An Attribute Grammar Modelling for Generating CCLP Languages

A. M. Thanos and G. 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
http://www.cslab.ece.ntua.gr

Abstract. The paper presents how we can model Concurrent Constraint Logic Programming (CCLP) using the formalism of Attribute Grammars(AGs). For this, we present PAGE, a general Parallel AG Evaluator. The proposed method is based on the equivalence of Attribute Grammars with Logic Programming (LP). The principles of this relationship have been extended to support a variety of programming paradigms (AND/OR parallelism, Concurrent Constraint Logic Programming) in a distributed computing framework. Each program is transformed into an equivalent Attribute Grammar. PAGE evaluates the grammar according to the operational semantics described by the dependency graph induced by the corresponding semantic rules. The system has been implemented and tested in a wide range of architectures, exhibiting encouraging results.

1 Introduction

Parallel architectures represent the most promising means to increase the computing power of computer systems in the future. Yet, programming for multiprocessors and distributed systems is a laborious task. Imperative languages lack an easy and straightforward way to deal with issues arising when we are programming in parallel, such as synchronization, variable dependency, binding delay, etc. "Thinking parallel" adds up significantly to the application's development costs. Moreover, there is a tight and error-prone dependency between logic and control of such programs, making them hard to be maintained, modified, or ported to different architectures. Programs written in such languages fail to keep their semantics the same, during the software lifetime.

On the contrary, declarative languages offer an interesting alternative to the problem of automatic parallelization. They are very high-level languages and enable programs to be developed with less time and effort than with imperative languages. Applications written in such languages exhibit a clear independence between logic and control [22]. In essence, they are implicitly parallel languages and they are amenable to parallel implementations, since they can be computed sequentially or in parallel without changing their declarative semantics. The programmer does not have to worry if the program runs on a sequential or a parallel architecture; in either case the programming style remains the same.

However, declarative languages alone are not a panacea. Pure declarative programming is sometimes inefficient or totally incapable to capture the whole range of a programmer's needs. Moreover, despite the interesting progress in hardware manufacture, declarative languages remain less efficient than imperative languages, even under parallel implementations. Combinatorial explosion is the main drawback in the execution of such programs. A lot of research effort has been dedicated to methods for controlling the unbearable growth of search space. The basic idea behind these methods is to introduce a form of *indeterminacy*, that is, to relax strict adherence to the non-deterministic control mechanism and to *commit* to choices made under certain conditions (committed-choice determinism or indeterminism). Control primitives have to be devised for systematical recognition of situations where one choice is selected in favor of another. Such conditions may form a repository of assertions about a domain of discourse.

Functional approaches to Knowledge Representation (KR) [23, 4] have led to the introduction of control primitives in the form of computation in some domains other than Herbrand space. 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) [12, 13] and Constraint Logic Programming (CLP) [16] 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($\mathcal{R}$) [17], CHIP [9, 10], PrologIII [5], AKL [14]. An early approach towards CLP has been presented in [29], 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 [27, 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 [28, 26, 24, 25, 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 [18] .

Although a lot of research work has been devoted to establishing the connection between LP and AG, or FLP and AG, no work, to the best of our knowledge, deals with the connection of CLP with AG. This paper describes a method for unifying CCLP paradigms with AGs. The method is based on PAGE [35], a general tool for parallel attribute grammar evaluation.

2 Semantic Foundations

2.1 Attribute Grammars

Attribute Grammars have been proposed by Knuth [20, 21] 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 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 $X \to \alpha$, 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 An *Attribute Grammar* conists 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 $p \in P$, $p : X_0 \to X_1 \cdots X_n$ (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) 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 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 $f : \mathcal{I}(X_{p,k_1}.a_1) \times \ldots \times \mathcal{I}(X_{p,k_m}.a_m) \to \mathcal{I}(X_{p,k}.a)$.
  - in case of relation, c has the form
    $c : c(X_{p,k_1}.a_1, \ldots, X_{p,k_m}.a_m)$
    where $c : \mathcal{I}(X_{p,k_1}.a_1) \times \ldots \times \mathcal{I}(X_{p,k_m}.a_m) \to \{true, false\}$.
- $\mathcal{I}$ is an *interpretation* of $\mathcal{L}$ in some S-sorted algebraic structure $\mathcal{A}$.

□

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 [20], are the $S-attributed$ grammars in which only synthesized attributes are allowed.

Due to the restrictive form ofi the S-attributed grammars, $L-ordered$ grammars are used in practice.

Definition 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 $\bigcup_{1\leq k<j} I(X_{p,k}) \cup S(X_{p,0})$ for $j = 1, \ldots, np$. □

2.2 AGs and Logic Programming

Operational semantics of Logic Programming can be modelled 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.

- 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 the terms in which the output attribute occurs and some input attribute instances of the syntactic rule

Fig. 1. AG Construction Rules for LP Equivalence

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-ordered 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 and Concurrent Constraint Logic Programming. 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 [33]. The idea of expressing the control information by its own, carefully designed language is not new but is prevalent for a long time [15]. 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".

3 Modelling Parallel LP 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 proper message interchange. Many different approaches have been proposed for parallel AG evaluation [18], 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 [35], a Parallel AG Evaluator.

3.2 PAGE

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.*

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 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 [11] and Kaplan and Kaiser [19]. 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 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 unify at the same time [36].

3.3 Modelling 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 [26]. In our approach we simulate parallel LP with parallel AG evaluation using PAGE.

(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** $sa_i(R^e_g) = sa_i(R^e_q)$
P3	t^e_{gi} is a variable and t^e_{qj} is another textual occurrence of the same variable	$ia_i(R^e_g) = ia_j(R^e_q)$ such that $q == 0$ and $g \neq 0$
P4	t^e_{gi} is a variable and t^e_{qj} is the nearest textual occurrence of the same	$sa_i(R^e_g) == sa_j(R^e_q)$ variable to the left $(q < g)$ such that $q \neq 0$ and $g \neq 0$
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** $sa_i(R^e_g) = sa_i(R^e_q)$
P3	t^e_{gi} is a variable and t^e_{qj} is the nearest textual occurrence of the same	$ia_i(R^e_g) = ia_j(R^e_q)$
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 $q == 0$ 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 modelling.

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. Modelling LP over PAGE introduces the use of transformation tables according to the kind of parallelism we want to achieve. 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.[1]

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

[1] 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).)

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 31 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-ordered AG and consequently follows the rules for operational completeness described in [8]. Thus the transformations lead to a sound and operationally complete grammar.

Example 31 The Fibonacci series problem written in Prolog-like notation.

```
fib(X,F) :-  is(X2,X-2), fib(X2,F2),
         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. 2. In this figure inherited attributes are located on the left of the corresponding predicate while synthesized attributes are located on the right. 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 section (constraints).

Observing the dependency graph in Fig. 2, one can clearly see that the basic primitive 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).

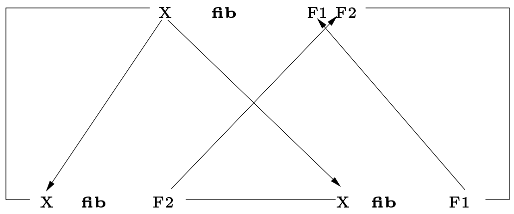

Fig. 2. Dependency graph for AG produced by Table 1.

4 Modelling CCLP using AGs

The capability of AGs to deal with non-term interpretations led us to model LP using constraints. The flexibility of controlling operational semantics in parallel LP can also be shown in the paradigm of CCLP. In this context we can apply the previously seen methodology in a more general class of programs belonging in the CCP family of languages. For our purposes we will be based on the Algebraic Syntax of the `cc` family of languages as described in [30]. Here we adopt a subset of the syntax. The key characteristic of these languages is the introduction of the Tell-and-Ask framework. Under this framework the computation is controlled from two basic actions on constraints, namely `Tell` and blocking `Ask`. These basic actions are in close cooperation with the commitment operations, namely `don't know`($\Rightarrow$), `sequential don't know`($\Longrightarrow$), `don't care`($\rightarrow$), and `determinate don't care`($\hookrightarrow$). With the expressive power of the AG framework we can model the basic actions of the CCP programs in an easy and straightforward way.

In Fig. 3 we show the underlying CFG grammar of the AG describing the algebraic syntax of the `cc` family. We perform the constructions rules described in Tables 1(a,b) (whichever we want) and we enhance our Φ component of the constructed AG with some extra semantic rules. We show the transformations by the following example.

Example 41 We present now a typical example of a program of CCP family. The program can be transformed into an AG with special semantic rules. Thus, based on the syntax of Fig 3 we consider the following attribute grammar:

- **Nonterminals:** $N = \{Environment, Agent, goal, Ask\_tell\_operator, Basic\_operation, Commitment\_operator, Disjunction\}$
- **Terminals:** $T = \{stop, fail, \downarrow, \Box, ||, :, \rightarrow, \Rightarrow, \Longrightarrow, \hookrightarrow\}$
- **Start Symbol:** $D = Environment$
- **Sorts:**
  $S = \{B\_action_{consistency}, B\_ask\_tell_{operator}, Commitment_{operator}, Disjunction, \mathcal{C}, \mathcal{N}\}$
  where
  - $B\_action_{consistency} = \{True, False\}$
  - $B\_ask\_tell_{operator} = \{Ask, Tell\}$
  - $Commitment_{operator} = \{DontKnow, SeqDontKnow, DontCare, DetDontCare\}$

```
1. Disjunction → Basic_action Commitment_operator Agent |
2.               Disjunction '□' Disjunction

3. Commitment_operator → '→' |
4.                       '⇒' |
5.                       '⟹' |
6.                       '↪'

7. Basic_action → Constraint Ask_tell_operator |
8.                Basic_action ':' Basic_action |
9.                Basic_action '□' Basic_action

10. Ask_tell_operator → '↓' (i.e.,Ask) |
11.                     ε (i.e.,Tell)

12. Agent → 'stop' |
13.         'fail' |
14.         goal |
15.         Disjunction |
16.         Agent ‖ Agent |
17.         Environment.Agent

18. Environment → goal '::' Agent |
19.               Environment'.'Environment
```

Fig. 3. Attribute Grammar Syntax for CCP Languages

- $Termination = \{Stop, Fail\}$
- $\mathcal{C}$ the sort of constraints
- $\mathcal{N}$ is the set of Natural numbers

$\Phi = \{Ask\_Tell, Tell, Ask, f_1, f_2, f_3, spawn\}$ where

- $Ask\_Tell : B\_ask\_tell_{operator} \times \mathcal{C} \rightarrow B\_action_{consistency}$
  $Ask\_Tell(op_1, op_2) =$ **if** $op_1 == Ask$ **then** $Ask(op_2)$
  else $Tell(op_2)$
- $f_1 : B\_action_{consistency} \times B\_action_{consistency} \rightarrow B\_action_{consistency}$
  $f_1(op_1, op_2) =$ **if** $op_1 == True$ **then**
  if $op_2 == True$ **then** $True$
  else $False$
  else $False$
- $f_2 : B\_action_{consistency} \times B\_action_{consistency} \rightarrow B\_action_{consistency}$
  $f_2(op_1, op_2) = op_1$ **or** op_2
- $f_3 : B\_action_{consistency} \times Commitment_{operator} \times \mathcal{N} \rightarrow Termination$

  $f_3(op_1, op_2, op_3) =$ **if** op_1 **then** $spawn(op_2, op_3)$
  else $fail$

– **Attributes:** $A = Syn \bigcup Inh$ so that
$Syn = \{Consistent, AT\_operator\} \bigcup S$

$Inh = I$
$Syn(Basic\_action) = Consistent$
$Syn(Ask\_tell\_operator) = AT\_operator$
$Syn(Commitment\_operator) = C\_operator$
$Syn(Agent) = id$

- **Productions:**
  the productions are those in fig. 3

- **Semantic Rules:**
  R(11): `Ask_tell_operator`.$AT\_operator{=}Tell$
  R(10): `Ask_tell_operator`.$AT\_operator{=}Ask$

  R(7) : `Basic_action`.$Consistent =$
  $Ask\_Tell($`Ask_tell_operator`.$AT\_operator$, `Constraint`$)$

  R(8) : $\texttt{Basic\_action}_0.Consistent =$
  $f_1(\texttt{Basic\_action}_1.Consistent, \texttt{Basic\_action}_2.Consistent)$

  R(9) : $\texttt{Basic\_action}_0.Consistent =$
  $f_2(\texttt{Basic\_action}_1.Consistent, \texttt{Basic\_action}_2.Consistent)$

  R(3) : `Commitment_operator`.$C\_operator{=}DontCare$
  R(4) : `Commitment_operator`.$C\_operator{=}DontKnow$
  R(5) : `Commitment_operator`.$C\_operator{=}SeqDontKnow$
  R(6) : `Commitment_operator`.$C\_operator{=}DetDontCare$

  R(1) : Apply the semantic rules from Table 1 plus
  `Disjunction`.$Terimnation{=}$
  $f_3($`Basic_action`.Consistent, `Commitment_operator`.C_operator, `Agent`.id$)$

  R(16) : Apply the semantic rules from Table 1

□

In the above example, I and S are the sets of inherited and synthesized attributes as are described in section 2.2. Syntactic rule 17 is referred to programs supporting block structure which are not in the scope of this paper. Syntactic rule 18,19 constitute the program representation. For each of the goals of this representation we perform the above construction rules, as described in the example. Syntactic rule 15 corresponds to OR parallelism (is a procedure), while Syntactic rule 16 corresponds to AND parallelism. Functions Ask and $Tell$ are build-in functions performing the obvious computation. Similarly, function *spawn* decides whether Agents have to be spawned to the network all together (don't know commitment), one by one (sequential care commitment), or just only one (don't care commitment).

5 Results and Related Issues

In PAGE we can use externally linked procedures or functions as semantic rules . There are three special purpose functions that are build-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 it 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 special purpose constraint solvers.

PAGE has been designed over ORCHID [34], a portable layer for distributed programming which supports the well-known distributed programming interfaces PVM and MPI, and extends them for multi-threaded programming. The system has been implemented and tested in a wide range of architectures. 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.

We have tested PAGE with a small class of CCLP programs, mainly with arithmetic problems (e.g, Fibonacci Series, Factorial, MinMax) and over the domain of simple synchronization problems (e.g., Bus Connection, Chain Connection etc.), where we got encouraging results [32, 31]. Our language does not capture all the expressive power of the aforementioned algebraic syntax of `cc` languages. It is formed merely from clauses with simple constraints of linear equations and inequalities. Moreover, it does not support *spawn* function yet; it follows the don't know commitment. The implementation of the *spawn* function, as well as the embellishment of our language with more CCLP features should be regarded as future work. Nevertheless, we believe that with the proposed modelling this would be not a hard task.

AG modelling can be proved very helpful in cases we are dealing with functional arguments or external functions. In either case not enough instantiated function calls impose delays on the computation. A distributed Multi-Pass Technique has been proposed [35], and has been implemented in [31] for obviating this problem.

References

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 Laguange 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 Implemantation 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. 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.
10. 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.
11. I. Fang. *FOLDS, a Declarative Formal Language Definition System.* PhD thesis, Computer Science Dept., Stanford University, Dec. 1972.
12. M. Hanus. The integration of functions into logic programming: from theory to practice. *The Journal of Logic Programming*, 19(20):583 – 628, 1994.
13. 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.
14. S. Haridi and P. Brand. ANDORRA prolog: An integration of prolog and commited choice languages. In *Proc. of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 745 – 754, Tokyo, Japan, November 1988.
15. P. J. Hayes. In defense of logic. In *International Joint Conference on Artificial Intelligence*, pages 287 – 296, 1977.
16. 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.
17. 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.
18. 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.
19. 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.
20. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127 –145, 1968.
21. D. E. Knuth. Semantics of context-free languages, correction. *Mathematical Systems Theory*, 5(1):95 – 96, 1968.
22. R. Kowalski. *Logic for Problem Solving.* Elsevier Science, New York, 1979.

23. 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.
24. 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.
25. J. Paakki. Attribute grammar paradigms – a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196 – 255, June 1995.
26. G. Panagiotopoulos, G. Papakonstantinou, and G. Stamatopoulos. Attribute grammars and logic programming. *Angewandte Informatik*, (5), 1988.
27. G. Papakonstantinou and J. Kontos. Knowledge representation with attribute grammars. *The Computer Journal*, 29(3):241–245, 1986.
28. G. Papakonstantinou, C. Moraitis, and T. Panayiotopoulos. An attribute grammar interpreter as a knowledge engineering tool. *Angewandte Informatik*, (9):382–388, 1986.
29. G. Papakonstantinou, C. Voliotis, and N. Sgouros. Dependency-directed binding of variables for constraint logic programming. In *Proceedings of the DEXA 94 Confrence*, Athens, Greece, 1994.
30. V. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg Pa 15213, January 20 1989.
31. 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)*, page To appear, Brisbane, Australia, 14-16 Dec. 1998.
32. 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*, page To appear, Virginia, U.S.A., 16-17 Nov. 1998.
33. A. Thanos, C. Voliotis, and G. Papakonstantinou. Modelling the control on parallel implementations of declarative programming languages. In *International Conference on Computational Intelligence for Modelling,Control, and Automation (CIMCA99)*, page To appear, Vienna, Austria, 17–19 Feb. 1999.
34. 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.
35. 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.
36. 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.