolog-like notation.

```
fib(X,F) :-  is(X2,X-2), fib(X2,F2),
```

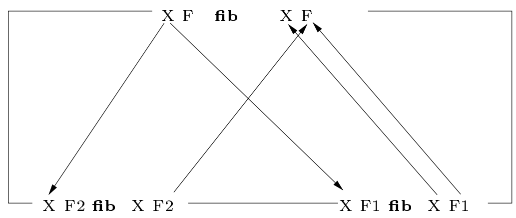


Figure 4: Dependency graph for AG produced by Table 1.

```
        is(X1,X-1), fib(X1,F1), add(F,F1,F2).

fib(0,1).
fib(1,1).
```
□

The rules described in Table 1(a) produce an AG. The dependences induced from this transformation are depicted in Fig. 4. In this figure inherited attributes are located on the left of the corresponding predicate while synthesized attributes are located on the right. The output AG uses two attributes, one inherited and one synthesized, for each one of the variables or the functional terms of the original program. This redundancy seems at first glance unnecessary if one takes into consideration the usual annotation methods used for describing the dependency graphs of AGs [8, 2, 27] in which each variable is characterized either synthesized or inherited in a static analysis procedure. However, our method does not strictly follows the static analysis stage. It takes also into account that the output AG is used for the specification of the operational semantics of a given program where the generated semantic rules induce dependences which, in run-time, consider each attribute occurrence as input or output according to the current context. Recall that in an AND/OR parallel environment many agents may compute simultaneously and an agent may considered that evaluates a rule which is either the head or a subgoal of the body of another rule.

This feature makes the tool suitable for scalable environments. Notice that in this figure we have omitted the predicates *is* and *add*, since we consider them as built-in predicates for assignment and addition respectively. We assume that once these built-in predicates have bound their "input" arguments, they immediately produce the corresponding result. For the moment we try to use pure declarative style in LP programming and avoid all that "impure" components; this is something we are to discuss in the next sections (functional terms, constraints).

Observing the dependency graph in Fig. 4, one can clearly see that the basic principle of the AG methodology has been followed; inherited attributes are output to the RHSSs, while synthesized ones are output to the LHSSs (see section 2.2). Recall that only those agents that correspond to rules that have all their inherited attributes bound can be spawned in the network for evaluation. Consequently, in the example, a parallel execution of the rules of the clause can be fired only if the variable X of the head of the clause is bound. Thus, as our computational model dictates, to preserve the evaluation order of the dependency graph all the user input arguments of the goal of the program correspond to the inherited attributes of that arguments (the minimal elements of the partial order induced by the dependency graph).

3.4. Modeling the Functional Logic Paradigm

Let us see now how PAGE can model the paradigm of FLP. The program in Example 3.3 is a functional logic program with defined symbols $+/\ 2$ and $-/\ 2$ (functional symbol / arity).

Example 3.3 The Fibonacci series problem written in Functional Logic Language.

	CONDITION	ACTION
F0	t^e_{gi} is a bound argument in the argument list of a function f and $g == 0$	**if** there are some unbound arguments in the argument list of f **then** insert in the production the semantic rule $sa(f) = f(\ldots, t^e_{gi}, \ldots)$ **else** insert in the production the semantic rule $sa(f) = ia(f) = f(\ldots, t^e_{gi}, \ldots) \downarrow$
F1	$\forall t^e_{gi} \mid t^e_{gi}$ is an unbound argument in the argument list of a function f and $g == 0$	replace the above semantic rule with $ia(f) = sa(f) = f(\ldots, sa_i(R^e_g), \ldots)$
F2	t^e_{gi} is a bound functional argument (i.e., $f = c$) and $g == 0$	insert in the production the semantic rule $ia(f) = c$
F3	t^e_{gi} is a bound argument in the argument list of a function f and $g \neq 0$	**if** there are some unbound arguments in the argument list of f **then** insert in the production the semantic rule $ia(f) = f(\ldots, t^e_{gi}, \ldots)$ **else** insert in the production the semantic rule $ia(f) = sa(f) = f(\ldots, t^e_{gi}, \ldots) \downarrow$
F4	$\forall t^e_{gi} \mid t^e_{gi}$ is an unbound argument in the argument list of a function f and $g \neq 0$ and $\exists t^e_{qi}$ occurrence of the same variable, $q = 0$	replace the above semantic rule with $ia(f) = sa(f) = f(\ldots, ia_i(R^e_g), \ldots)$. replace all the remaining arguments with $sa_i(R^e_g)$

Table 2: Transformation Table for Functional Logic Programming.

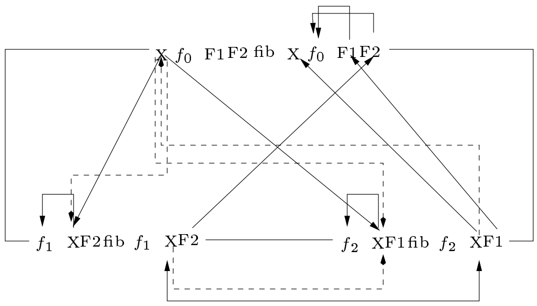


Figure 5: Dependency graph for AG in Table 3 (solid lines).

```
fib(X,F1+F2) :- fib(X-2,F2), fib(X-1, F1).

fib(0,1).
fib(1,1).
```
□

PAGE can model this programming paradigm introducing a new transformation table (Table 2) which is used in conjunction with the tables used for the LP paradigm. Now we consider that functional arguments have the same notational significance as the previously seen ordinary variables. In a similar way, they are also transformed into inherited and synthesized attributes. Functional arguments are prioritized in the unification procedure (the unification procedure becomes matching procedure since we are dealing with interpreted functional terms), so that when we have to unify a variable argument which is in the argument list of a functional argument we prefer to unify the latter and discard the former. This can easily be seen in Table 3, where the equivalent AG is given after the use of transformation Table 1 in conjunction with the transformation Table 2. Fig. 5 shows the dependency graph induced by the equivalent grammar in Table 3 (solid lines).

The idea behind these transformations looks familiar. Nothing has changed compared to the previous tables of the LP paradigms, except for our uniform notion of functional terms in the framework of AG. We allow functional terms to have inherited and synthesized attributes (note however that we do not make any distinction between them) and so to participate in the dependency

#	Semantic Rule	Table 1	#	Semantic Rule	Table 2
1	$sa(fib_0.X) = sa(fib_2.X)$	P2	7	$sa(fib_0.f_0) = f_0(sa(fib_0.F_1), sa(fib_0.F_2))$	F1
2	$sa(fib_0.F_1) = sa(fib_2.F_1)$	P2	8	$ia(fib_1.f_1) = f_1(ia(fib_1.X))$	F4
3	$sa(fib_0.F_2) = sa(fib_1.F_2)$	P2	9	$ia(fib_2.f_2) = f_2(ia(fib_2.X))$	F4
4	$ia(fib_1.X) = ia(fib_0.X)$	P3			
5	$ia(fib_2.X) = ia(fib_0.X)$	P3			
6	$sa(fib_1.X) == sa(fib_2.X)$	P4			

Table 3: The semantic rules of the equivalent AG for the Fibonacci series problem example 3.3 (FLP paradigm).

graphs. This is an important feature which permits the integrated view of the two paradigms; LP and FLP. Again here the same rules apply : Complete operational semantics is assured if we preserve the evaluation order of the corresponding AG.

3.4.1. Multi-pass execution (simple case)

The method described so far (Table 2 is used) is operationally incomplete when the minimal elements in the partial ordering induced by the generated dependency graph are unbound (for instance some of the arguments in the argument list of a functional argument are unbound). In such cases, a delayed binding mechanism has to be used. PAGE introduces a Multi-Pass schema using the transformation Table 4 in conjunction with the tables previously used. This table produces new semantic rules in the production inducing an R-annotated dependency graph (Right to Left dependences). These semantic rules are used as soon as possible (when new incoming solutions induce new inherited attributes) and not only when the search procedure has no more solutions to give. The interleaving of different passes is possible due to parallelism. To see how the Multi-Pass method works, we will take the following example.

Example 3.4 Let us have the following system of equations

$$\begin{aligned} D &= C \times 2 \\ K &= B \times 3 \\ C &= A + 1 \\ B &= D + (-3) \end{aligned}$$

declaratively expressed by the program

```
S(A,B,K) :- mul(D,C,2), mul(K,B,3), add(C,A,1), add(B,D,-3).
? S(2,B,K).
```

In Fig. 6 we can see the dependency graph for the equivalent AG corresponding to Table 5 generated after the the use of Table 1 in conjunction with Table 4. Here, we do not have functional arguments and so we do not apply the transformation Table 2. Arrows corresponding to Table 1 are designed with solid lines, while arrows corresponding to Table 4 are designed with dashed lines. Note that semantic rules 16 and 17 invalidate semantic rules 9 and 10 after the re-initialization of the inherited attributes in rules 14 and 15. See also the cyclic dependences introduced by the new inserted (dashed lines) dependences of Table 4. This cyclic graph depicts the data-flow between the passes. Each cycle represents a different pass.

□

Remark: The method of multi-pass execution is not new. It has been used for static analysis of operational completeness of functional logic programs [27, 3]. However, there is no method suggested

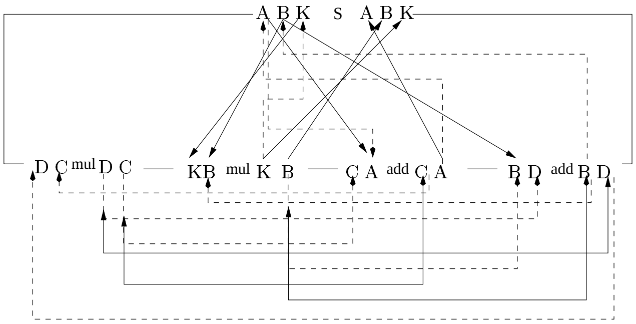


Figure 6: Dependency graph for AG of example 3.4 in Table 4.

	CONDITION	ACTION
M0	t^e_{gi} is a variable and $g == 0$	**if** t^e_{qj} is the last textual occurrence of the same variable in the production **then** $ia_i(R^e_g) = sa_j(R^e_q)$
M1	t^e_{gi} is a variable and t^e_{qj} is the nearest textual occurrence of the same variable such that $q == 0$ and $g \neq 0$	$ia_i(R^e_g) \ = \ ia_j(R^e_q)$
M2	t^e_{gi} is a variable and t^e_{qj} is the nearest textual occurrence of the same variable to the left ($q < g$) such that $q \neq 0$ and $g \neq 0$	$ia_i(R^e_g) \ = \ sa_j(R^e_q)$
M3	there are many occurrences of the same variable in a goal R^e_g and $g \neq 0$	**if** all the variables have not the same value or those which are different are not nil **then** cancel this pass **else** make those which are nil take the values of the others

Table 4: Transformation Table for dynamic Multi-Pass Execution.

for parallel execution framework. The method we propose here dynamically detects new evaluation orders, based on the current circumstances and constraints and enforcing parallel execution of agents.

The method increases the operational completeness of PAGE but it still remains incomplete under special circumstances, especially when new sets of inherited attributes cannot be entailed because of term interpretation restrictions. For example if we try to answer the query `Fib(N,6765)` in the example 3.3 it is impossible to give an answer. Indeed, the M0 rule of Table 4 which is responsible for the generation of new inherited attributes, and so new passes, can not evaluate the synthesized attribute $sa(fib_2.X)$ since $fib_2.X$ can never bind its synthesized attribute. Only f_2 (functional argument) can do this. In such cases, constraint solving techniques have to be used.

3.5. The Concurrent Constraint Logic Programming Paradigm

Constraint Logic Programming (CLP) is a programming paradigm emerged to overcome the difficulties of Logic Programming by enhancing the latter with constraint solving mechanisms [17]. The hardest problems in LP are the restrictive nature of unification, since it deals with uninterpreted structures, and the unbearable search tree evolution during the *generate and test* resolution procedure which is the basic obstacle for efficient implementations.

CLP uses terms interpreted over a domain of discourse which makes the programming schema more expressive and which, in turn, replaces the computational heart of LP, (i.e., unification), by constraint solving. Especially in the case of Concurrent Constraint Logic Programming constraint solving is used as source of synchronization through the primitive operations blocking *Ask* and atomic *Tell*, as described in section 2.4.

#	Semantic Rule	row of Table 1	#	Semantic Rule	row of Table 4
1	$sa(S.A) = 2$	P0	13	$sa(mul_2.B) == sa(add_2.B)$	P4
2	$ia_2(mul_1) = 2$	P1	14	$ia(S.B) = sa(add_2.B)$	M0
3	$ia_2(mul_2) = 3$	P1	15	$ia(S.K) = sa(mul_2.K)$	M0
4	$ia_2(add_1) = 1$	P1	16	$ia(add_2.B) = ia(S.B)$	M1
5	$ia_2(add_2) = -3$	P1	17	$ia(mul_2.K) = ia(S.K)$	M1
6	$ia(add_1.A) = sa(S.A)$	P3	18	$ia(add_2.D) = sa(mul_1.D)$	M2
7	$sa(S.B) = sa(mul_2.B)$	P2	19	$ia(add_1.C) = sa(mul_1.C)$	M2
8	$sa(S.K) = sa(mul_2.K)$	P2	20	$ia(add_2.B) = sa(mul_2.B)$	M2
9	$ia(add_2.B) = ia(S.B)$	P3	21	$ia(mul_1.D) = sa(add_2.D)$	M2
10	$ia(mul_2.K) = ia(S.K)$	P3	22	$ia(mul_1.C) = sa(add_1.C)$	M2
11	$sa(mul_1.D) == sa(add_2.D)$	P4	23	$ia(mul_2.B) = sa(add_2.B)$	M2
12	$sa(mul_1.C) == sa(add_1.C)$	P4			

Table 5: The semantic rules of the equivalent AG for the example 3.4 (Equations system paradigm).

	CONDITION	**ACTION**
C0	t^e_{gi} is restricted by an ordinary constraint	if t^e_{gi} is a local variable **then tell** the constraint corresponding to the $sa_i(R^e_g)$ **else tell** the constraint corresponding to the $ia_i(R^e_g)$
C1	∀ ACTION of the transformation tables of the other paradigms do	**if** some synthesized attributes are restricted by a constraint **then tell** the constraint
C2	∀ ACTION of the transformation tables of the other paradigms do	**if** some inherited attributes are restricted by a constraint **then ask** the constraint

Table 6: Transformation Table for Constraint Logic Programming.

PAGE can model this programming paradigm introducing the transformation Table 6. The key idea is simple: we regard a CLP program be as an ordinary FLP program with extra constraints restricting the variables of the clauses. We perform the previous transformations of the tables used in the FLP paradigm in conjunction with those outlined in Table 6. Recall that Tell a constraint means that a new constraint is added (accumulated) in the constraint store. Ask for a constraint means waiting until the asked constraint is either entailed or disentailed from the constraint store (the information accumulated in the store so far). In this table Tell operations are performed each time a synthesized attribute is restricted by a constraint. On the other hand, Ask is performed each time an inherited attribute is restricted. As constraints we can consider the native constraints of the original program, as well as the semantic rules generated from the transformations defined in the transformation tables introduced above. The solver is informed each time an attribute occurrence is bound.

Ask and Tell actions are implemented as described in the computational model of PAGE. Constraints are collected in a central store in the Network Supervisor. Each constraint bares with it some information about the grammar rule (agent) from which it has been produced. Also some additional information is kept in the store dealing with techniques for constraint solving (local propagation, renaming, Gauss Elimination, etc.). It is not in the scope of this paper to fully describe the structure of the store and the solver. What is important is that PAGE welcomes overloaded `Tell`, `Ask` and `Solve` functions, allowing one to experiment with different kind of constraints [37].

In the current version of PAGE, built-in constraint operations are developed for solving linear equalities and inequalities constraints.

3.5.1. Multi-pass execution (CLP case)

The constraint store can improve the efficiency of the Multi-Pass schema previously described, since it can entail new passes (new sets of inherited attributes) not available before. Now the system can

#	Semantic Rule	row of Table 1,6
1	**tell** $sa(fib_0.X) = sa(fib_2.X)$	P2,C1
2	**tell** $sa(fib_0.F_1) = sa(fib_2.F_1)$	P2,C1
3	**tell** $sa(fib_0.F_2) = sa(fib_1.F_2)$	P2,C1
4	**ask** $ia(fib_1.X) = ia(fib_0.X)$	P3,C2
5	**ask** $ia(fib_2.X) = ia(fib_0.X)$	P3,C2
6	**tell** $sa(fib_1.X) == sa(fib_2.X)$	P4,C1

#	Semantic Rule	row of Table 2,6
7	**tell** $sa(fib_0.f_0) = f_0(sa(fib_0.F_1),sa(fib_0.F_2))$	F1,C1
8	**tell** $sa(fib_1.f_1) = f_1(ia(fib_1.X))$	F4,C1
9	**tell** $sa(fib_2.f_2) = f_2(ia(fib_2.X))$	F4,C1

#	Semantic Rule	row of Table 4,6
10	**ask** $ia(fib_0.X) = sa(fib_2.X)$	M0,C2
11	**ask** $ia(fib_1.X) = ia(fib_0.X)$	M1,C2
12	**ask** $ia(fib_2.X) = ia(fib_0.X)$	M1,C2
13	**ask** $ia(fib_2.X) = sa(fib_1.X)$	M2,C2

#	Semantic Rule	row of Table 6
14	**tell** $ia(fib_0.X) > 1$	C0

Table 7: The semantic rules of the equivalent AG for the Fibonacci series problem example 3.5 (CLP paradigm).

Ask for the entailment of new sets of inherited attributes which can fire a new pass. Notice that now if we try to run `fib(N,6765)` the system will answer the right value (i.e., `N=20`).

Example 3.5 The Fibonacci series problem written in Constraint Logic Language.

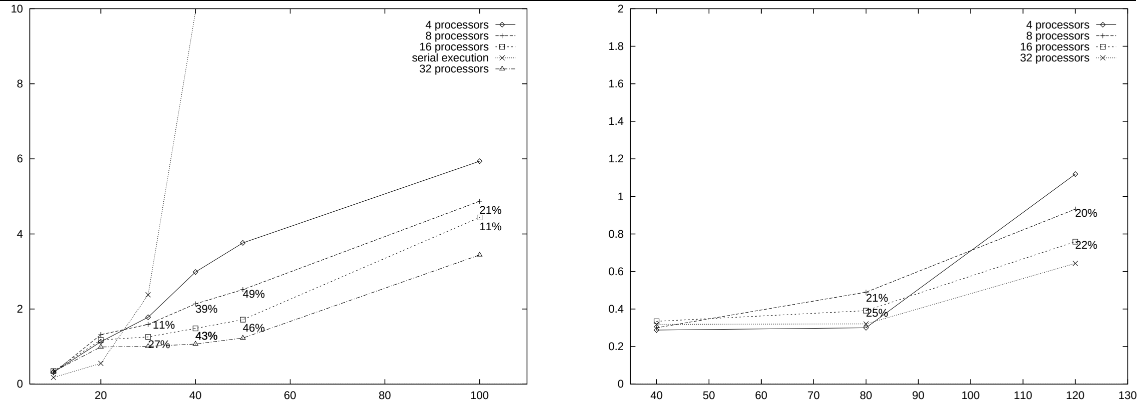
```
fib(X,F1+F2) :- X>1, fib(X-2,F2), fib(X-1,F1).

fib(0,1).
fib(1,1).
```

In Table 7 we see the program of example 3.5 transformed into an equivalent AG and enhanced with extra constraint operations. When the system asks the constraint in ACTION 10, the constraint store has among others the constraint told in ACTION 9. Both of them are sufficient to solve the system of the equations and take the new inherited attribute ($sa(fib_2.f_2)$ is considered as a bound attribute when predicate fib_2 gets successfully evaluated and so ACTION 9 infer that $c = f_2(ia(fib_2.X))$, where c is the newly bound value of $sa(fib_2.f_2)$. That means that the solver can infer that $ia(fib_2.X) = c+1$ (i). The constraint asked by ACTION 5 and the constraint (i) infer that $ia(fib_0.X) = c + 1$ which is the new inherited attribute). This is shown in Fig. 5 with the dashed lines. □

It is noteworthy that the same behaviour is possible if we supply the FLP tranformation table (Table 2) with extra transformation actions, simulating this way the constraint solver. However, that actions are problem depented and they do not fit in a declarative way of programming.

It is clear now, how multi-pass schema introduce a dynamic behavior of the dependences. Each cycle in the dependency graph represents a potential pass. It is left to the solver to dynamically choose what dependences are valid and what are not. On the other hand, if the constraint solver cannot entail a constraint, a successive pass may augment the store and make it sufficient for this. Thus, multi-pass schema can enchance the efficiency of the solver.

The modeling we propose allows simple constraints imposed in a guarded fashion. On the contrary, it does not support the full range of operators with agents proposed by Milner and Saraswat [26, 33]. Additionally, our transformations does not allow explicitly imposed control operations; ask and tells are performed implicitly in order to keep the programming style as close to logic programming style as possible. However, as we will see in the following, the method can seamlessly support a wider range of programming paradigms.



(a) (b)

Figure 7: Comparative results for (a) the Fibonacci(N) and (b) the NLP problem.

4. Modeling parallel programming using AGs

We saw that all the above capabilities are described with AG dependency graphs and, in turn, with a corresponding set Φ of semantic rules. As in compiler technology, we use AG as a specification language for the description of the programming paradigms under consideration. Each of the programming paradigms is outlined from a transformation table. These tables consist of transformation actions that have to be applied, under some conditions. More precisely, we have built a layer between the program executed and the control. According to Kowalski[23], Program = Logic + Control.
In this work we go one step further and we separate control into *control rationale* and *control principles*. Thus, now we have
Program = Logic + Control.
Control = Rationale + Principles.
While the principles component of the control is a set of mechanisms depending on the underlying implementation of the system (computational model), control rationale component is a set of specifications described via proper semantic rules and reflects the special operational semantics of the programming paradigm under consideration. Using AG technology to specify the control rationale is the semantic basis of PAGE. On the other hand AGs can still be used for the specification of the syntax of the language through their underlying CFG G. Consequently, we use AG paradigm to support the automation of the development of modern parallel declarative programming paradigms providing a powerful set of tools for the specification of the syntax of the language, for the modeling, and for the control of the execution of the program. AG formalism provides a uniform way for the specification of the declarative semantics of such programs and for the description of new operational concepts. Such a specification is given in [35], where the previously seen methodology is aplied in a more general class of programs belonging in the CCP family of languages [33].

5. Experimental Results

We have evaluated PAGE running LP and Natural Language Processing applications (NLP). In the case of LP, we have ran simple mathematical problems like Fibonacci(N) series or Factorial(N) and sorting problems like QuickSort or BubbleSort. In case of NLP we have used a grammar modeling the geography of the eastern Europe and processing queries about the natural features of some region. PAGE has been evaluated over ORCHID [39, 40] platform in an heterogeneous network of Ultra SPARC 140Mhz and 170Mhz, SPARC Station 40MHz, and Intel Pentium 133Mhz workstations, running Solaris. Tests have been performed on 4, 8, 16, and 32 workstations. Both LP and NLP applications have exhibited encouraging results as it is shown in Fig. 7.

In the graphs we can see the execution times in seconds with respect to the size of the input and for five different network configurations. Each line corresponds to a fixed number of processors, while X-axis corresponds to the size of the input and Y-axis to the execution time in seconds. To measure the execution time we have performed 20 tests per case. As final time we got the average value. Aside the execution time we show in parenthesis the percent improvement in execution time with respect to the previous case of fewer processors and for the same amount of input. For example, the execution in network of 8 processors and for input size 30 is 11% faster than the execution in network of 4 processors and for the same input according the results in Fig. 7(a).

For small input, the timings are relatively the same in either network, so we can not argue about any improvements. This seems to be a normal behavior since the small amount of input implies small amount of PAGE agents. This trivial number of agents does not exploit the computational power of the total network of workstations. Moreover, small amount of agents does not overload the network traffic for message passing. The larger the input the more significant the network traffic penalty gets. Additionally, each increase in the number of PAGE agents requires the participation of more processors in the computation for relieving the others from the large computational load. When we tried to run the programs in extreme cases, like calling a goal with incomplete arguments (e.g., $\leftarrow$ Fib(N, 6765)) or calling a goal with lots of unbound local variables, the efficiency of the system decreased almost uniformly by 12% in LP problems and by 10% in NLP problems. This is fair since in these cases the communication with the constraint solver and the generation of multiple pass execution agents slows the system down. In LP application results we also show the execution time we got for the serial execution of the Fibonacci series problem written in C, using the recursive algorithm (which is the closest to the declarative programming style). As one can see, the times are very disappointing for large size of input.

Observe that there is a “wave” of relatively stable percentage of improvement of execution times as we advance the computation to more workstations and more PAGE agents (larger input). It can be seen from the graph that the lines diverge as the size of input increases to start converging again as the size of input gets large enough. This means that there is a point in the computation where the system achieves its maximum speedup. This point depends on the number of active agents and the available processors in the network at any moment. Normally, it is infeasible to have a network of unrestricted amount of processors. Even in cases where indeterminate branches take place, always the danger of agent overflow exists. Thus, we have to resort to solutions for agent scheduling. Since our main concern in this work was to give a uniform method for programming in a plethora of parallel programming paradigms, this stage should be regarded as future work.

6. Conclusions

We have presented a method for modeling different control operations on different parallel implementations of declarative programming languages. To do so we use a program analysis methodology based on Attribute Grammars dependency graphs. We have built a *control rationale* layer that lies between program logic and *control principles* of the underlying platform architecture. Control rationale captures the programmable part of the control. Simply put, control rationale specifies the ways we can “control the control”. Attribute Grammar technology is used as the means for that specification. We have proposed some transformation tables, each one carrying the semantics of a different programming paradigm. Each of the paradigms follows the operational semantics induced by the dependency graphs corresponding to an AG. Operationally complete execution is granted by following the evaluation order prescribed by the dependency graphs. Externally linked functions can be embedded into PAGE allowing more complex semantic rules supporting FLP. Special purpose build-in constraint control primitives, such as blocking `ask` and atomic `tell` are used for supporting CCLP. However, the methodology could be proved a very valuable tool for supporting the automation of the implementation of more general classes of parallel programming paradigms.

For evaluating the proposed method we have implemented PAGE, a Parallel AG Evaluator extended to support a wide range of LP programming paradigms by exploring AND/OR parallelism. The tool has been implemented on the top of MPI and PVM prototype properly extented [38, 36] for supporting multi-threaded development (ORCHID extension). Actually the extended prototype forms the *control principles* layer and captures the properties and the restrictions of the underlying platform architecture. Although experimentally we got encouraging results in quite different applications[37, 36, 38] and intuitively the proposed methods seem to be sufficient enough to support a wider range of parallel programming paradigms, we have to go further and theoretically prove the sufficiency of the transformation methods. We are working on the specification of a special programming language for programming the control rationale. Actually this effort will lead us to the introduction of a meta-programming layer within which we can control the control of the declarative programming languages in a clean and sufficient way.

Bibliography

[1] S. Bonnier and J. Małuszyński. Towards a clean amalgamation of logic programs with external procedures. In *5th International conference and symposium on logic programming*, pages 311 – 326. MIT press, 1988.

[2] J. Boye. S-SLD resolution - an operational semantics for logic programs with external procedures. In *Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 383 – 393. Springer Verlag, 1991.

[3] J. Boye. Avoiding dynamic delays in functional logic programs. In *Programming Language Implementation and Logic Programming 93*, number 724 in LNCS. Springer-Verlag, 1993.

[4] R. J. Brachman and H. J. Levesque. The tractability of subsumption in frame-based description languages. In *Proc. of the National Conference Artificial Intelligence*, pages 34 – 37, 1984.

[5] A. Colmerauer. An introduction to prolog-III. *Communications of the ACM*, 33(7):69–90, July 1990.

[6] J. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publisher, 1987.

[7] P. Deransart and J. Małuszyński. Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2(2):119–156, 1985.

[8] P. Deransart and J. Małuszyński. *A Grammatical View of Logic Programming.* The MIT Press, 1993.

[9] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243 – 320. Elsevier, 1990.

[10] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving large combinatorial problems in logic programs. *Journal of Logic Programming*, 8(1-2):74–94, 1990.

[11] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proc. of the International Conference on Fifth Generation Computer Systems FGCS-88*, Tokyo, Japan, December 1988.

[12] I. Fang. *FOLDS, a Declarative Formal Language Definition System.* PhD thesis, Computer Science Dept., Stanford University, Dec. 1972.

[13] M. Hanus. The integration of functions into logic programming: from theory to practice. *The Journal of Logic Programming*, 19(20):583 – 628, 1994.

[14] M. Hanus. A unified computation model for functional and logic programming. In *24th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 80 – 93, Paris, January 1997. ACM.

[15] S. Haridi and P. Brand. ANDORRA prolog: An integration of prolog and committed choice languages. In *Proc. of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 745 – 754, Tokyo, Japan, November 1988.

[16] P. J. Hayes. In defense of logic. In *International Joint Conference on Artificial Intelligence*, pages 287 – 296, 1977.

[17] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL '87 14th ACM annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 111–119, Munich, January 1987. ACM.

[18] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[19] M. Jourdan. A survey of parallel attribute grammar methods. In A. Alblas and M. B., editors, *Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 234 – 255. Springer - Verlag, New York, 1991.

[20] G. E. Kaplan and S. M. Kaiser. Incremental attribute evaluation in distributed language-based environments. In *Proc. of the 5th ACM Symposium on Principles of Distributed Computing*, pages 121–130, Aug. 1986.

[21] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127 –145, 1968.

[22] D. E. Knuth. Semantics of context-free languages, correction. *Mathematical Systems Theory*, 5(1):95 – 96, 1968.

[23] R. Kowalski. *Logic for Problem Solving.* Elsevier Science, New York, 1979.

[24] H. J. Levesque and R. J. Brachman. A fundamental tradeoff in knowledge representation and reasoning. In *Readings in Knowledge Representation*, pages 41–70. Morgan Kaufmann Publishers, 1985.

[25] J. Lloyd. *Foundations of Logic Programming.* Spinger Verlag, second extended edition, 1987.

[26] R. Milner. *A Calculus of communicating systems.* Number 92 in LNCS. Springer-Verlag, 1980.

[27] J. Paakki. Multi-pass execution of functional logic programs. In *21th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 361 – 374, Portland, Oregon, January 17 - 21 1994. ACM.

[28] J. Paakki. Attribute grammar paradigms – a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196 – 255, June 1995.

[29] G. Panagiotopoulos, G. Papakonstantinou, and G. Stamatopoulos. Attribute grammars and logic programming. *Angewandte Informatik*, (5), 1988.

[30] G. Papakonstantinou and J. Kontos. Knowledge representation with attribute grammars. *The Computer Journal*, 29(3):241–245, 1986.

[31] G. Papakonstantinou, C. Moraitis, and T. Panayiotopoulos. An attribute grammar interpreter as a knowledge engineering tool. *Angewandte Informatik*, (9):382–388, 1986.

[32] G. Papakonstantinou, C. Voliotis, and N. Sgouros. Dependency-directed binding of variables for constraint logic programming. In *Proceedings of the DEXA 94 Conference*, Athens, Greece, 1994.

[33] V. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg Pa 15213, January 20 1989.

[34] V. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[35] A. Thanos and G. Papakonstantinou. An attribute grammar modeling for generating CCLP languages. In *4th International Conference on Principles and Practices of Constraint Programming CP98, Workshop on Modeling and Computing with Concurrent Constraint Programming*, Pisa, Italy, 30 Oct. 1998.

[36] A. Thanos, G. Papakonstantinou, and P. Tsanakas. Towards an automatic generator of multi-paradigm programming languages. In *Parallel and Distributed Computing and Networks (PDCN 98)*, Brisbane, Australia, 14-16 Dec. 1998.

[37] A. Thanos, G. Papakonstantinou, and P. Tsanakas. Attribute grammar based system for multi-paradigm distributed computing: The concurrent constraint logic programming paradigm. In *10th IITT Expersys-98 Conference on Artificial Intelligence Applications*, Virginia, U.S.A., 16-17 Nov. 1998.

[38] A. Thanos, C. Voliotis, and G. Papakonstantinou. Modeling the control on parallel implementations of declarative programming languages. In *International Conference on Computational Intelligence for Modeling,Control, and Automation (CIMCA99)*, Vienna, Austria, 17–19 Feb. 1999.

[39] C. Voliotis, G. Manis, H. Lekatsas, P. Tsanakas, and G. Papakonstantinou. Orchid: A portable platform for parallel programming. *Euromicro Journal of Systems Architecture*, 43(6-7):459–478, April 1997.

[40] C. Voliotis, G. Manis, A. Thanos, G. Papakonstantinou, and P. Tsanakas. Facilitating the development of portable parallel applications on distributed memory systems. In *Proc. Massively Parallel Programming Models MPPM-95 conference*, Berlin, 1995. IEEE Computer Society Press.

[41] C. Voliotis, N. M. Sgouros, and G. Papakonstantinou. Attribute grammar based modeling for concurrent constraint logic programming. *International Journal on Artificial Intelligence Tools*, 4(3):383 – 411, 1995.

[42] C. Voliotis, A. Thanos, N. Sgouros, and G. Papakonstantinou. Daffodil: A framework for integrating AND/OR parallelism. In *5th Hellenic Conference on Informatics*, Athens, Dec 1995.