INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-510-642-4274 • FAX 1-510-643-7684

Process Grammar Processor: An Architecture for a Parallel Parser

Massimo Marino*

TR-92-054

August 1992

Abstract

A parallel architecture of a parser for Natural Language is described. A serial architecture has been already realized [6] and is currently used in the Process Grammar Development Environment (PGDE) [8], a system for the construction and testing of Natural Language grammars and the generation of the corresponding parsers. The PGDE is built around the Process Grammar Processor (PGP) running a model of grammar suited for the generation of Natural Language applications. The grammar model, named Process Grammar (PG), is an extension of an augmented context-free phrase-structure grammar, and the parser is designed to use such a grammar model. A PG is a set of rules that are treated by the processor as static descriptors of dynamic processes that are scheduled and applied if the conditions for their execution hold: from this the name Process Grammar. In this report the PG model is extended in order to allow a more structured and modular construction of grammars, even of big dimensions, keeping separated parsing control, and syntactic and semantic specifications, partitioning a PG in clusters of rules, completely independent one from each other, carrying on their own dedicated recognition of specific parts of speech. The main steps of the serial PGP are realized in the parallel architecture as parallel processes that communicate between them the results of their computations using a message passing protocol. This allows the realization of some interesting parsing strategies and the implementation of parsing mechanisms extending the recognition

*International Computer Science Institute, 1947 Center Street, Suite 600, Berkeley CA 94704-1105, and AITech s.n.c., Vicolo S. Cosimo 2, 56125 Pisa, Italy. E-Mail: massimom@icnucevm.cnuce.cnr.it

capacity of the parser, e.g., it is possible to recognize context-sensitive grammars by means of a context-free device, and it is possible to perform more complex recognition steps that could not be possible in a serial and standard context-free parsing environment. Both serial and parallel versions of the parser are introduced and described, looking in greater detail the mechanisms of process scheduling, and how they can be used and extended for implementing various cases of parsing strategies.

1 Introduction

The basic main issues addressed in this report for the architecture of the Process Grammar Processor (PGP) and for the Process Grammar (PG) model are those of efficiency, control of parsing process, modularity of the involved PGP tasks, and tools for the grammar construction. Some of these issues would not arise when small grammars are used; on the contrary, they become critical whenever grammars of big dimensions are implemented, and when they produce many interpretations for the same input. In both the serial and parallel architectures the core parsing engine is a pure bottom-up parser that has been extended in two ways: introducing new states of computation in the processor itself, and introducing new mechanisms for parsing control that can be used in the rules of the PG. Both the extensions allow the implementation of parsing strategies and mechanisms useful in facing some phenomena occurring in Natural Languages, whose treatment can be realized by adding some control information to a set of rules without affecting the syntactic or semantic information that is kept separated.
Following in part [3], it is possible to classify parsing algorithms in three main classes, depending on the kind of the algorithm:

- **Serial-Serial** :

  A serial algorithm that carries on one analysis at a time (with backtracking);

- **Serial-Parallel** :

  A serial algorithm that carries on all possible analyses altogether (no backtracking); parallelism is not real, but refers to the set of the analyses considered;

- **Parallel-Parallel** :

  A parallel algorithm that performs all analyses in parallel (no backtracking); parallelism is real and for every analysis the parser behaves like a serial-serial parser without backtracking.

The PGP falls in the serial-parallel category; the parallel extended architecture introduced in the next Sections falls in the parallel-parallel category. In order to introduce clearly the PGP architecture and the associated mechanisms, it is better to introduce in a general way the serial PGP first, and afterwards the parallel PGP.

1.1 Serial-Parallel PGP

The PGP architecture as described in [6],[8] can be defined to be a *serial-parallel* parsing architecture since the algorithm is serial, and there is only one structure that evolves describing the state of the PGP, called *State of Computation* (SoC); the parsing structure built by the parser, the Parse Graph Structure (PGS), contains all possible analyses and they are constructed in parallel, that is, the PGP tries all possible analyses at the same time; they are then considered one at a time, storing in a queue the awaiting ones. The PG is a set of grammar rules each one defining a reduction rule of the form $(A \leftarrow c_1 \ldots c_n), n \geq 1$, and possibly a set of actions to be applied when some conditions occur; they are static representations of potential processes that are scheduled by the PGP depending on the result

of finding a process environment, where for process environment it is meant a set of nodes in the PGS matching the right-hand side of the reduction rule.
An important matter in parsing is how a set of rules is fired, that is, which scheduling mechanism must be used for firing rules when a current context where the rules can apply is provided. Once a scheduling strategy is stated it is also defined the scheduling mechanism adopted by the parser, and this deals with the issues of efficiency and control in a strong way. Some literature exists about this topic, mainly investigating different strategies in chart parsing ([4],[9],[10],[11]). The PGP is not a chart parser, even if the PGS, as described in Section 3, looks like a chart since it stores almost the same kind of information, but the difference between the two data structures is that they are used differently. In fact, the common information in the PGS is just used for the purpose of searching process environments for the rule at hand. The main structures stored in the PGS are parse trees, and they are the real structure a PG works on; all the other information is a support for carrying out an efficient search in the subtrees.
The task of the scheduling mechanism of the PGP is performed in two steps: first a scheduler gets access to the set of the possible triggerable rules by means of the scheduling strategy, building a process descriptor for every rule found; afterwards, the process descriptor, that must contain all the necessary information for firing the process, is passed to the matcher that must find a set of process environments; for every process environment found a process is fired by replicating the process descriptors. The strategy adopted in the PGP is the right-corner (RC) strategy which is based on the detection of the occurrence of two main events because a set of PG rules could be fired: terminal node scanning, and non-terminal node construction when a reduction has been performed. Therefore, the scheduler has to access all the rules having the category of the node involved in one of the two previous events as the right-most one in the right-hand side of the reduction rule. These rules are only potentially applicable; the next step, the matching phase, must determine if a context in the PGS exists for each one of them.
The RC strategy is opposed to the left-corner (LC) strategy, and using either one or the other implies different consequences that also depend on the type of parsing carried on. Furthermore, considering the grammar rules as processes to be scheduled by the parser-processor, it is needed that they must be scheduled when this is possible: this means that LC is not suitable since it implies that not all the constituents of a right-hand side of a reduction rule are already available when the rule is accessed by the scheduler; therefore, a sort of caching of the pending processes would be required; this is what happens in chart parsing when active edges are added to the chart. On the contrary, the use of RC allows a prompt scheduling for those rules that, when sent to the matcher, are really fired if and only if all the constituents needed for setting up the process environment are already present.
The PG model used in this architecture has been extended in the parallel one and it will be described in a formal way in Section 4, so an informal description of the model used in the serial-parallel PGP is given here.

1.1.1 Process Grammar for Serial-Parallel PGP

A PG is a phrase-structure grammar composed by rules augmented by a set of operations, the *augmentations*, which are applied by the PGP depending on the context. The structure of a PG rule is defined as follows:

(**defrule**

:RNAME	*rname*	; The rule name
:ST	*status*	; *status* ::= active | inactive
:RED	*reduction-rule*	; *reduction-rule* ::= $(A\ (c_1 c_2 \ldots c_n)) \mid (empty\ (c_1 c_2 \ldots c_n))$
:TEST	*tests*	; The tests
:RATF	*ratf*	; Recovery actions on tests failure
:ACT	*actions*	; Actions on tests success
:RAMF	*ramf*)	; Recovery actions on match failure

Henceforth, if $FIELD$ is a field name of a PG rule r, then $FIELD(r)$ will denote the content of that field. The following code gives an idea of how the different augmentations are applied by the PGP:

```
if match(RED(r))
     then if TEST(r)
               then ACT(r)
               else RATF(r)
     else RAMF(r);
```

The **defrule** format is a static description of potential processes that are scheduled by the PGP depending on the result of finding a process environment. By means of some parsing mechanisms the process environments can also be built dynamically and passed through different rules. These mechanisms are introduced and described as soon as the following discussions are concerned. The state of a rule is a constraint for the schedulability of a process since its applicability is also determined according to it. Two kinds of state are distinguished:

- the *original state*, declared in the rule definition; a rule is *originally active* when it is active, otherwise, it is *originally inactive* when inactive;
- the *real state*, assumed by the rule at parsing time; at the beginning the real state coincides with the original state; during the parsing it can be changed by calling some specific functions that do this task: a rule is enabled if it is set really active, otherwise it is disabled.

Really inactive rules cannot be scheduled even if it is possible to apply them in a process environment, otherwise they are always scheduled. Really inactive rules can play a central role in the implementation of parsing mechanisms and different firing strategies. The reduction rule provides the pattern to be matched; two different kinds of reduction rules are possible: context-free rule, and empty rule. Empty rules are used in implementing special

mechanisms, such as the definition of context-sensitive rules, which is the easiest kind to implement. In general, an empty rule simply provides a pattern to be matched to get a process environment, and depending either on the existence or not of the process environement, or on the results of some tests performed in the process environment found, some further actions can be performed, e.g., some rules are fired following a specific scheme or strategy, or some rules are enabled/disabled.
As already mentioned, it is possible to implement various parsing strategies and different mechanisms for the same kind of phenomena; some examples will be given later after the description of the parallel PGP, but since one of these possible mechanisms, the implementation of context-sensitive rules, has been mentioned, here is an example about it:

Example 1.1 *Context-sensitive languages can be generated by rules of the following kind*[1]*:* $\alpha X \beta \leftarrow \alpha\gamma\beta$*. Using context-free rules it is necessary to split the application of such a rule in two steps: search for a process environment matching* $\alpha\gamma\beta$*; apply a rule performing the reduction* $X \leftarrow \gamma$ *over the subset of the process environment for* γ*. This is an easy task as two rules are needed: the first one, say* $Re1$*, is an empty rule with reduction* $(empty \leftarrow \alpha\gamma\beta)$*, and, through the call to a special function, the second rule, say* Rx*, is scheduled with the proper process environment matching its reduction rule* $(X \leftarrow \gamma)$*. It is important that* Rx *be really inactive since it must be fired by* $Re1$ *only. The scheduling of* Rx *does not depend on a new matching process, thus its application must immediately follow that of* $Re1$*. This reveals that there are some priorities among different kinds of rules, as it will be explained later. In this case* Rx *is a higher priority rule and all other rules already scheduled must wait longer because the application chain* $Re1$–Rx *cannot be broken since it represents a unique task. This could be represented in the following notation:*

$$(empty \leftarrow \alpha\gamma\beta)_{Re1}\{[\overset{\mathrm{RA}}{\Rightarrow} (X \leftarrow \gamma)_{Rx}]\}_{ACT(Re1)}$$

where RA is the function that does the job of scheduling Rx*. The above task written in* **defrule** *format is worked out as follows:*

(**defrule**
  :RNAME Re1
  :RED (empty (a b c d e f)) ; $\epsilon \leftarrow abcdef$
  :ACT ((RA Rx)))

(**defrule**
  :RNAME Rx
  :ST inactive
  :RED (X (c d e))) ; $abXf \leftarrow abcdef$

In general, this mechanism can be extended to a k *context-sensitive reductions set:*

$$(empty \leftarrow \alpha_1\gamma_1\beta_1 \ldots \alpha_k\gamma_k\beta_k)_{Re1}\{[\overset{\mathrm{RA}}{\Rightarrow} (X_1 \leftarrow \gamma_1)_{RX1}] \ldots [(X_k \leftarrow \gamma_k)_{RXk}]\}_{ACT(Re1)}$$

where it is possible to perform the k *reductions in parallel.*

[1] Following an almost standard notation throughout the text, the greek letters $\alpha, \beta, \ldots$ denote strings of terminal and non-terminal symbols, unless stated otherwise; the lower-case letters $a, b, \ldots$ denote terminal symbols, and upper-case letters $A, B, \ldots$ non-terminal symbols.

1.1.2 Operations in Serial-Parallel PGP

The serial-parallel PGP has basically three main modules: the scheduler, the matcher, and the proper parser. As already described, the scheduler choices the rules on the base of the RC firing strategy inserting in the appropriate stack a process descriptor through which the process to be fired is identified, along with other information useful to the matcher. Depending on the kind of rule fired and on the kind of operation performed by the parser (that usually depends on the rule or on how the rule has been fired) the process descriptors are put on different stacks; there are five stacks, but only two are used by the core PGP as the others are involved in the parsing process when special kinds of rules are fired.
The main component of the running state of the PGP described by the SoC is the *operation* Op the PGP is executing: such an operation represents a state in which the parser is, and transitions from an operation to another are determined by the state of the appropriate stacks. The five stacks form together a priority queue; their relative position in the queue indicates that the rules have different execution priorities. The priority queue with the names of the stacks listed in is represented as follows:

$[S02.S03.S1.S2.S3]$

and Figure 1 shows all possible transitions between the operations of the PGP, and which conditions on the stacks must hold for making transitions; operations that involve the scheduler are labeled with an 'S', whereas operations that involve the matcher are labeled with an 'M'. The scheduler works exclusively on the stacks S2, and S3: the stack S3 is for the lowest priority rules having a non-empty reduction rule, also named standard rules, and it is accessed by the operation **reduce**; instead the stack S2, accessed by the operation **e-reduce**, is for the empty rules, and these rules have a higher priority than those in stack S3. The stack S1 is reserved for rules scheduled by the function RA, e.g., in Example 1.1 the rule Rx is scheduled in the stack S1 by the rule Re1 applied by the PGP when $Op = \mathbf{e-reduce}$, and since an obvious restriction on RA is that it should be used in empty rules only, this happens when the operation executed by the PGP is either **e-reduce** or **he-reduce**, which are the operations devoted to the application of empty rules. When in **activate** a process descriptor must be extracted from the stack S1 then the transition in **h-reduce** is made if the activated rule is standard, otherwise, if the rule is empty the transition is in **he-reduce**.
The description of the PGP operations in Figure 1 can be made easier considering the possible subsets the PGP can use in a parsing process. First of all, two core operations are individuated: **scan**, and **activate**:

- **Scan** :

  In **scan** the input is scanned one word at a time, the scheduler is called and the process descriptors can be put in the stacks $S2, S3$;

- **Activate** :

  In **activate** the stacks $S1, S2, S3$ are checked; the transition is made depending on the highest priority non-empty stack; the process descriptor popped from that stack is then passed to the operation that must handle the fired process.

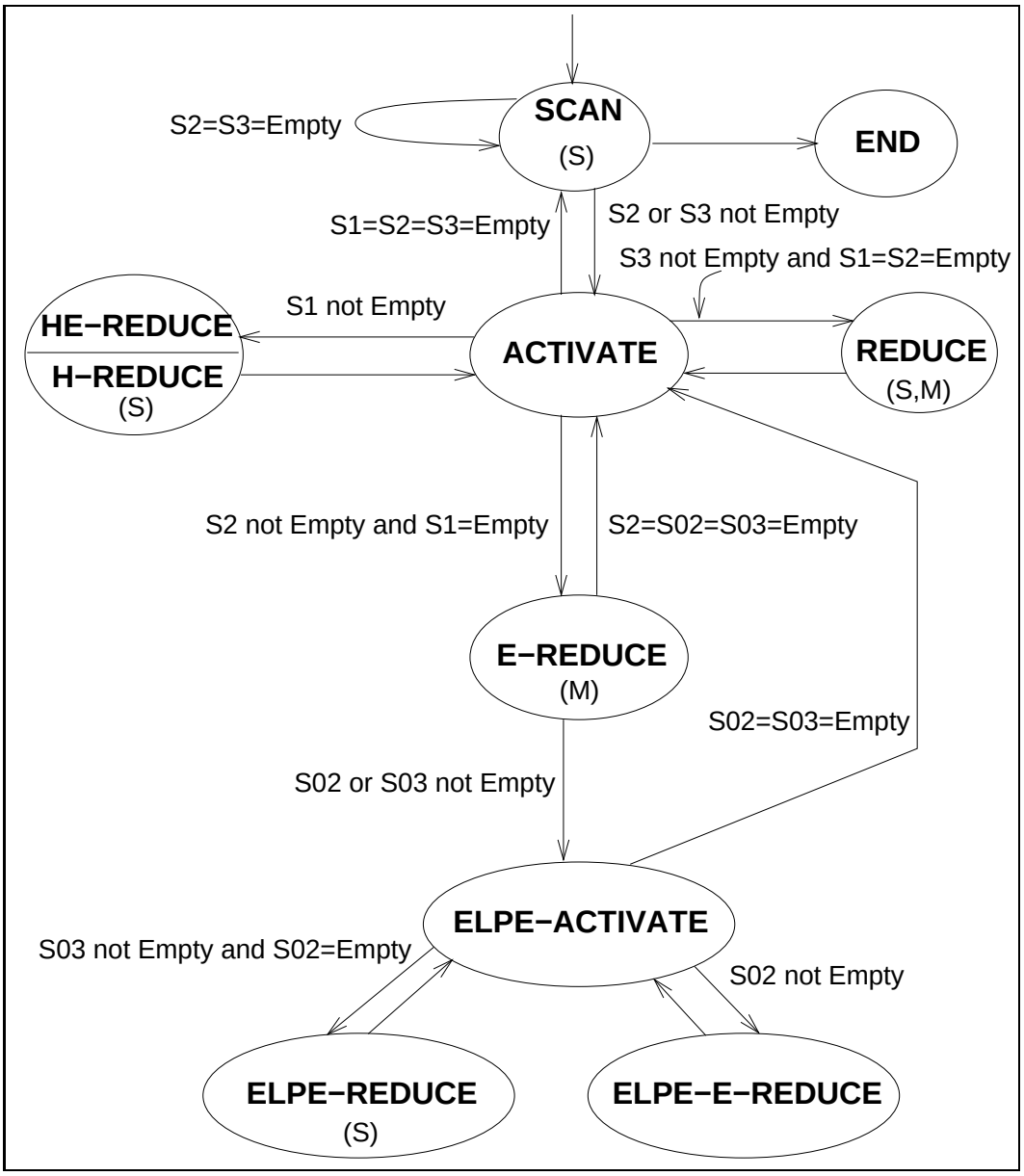


Figure 1: Operations for the serial-parallel PGP.

Definition 1.1 $CoreOps = \{\textbf{scan}, \textbf{activate}, \textbf{end}\}$.

The set $CoreOps$ is augmented with other operations in order to obtain some operations subsets to be associated with different classes of grammars.

Definition 1.2 $CFOps = CoreOps \cup \{\textbf{reduce}\}$.

Definition 1.3 $CSOps = CoreOps \cup \{\textbf{reduce}, \textbf{e-reduce}, \textbf{h-reduce}, \textbf{he-reduce}\}$.

Definition 1.4 $ElpeOps = \{\textbf{e-reduce}, \textbf{elpe-activate}, \textbf{elpe-e-reduce}, \textbf{elpe-reduce}\}$.

From these definitions the following properties follow:

Property 1.1 *$CFOps$ allows the recognition of context-free languages.*

Property 1.2 *$CSOps$ allows the recognition of context-sensitive languages.*

Property 1.3 *$ElpeOps$ allows the recognition of regular expressions. The set $CoreOps \cup ElpeOps$ has the same recognition power of $CFOps$.*

When the PGP makes transitions in only one of the above subsets its recognition power is the one associated with that subset as shown above; therefore, a Process Grammar G can be classified on the transitions that G forces the PGP to make, that is, which one of the operations subsets is fully used by the PGP.
A proof of Property 1.1 comes from the previous descriptions about the tasks of the operations in $CFOps$; for Property 1.2 a proof comes from Example 1.1 and the above description of the operations involved. In the set $CSOps$ there is the operation **he-reduce**; the simple motivation for its existence is the handling of empty rules fired by other empty rules by means of the function RA. In general, such a possibility might seem useless, unless some particular cases are considered. An example is given.

Example 1.2 *In this example the notation:*

$A \leftarrow [\alpha]\gamma[\beta]$

is used to indicate that the reduction $A \leftarrow \gamma$ is made within a possibly non-adjacent context determined by α and β, that is, the context used to make the reduction may be far away from γ. This also means that the process environment contains the reduced context plus the far context. Let consider the following reduction:

$\alpha_1\alpha_2 XY\beta_1\beta_2 \leftarrow \alpha_1\alpha_2\gamma_1\gamma_2\beta_1\beta_2$

where:

$X \leftarrow [\alpha_1]\gamma_1[\beta_1]$
$Y \leftarrow [\alpha_2]\gamma_2[\beta_2]$

It also must be assumed that this context dependency is not only structural but it depends on the success of some tests on information contained in the far constituents; therefore, the nodes for X and Y should contain new information as the result of actions performed on the extended context. Summarizing, X and Y depend on γ_1 and γ_2 both structurally and functionally, and only functionally on α_1, β_1 and α_2, β_2.
A first rough solution to the problem can be:

$$(empty \leftarrow \alpha_1\alpha_2\gamma_1\gamma_2\beta_1\beta_2)_{Re1}\{[\overset{\text{RA}}{\Rightarrow} (X \leftarrow \gamma_1)_{Rx}][\overset{\text{RA}}{\Rightarrow} (Y \leftarrow \gamma_2)_{Ry}]\}_{ACT(Re1)}$$

where the syntactic reduction is correct but the context dependencies are not; in fact, Rx and Ry do not have access through the process environment to the far contexts, thus lacking of the necessary information for performing correctly their actions.
It is clear now that a different chain of activations is needed, and the mechanism should work through the empty rule $Re1$ firing two other empty rules: this shows how the operation **he-reduce** *comes at work as these two rules must be handled by it. Here is another possible chain of activations:*

$$\begin{aligned}(empty \leftarrow \alpha_1\alpha_2\gamma_1\gamma_2\beta_1\beta_2)_{Re1}&\\ \{[\overset{\text{RA}}{\Rightarrow} (empty \leftarrow \alpha_1\alpha_2\gamma_1\gamma_2\beta_1)_{Re2}\{[\overset{\text{RA}}{\Rightarrow} (X \leftarrow \gamma_1)_{Rx}]\}_{ACT(Re2)}]&\\ [\overset{\text{RA}}{\Rightarrow} (empty \leftarrow \alpha_2\gamma_1\gamma_2\beta_1\beta_2)_{Re3}\{[\overset{\text{RA}}{\Rightarrow} (Y \leftarrow \gamma_2)_{Ry}]\}_{ACT(Re3)}]\}_{ACT(Re1)}&\end{aligned}$$

Even this solution has the same drawback of the previous one. It is clear at this point what the task of the operation **he-reduce** *is, in this case to apply the rules $Re2, Re3$, but it is not sufficient for the problem at hand since the final rules do not have the context available yet. A solution can be obtained extending the capabilities of the scheduling function RA in the following way: in addition to the rule to be fired it is optional to specify the far contexts that can be also accessed through the process environment. Therefore, the solution becomes the following:*

$$\begin{aligned}(empty \leftarrow \alpha_1\alpha_2\gamma_1\gamma_2\beta_1\beta_2)_{Re1}&\\ \{[\overset{\text{RA}}{\Rightarrow} (empty \leftarrow [\alpha_1]\gamma_1[\beta_1])_{Re2}\{[\overset{\text{RA}}{\Rightarrow} (X \leftarrow [\alpha_1]\gamma_1[\beta_1])_{Rx}]\}_{ACT(Re2)}]&\\ [\overset{\text{RA}}{\Rightarrow} (empty \leftarrow [\alpha_2]\gamma_2[\beta_2])_{Re3}\{[\overset{\text{RA}}{\Rightarrow} (Y \leftarrow [\alpha_2]\gamma_2[\beta_2])_{Ry}]\}_{ACT(Re3)}]\}_{ACT(Re1)}&\end{aligned}$$

In the PG model a grammar is a set of **defrule***'s and there are subsets of these rules that are built according to some particular implementation of parsing strategies. Given a classification of the possible strategies and mechanisms that can be realized it could be possible to provide a higher-level specification of the rules, closer to the classical formalism used for describing formal languages, used to generate the corresponding set of PG rules . Let consider Example 1.1 for a while. It is possible to derive the proper set of* **defrule***'s from a general context-sensitive reduction written in a given formalism, e.g., $Rx : (X/cde \leftarrow ab\_f)$, or $Rx : (\alpha_1 X_1/\gamma_1\beta_1 \ldots \alpha_k X_k/\gamma_k\beta_k)$. Thus, without entering in deeper details, the previous problem might be put in the following form:*

$Re1 : (\alpha_1\alpha_2 X/\gamma_1 Y/\gamma_2\beta_1\beta_2)$
$Rx : (X \leftarrow [\alpha_1]\gamma_1[\beta_1])$
$Ry : (Y \leftarrow [\alpha_2]\gamma_2[\beta_2])$

or even:

$Re1 : X/[\alpha_1]\gamma_1[\beta_1], Y/[\alpha_2]\gamma_2[\beta_2] \leftarrow \alpha_1\alpha_2\gamma_1\gamma_2\beta_1\beta_2$

The rewriting rules above describe a specific strategy whose meaning is clear, and they can be compiled into a set of PG rules.

(**defrule**
:RNAME Re1
:RED (empty ($\alpha_1\alpha_2\gamma_1\gamma_2\beta_1\beta_2$))
:ACT ((RA Re2 (α_1 β_1))(RA Re3 (α_2 β_2))))

(**defrule**
:RNAME Re2
:ST inactive
:RED (empty (γ_1))
:TEST {Some useful tests}
:ACT ((RA Rx (α_1 β_1))))

(**defrule**
:RNAME Re3
:ST inactive
:RED (empty (γ_2))
:TEST {Some useful tests}
:ACT ((RA Ry (α_2 β_2))))

(**defrule**
:RNAME Rx
:ST inactive
:RED (X (γ_1))
:ACT {Some useful actions})

(**defrule**
:RNAME Ry
:ST inactive
:RED (Y (γ_2))
:ACT {Some useful actions})

As already pointed out, the intermediate step with the rules $Re2, Re3$ should be undertaken because of the tests on the restricted contexts; therefore, if the tests hold then the final rules Rx, Ry perform the reductions on the restricted contexts and just at this point the actions have to process the information coming from them.

Property 1.3 is an extension to the core PGP: basically it can be used in recognizing regular expressions, but even though it is not limited to this kind of operations a description of how these operations work is given here, whereas a full description of how a set of PG rules can be generated from both a regular and a context-free grammar is given in Section

4.2. When the PGP enters in the three ELPE states it starts behaving like a finite state machine whose control is given by a set of **defrule**'s using a specific scheduling function, named ELPE (Extend Leftward the Process Environment). By means of this function one or more parallel chains of process activations are generated, and a global process environment which is the chain of the single process environments determined at every activation step is constructed. This function has a match-and-schedule task of just one of the rules provided as its arguments; thus, for (ELPE $r_1 \ldots r_k$) the matcher is called on this ordered set of rules and the first successfull matched rule is scheduled ignoring the remaining ones in the list. This function can be better used in the tests of a rule, thus it is supposed to return a true value when a rule is scheduled and a false value otherwise. Instead, whenever a sequence (ELPE r_1)...(ELPE r_k) is given then k parallel extensions are started through k different rules, and depending on the situation in the PGS it could happen that no one, or even all extensions could come to a success. The main purpose of the ELPE function is the leftward extension of the process environment of all the rules scheduled by it; this extension works building the process environment for the rule whose matching succeded along with the process environment of the rule that has called ELPE, and when an extension comes at end the resulting process environment is used by the last rule for building a node over a set of nodes determined by a set of rules and not by only one. An example is given.

Example 1.3 *In this example a PG for the regular expression $a^*bc^+d^*$ is described. How a set of rules in the extended PG for the parallel PGP can be generated from the regular grammar for the above regular expression is given in Section 4.2.1. Furthermore, the call to the match-and-schedule function is explicit here and occurs in the tests of the rules. In Section 4 the PG is extended so that the control information is added to the rules in separate slots, thus avoiding a possibly confusing mix of control actions and syntactic/semantic actions.*
The rules for the above regular expression are the following.

*(***defrule**
:RNAME *Rc*
:ST *active*
:RED *(empty (c)) ; $\epsilon \leftarrow c^*c \mid c^*cd^*$*
:TEST *((ELPE Rc Rb)))*

*(***defrule**
:RNAME *Rd*
:ST *active*
:RED *(empty (d)) ; $\epsilon \leftarrow d^*d$*
:TEST *((ELPE Rd Rc)))*

*(***defrule**
:RNAME *Rb*
:ST *inactive*
:RED *(empty (b)) ; $\epsilon \leftarrow bc^+d^*$*
:TEST *((ELPE Ra))*
:RATF *((RA RBb)))*

(**defrule**
    :RNAME *Ra*
    :ST *inactive*
    :RED *(empty (a)) ;* $\epsilon \leftarrow a^*abc^+d^*$
    :TEST *((ELPE Ra))*
    :TEST *((RA RBa)))*

(**defrule**
    :RNAME *RBb*
    :ST *inactive*
    :RED *(S (b))) ;* $S \leftarrow bc^+d^*$

(**defrule**
    :RNAME *RBa*
    :ST *inactive*
    :RED *(S (a))) ;* $S \leftarrow a^*bc^+d^*$

Rc, Rd are the starting rules; in fact, depending on the symbol scanned, one of the two is fired and applied in the operation **e-reduce***; then, depending on which rules in the ELPE calls get a match, the current process environment is passed to them. The rules Ra, Rb are applied in the operation* **elpe-e-reduce***, whereas RBb, RBa in* **h-reduce***. It is easier to see the possible chains of activations in Figure 2 where the nodes represent the processes and the arcs connect the processes to the next possible ones with their right-hand sides as labels. The way of processing this set of rules is not, in general, deterministic, and eventually it can get multiple process environments for which the analysis succeeds; therefore, when the PGP works using such a kind of PG generated from a regular grammar and enters the ELPE states then it behaves like a parallel non-deterministic finite state machine.*

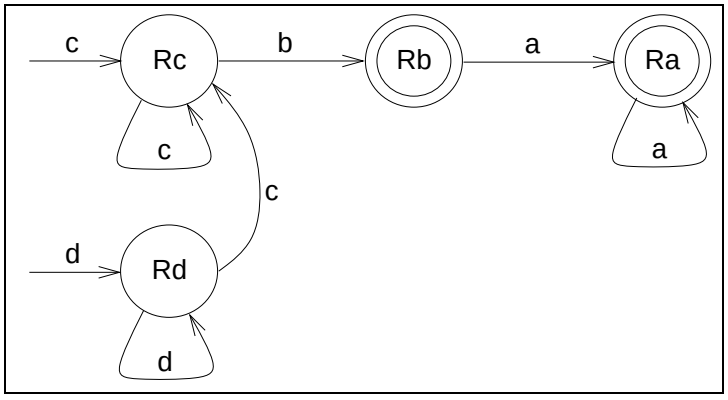


Figure 2: Transition graph for the example.

2 Parallel-Parallel PGP and the Extended PG

What is important to note about the serial-parallel PGP is that all the job is controlled by a unique SoC that evolves through several states representing during the parsing different analyses that are all stored in a unique common data structure; the parallelism only regards the analyses and is simulated by the serial device keeping track of them in the stacks and are worked out by the PGP setting the proper SoC for each one of them. Let see again what happens when a node of category c is scanned or a node of category A is reduced: the scheduler must find among all the really active rules having c or A as their right-corner which ones have a match in the PGS. The set of all these rules, before matching, is denoted by $\Pi(x)$, for any category x; obviously, no scheduling occurs for empty reduction rules. Henceforth, by the term *transition* it is meant the action of passing from the current running process to a set of next ones through the scheduling mechanism. Therefore, when a transition is made it is clear that the SoC does not contain properly the PGS as it is common and unique for all the processes, but it merely represents the correct environment for applying the fired process, whereas the others are awaiting in the priority queue. As the PGP carries on the computation and the SoC is modified at each step, a new environment is set, but the parsing data are only changed in the global PGS and the new SoC has access to the modified PGS; therefore, a graphical view of what is going on can be the one depicted in Figure 3, where the sequence of dots represents the temporal evolution of the SoC.

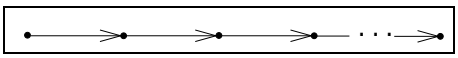


Figure 3: Serial transitions of the SoC in the serial-parallel PGP.

The parallel PGP is a parallel-parallel parser since the algorithm is able of handling many SoCs, each one evolving through one analysis task and describing the state of a serial-serial parser working on a PGS local to the SoC; thus, what happens when the set $\Pi(x)$ is proposed to the matcher is that for every rule whose matching succeeds there is a split of the current SoC such that there also is a replication of the PGS. This can be represented as shown in Figure 4 where each path represents an individual analysis as it would be carried on by a serial-serial parser.

Splitting up a SoC implies that at least one rule is scheduled for firing in every new SoC, but if there is some case when a set of rules mutually dependent is enqueued then this means that there must be a single splitting for all the set of mutual dependent rules. This rule is valid in general for the mechanisms seen so far; in fact, the scheduler has to split up the current SoC for every fired rule; the same is for the function ELPE when a sequence of calls to it occurs in an augmentation, and when a call to RA for a single reduction within a context is made, but when the function RA is called for multiple reductions within a context then these reductions are part of a unique rule and they depend on one each other, therefore the new SoC must contain all these fired rules to be applied next.

A first consequence of this functioning is that the stacks used in the serial-parallel PGP become useless in the parallel-parallel PGP since a unique queue is sufficient for handling the fired awaiting processes: the elements of the queue, the process descriptors, can contain

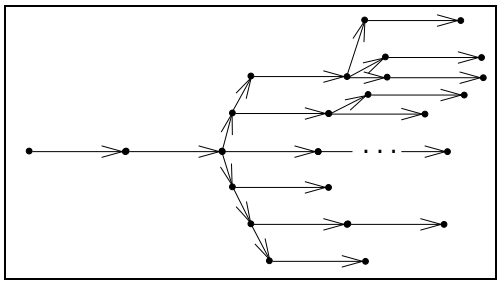


Figure 4: Parallel transitions of the SoCs in the parallel-parallel PGP.

a flag indicating which kind of operation must handle the processes. A second consequence is that, whenever a reduction is made in the PGS, it is not necessary to store in the newly splitted SoCs all the nodes of the old process environment but it is sufficient to transmit the reduced node only. Another advantage of the parallel-parallel architecture is the possibility of realizing some parsing strategies affecting the structure of the PGS local to the SoC that could not be possible otherwise in the serial-parallel implementation since every structure modification does not preserve in general the soundness of the parser, that is, some analyses might not be considered or cut out by altering the structure. It is interesting to see an example about this last point.

Example 2.1 *It is known that bottom-up parsers have problems in managing rules with common factors in the right-hand side such as in the following set of rules:*

$R1 : X \leftarrow ABCD$
$R2 : X \leftarrow BCD$
$R3 : X \leftarrow CD$
$R4 : X \leftarrow D$

since some or all of the above rules can be fired and build unwanted and useless nodes. A strategy called top-down filtering *has been stated in order to circumvent such a problem in bottom-up parsers ([4],[9],[10],[11]) where a top-down parser is simulated together with the bottom-up parser. The solution given here is not of this kind and can be used only in the parallel-parallel PGP. A first straightforward solution could use the match-and-schedule function ELPE; but another interesting solution can be proposed by modifying the parsing structure as long as new constituents are discovered to be adjacent to others. The parsing structure is modified by the function named ASR (Add Son Relation) in the following way: given a subtree rooted in a node of category X, the coverage of the node is extended to an adjacent node of category Y if the function ASR is called as (ASR X Y); the effect of such operation is shown in Figure 5.*
The function ELPE as described so far has been used in empty rules where the process environment of a rule is passed to a rule fired by the function itself; now it is necessary to extend this functioning to non-empty rules: whenever a rule R_i such that $RED(R_i) = (A \leftarrow \alpha)$ has a call to ELPE that succeeds on a rule R_j, then the process environment passed to R_j

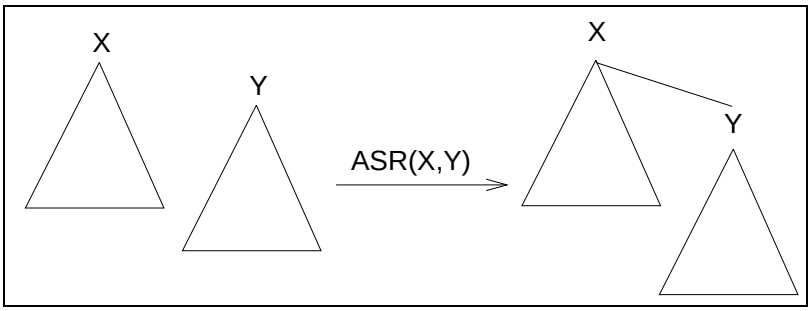


Figure 5: The effect of (ASR $X\ Y$) on a node X.

simply corresponds to the newly node built by R_i along with the previous one set up by the preceding extensions, and the reduction for R_j would result as $RED(R_j) = (\cdot \leftarrow \beta[A][\gamma])$, where R_j can be either empty or not, and the category A and the previous string γ matched to some process environments do not really occur but they are shown for clarity. The set of rules would then be the following.

(**defrule**
:RNAME RD
:ST inactive
:RED (X (D)) ; $X \leftarrow D$
:TEST ((ELPE RC)))

(**defrule**
:RNAME RC
:ST inactive
:RED (empty (C)) ; $\epsilon \leftarrow C[X] \equiv X \leftarrow C\,D$
:ACT ((ASR X C)
(ELPE RB)))

(**defrule**
:RNAME RB
:ST inactive
:RED (empty (B)) ; $\epsilon \leftarrow B[X] \equiv X \leftarrow BCD$
:ACT ((ASR X B)
(ELPE RA)))

(**defrule**
:RNAME RA
:ST inactive
:RED (empty (A)) ; $\epsilon \leftarrow A[X] \equiv X \leftarrow ABCD$
:ACT ((ASR X A)))

The result that can be obtained through these rules is a unique node X covering one of the

four possible cases without overgeneration of useless nodes. Furthermore, this technique must be only used in the parallel-parallel PGP as the soundness of the parser is preserved; this is why what happens in the current SoC is in one of the possible analyses; the problem in the serial-parallel parser would be that the structure modification can be shared by many other analyses, and rules that are fired along with the one calling (ASR X Y) requiring the adjacency between X and Y could not find adjacent the two nodes any longer when applied. The function ASR as it is used in the above rules performs leftward extensions of the one node's coverage; obviously, it can be also considered the rightward extension, but only in the leftward extension there must be an underlying scheduling effect. Let consider the situation in Figure 6 after the execution of (ASR X C).

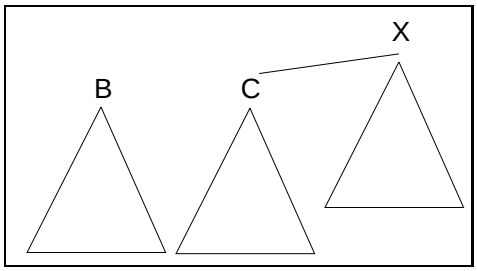
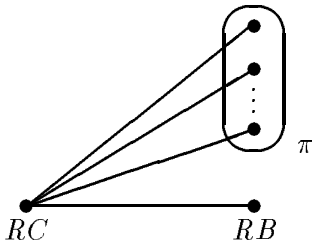


Figure 6: After (ASR X C).

The node X is now adjacent to the node B, and possibly to other subtrees. It is sensible to suppose that such action is similar to the construction of a node having the above structure. In general, the PGP by means of the operation **reduce** *schedules* $\Pi(X)$*, for some non-terminal X, but since the rule that calls ASR is empty, and no scheduling is involved in the operations applying empty rules, it would be proper that ASR, as its own side-effect, would schedule* $\Pi(X)$*. The splitting of the SoC where RC is applied is depicted in the following picture: the set* π *is the set of SoCs for the rules that has been scheduled through ASR, and RB is the rule scheduled by ELPE.*



The parallel architecture has been extended with respect to the serial one in two main directions: extension of the PG model, with special regard to control specification and modularity issues, and parallel architecture of the PGP.
The extension of the PG model should keep into account some issues dealing with control, modularity, and grammar design and construction. Grammar design is not an easy task,

especially when a grammar of big dimensions is designed. A wide covering grammar can have reasonably some hundreds of rules, and the control of such a big stuff can become so hard that it could be easier not to take it into account. For practical applications this is not an advantage and it should be necessary to face the problem, anyway. Furthermore, the grammar designer - usually a person that deals with syntactic and semantic tasks - cannot be involved in such control matters. A possible alternative way is to keep separated control and pure grammar matters, and to have a further specification concerning the organization of a PG that does not involve the control in an explicit way, so that the grammar designer's work can be made easier. The PG model has been extended in three main ways: it is possible to define some *control sets* and some *control actions* in every rule; a PG can be partitioned in *clusters* of rules; due to the previous extensions the **defrule** format changes so that for every rule in the PG there is a grammar rule with syntactic and semantic augmentations, and a control rule with control sets/actions augmentations, such that both can be defined by means of two different rule formats; furthermore, the scheduling mechanism must be revised for the handling of clusters and control sets/actions.

2.1 Control Sets and Control Actions

For the PG model seen so far there is a main scheduling mechanism in the PGP; this mechanism can be partially overridden sometime through the use of special functions specifying explicitly which rules can be fired and under which conditions. Some non-standard grammar models that restrict in some ways the rewriting actions for a rule have been studied [2], such as matrix and programmed grammars, and cooperating grammar systems [7]. These restrictions usually provide the sequence of rules in a derivation, which rules should follow next, which rules must be chosen depending on the current derivation string. Some of these already are possible, as seen in the previous examples, but two requirements are in order for grammar design: the modularity of the rules format, and the need of considering all possible kinds of actions that can be undertaken when a rule applies. A natural consequence is that the augmentations can only contain sequences of operations regarding syntactic and semantic analysis, and every augmentation should be associated with a corresponding set containing two kinds of information: control sets, and control actions. Control sets can explicitly specify the next rules to be scheduled, whereas control actions can specify which operations affecting control structures should be performed, e.g., RA and ELPE should be called in this kind of augmentations. What control sets should do is to override the standard firing mechanism of the PGP. This can work whenever a reduce operation is performed for a reduction rule $RED(R) = (A \leftarrow \alpha)$; the scheduler must then consider the set $\Pi(A)$, but if the control set associated to $ACT(R)$ is not empty then the scheduler has to take it in place of $\Pi(A)$. These sets are introduced and described in Section 2.3 where the rule format is discussed.

2.2 PG Clusters

The PG model for the serial-parallel PGP does not have any kind of structuring among the rules, so that it is easy to understand how transitions are performed in a PG. Figure 7 gives an idea of this transition mechanism regulated by the built-in scheduling strategy and the procedural specifications given in the rules. A common problem in the serial-parallel PGP

dealing with control issues is that different sets of rules, each one dedicated to the treatment of a specific phenomenon, could interfere one each other generating unwanted analyses or useless results. A higher logical level can be introduced in a PG, thus increasing modularity and insulation of special purpose sets of rules that could not interact freely anymore with other special purpose sets as instead it might happen in a situation as depicted in Figure 7. A PG can be partitioned in clusters of rules following a logical design; in fact, the clusters

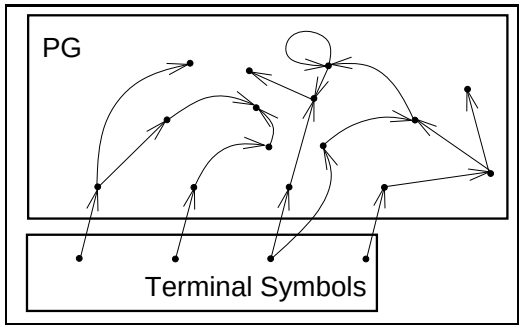


Figure 7: Transitions in a PG.

must subdivide logically the PG in a partition defined by the grammar designer so that a cluster is an independent special purpose set of rules (ideally, they could be considered as PG subgrammars), designed for a specific recognition task. Reasoning in terms of clusters is at a higher level than reasoning in terms of rules; this means that it can be easier to understand how transitions are performed in a clustered PG, as shown in Figure 8. One

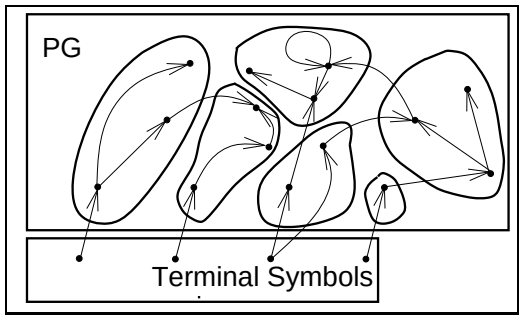


Figure 8: Transitions in a clustered PG.

of the main reasons for insulation between different rules is that clusters does not have any visibility of rules in other clusters, and the only interaction is through the scheduling mechanism. In fact, it is necessary to state how the built-in scheduling mechanism is affected by introducing clustering; how and when transitions from rules to other rules of the same cluster are performed; how and when transitions from a cluster to other clusters are performed; how all the previous kinds of transitions affect the splitting of the current SoC. It also can be useful to have an analogous mechanism as that described above for

control sets: the built-in scheduling mechanism can be overridden by naming the rules in the same cluster to be fired and/or naming the clusters where the rules to fire are. These scheduling matters are described in Section 2.4.

2.3 Rule Format

The extended format for clustered PG rules in the parallel-parallel PGP is composed by a grammar rule and by a control rule. In a grammar rule all the information and operations dealing with syntactic and semantic analysis to be carried on in the context determined in the process environment are given. A control rule provides all the information and operations dealing with the control of the rule activity itself, and with the possible actions to be performed for implementing some parsing strategies where such control matters have a central role in terms of efficiency and rule firing control. Two different formats are introduced for each kind of rule: the **dgr** format for the grammar rule, and the **dcr** format for the control rule. The rule formats for non-empty rules are as follows:

(**dgr**
:CLUSTER *cname* ; Cluster name
:RNAME *rname* ; Rule name
:RED $(A\ (\alpha\ c))$; α possibly empty
:RAMF *ramf*
:TEST *tests*
:ACT *actions*
:RATF *ratf*)

(**dcr**
:CLUSTER *cname*
:RNAME *rname*
:ST *status*
:MFSET (:RULES *f-rules-list* ; Match failure set rules list
:CLUSTERS *f-clusters-list*) ; Match failure set clusters list
:SSET (:RULES *s-rules-list* ; Success set rules list
:CA *s-control-actions* ; Success set control actions
:CLUSTERS *s-clusters-list*) ; Success set clusters list
:TFSET (:RULES *f-rules-list* ; Test failure set rules list
:CA *f-control-actions*)) ; Test failure set control actions

Depending on which one of the grammar rule augmentations is applied there is a corresponding control rule augmentation that is taken for control operations. The tests only do not have a corresponding control augmentation, whereras for the others the associations are so made:

$RAMF(r) \rightarrow MFSET(r)$
$ACT(r) \rightarrow SSET(r)$
$RATF(r) \rightarrow TFSET(r)$

where for every control augmentation the control sets are $RULES(MFSET(r))$, $CLUSTERS(MFSET(r))$, $RULES(SSET(r))$, $CLUSTERS(SSET(r))$, and $RULES(TFSET(r))$, whereas $CA(SSET(r))$, $CA(TFSET(r))$ contain control actions. If `gr-apply` is the function for applying the grammar augmentations, and `cr-apply` is the function for handling the control augmentations during a rule application, the following code shows how the augmentations of a non-empty PG rule should be applied:

```
if match(RED(r))
     then if gr-apply(TEST(r))
               then gr-apply(ACT(r));cr-apply(SSET(r));
               else gr-apply(RATF(r));cr-apply(TFSET(r));
     else gr-apply(RAMF(r));cr-apply(MFSET(r));
```

Therefore, $MFSET(r)$ is a control augmentation providing a control set of rules and/or clusters to be fired next when the match for $RED(r)$ fails and $RAMF(r)$ are applied; $SSET(r)$ is a control augmentation providing the same control sets as for $MFSET(r)$, and, in addition, some control actions in $CA(SSET(r))$ to be applied when tests $TEST(R)$ succeed and $ACT(r)$ are applied. A similar structure is for $TFSET(r)$ that are taken when tests $TEST(r)$ fail. It is needed that all the rules occurring in *f-rules-list* must have c as their right-corner.
The rule formats for empty rules are as follows:

(**dgr**
:CLUSTER *cname* ; Cluster name
:RNAME *rname* ; Rule name
:RED (*empty* (α *c*)) ; α possibly empty
:RAMF *ramf*
:TEST *tests*
:ACT *actions*
:RATF *ratf*)

(**dcr**
:CLUSTER *cname*
:RNAME *rname*
:ST *status*
:MFSET (:RULES *f-rules-list* ; Match failure set rules list
:CLUSTERS *f-clusters-list*) ; Match failure set clusters list
:CTA *control-tests* ; Control Tests
:SSET (:CA *s-control-actions*) ; Success set control actions
:TFSET (:RULES *f-rules-list* ; Test failure set rules list
:CA *f-control-actions*)) ; Test failure set control actions

Due to the special form of the reduction rule, the use of this kind of PG rules involves a greater flexibility and power for parsing control. In fact, empty rules provide a process environment that can be tested for taking decisions about what to do next, such as undertaking further actions, further analyses, or recovery strategies. This is the reason why in the

dcr format there is a slot $CTA(r)$ for control tests that must be associated with the tests $TEST(r)$. The control tests have a higher priority than $TEST(r)$; in fact, if control tests fail then the tests are not considered because this kind of rules, unlike non-empty rules, has a control task to be accomplished, and this implies that when $CTA(r)$ is used it would be useful not to use the grammar rule augmentations for syntactic/semantic analysis. The control tests also are the right place where to put calls to ELPE when the PGP must work in the ELPE states. The following code shows how the augmentations of a PG empty rule should be applied:

```
if match(RED(r))
     then if cr-apply(CTA(r))
               then if gr-apply(TEST(r))
                         then gr-apply(ACT(r));cr-apply(SSET(r));
                         else gr-apply(RATF(r));cr-apply(TFSET(r));
               else cr-apply(TFSET(r))
     else gr-apply(RAMF(r));cr-apply(MFSET(r));
```

The control set $RULES(SSET(r))$ is not considered here since it should contain next applicable rules whose right-corner should be the left-hand side of the reduction rule, but in this case the reduction rule does not have any left-hand side and then it is useless.

2.4 Scheduling Mechanism

Even if the scheduling task can be located in a single process called scheduler, it is better to identify two distinct processes for this task, the scheduler and the matcher, since the first one has to access the PG structures for getting all the rules that are schedulable in a given moment, and for setting up all the information under the form of process descriptors to be sent to the matcher that has to access the PGS for finding the reduction sets.
The built-in scheduling mechanism is based on the structure of the PG and on the RC strategy. A PG is a set of clusters $\mathcal{C} = \{C_1, \ldots, C_{|\mathcal{C}|}\}$ where $C_i = \{r_{i1}, \ldots, r_{id_i}\}$, for $i = 1, \ldots, |\mathcal{C}|$; and RC_i is the set of the right-corners in C_i. For every terminal and non-terminal symbol x, and for every cluster C_i the set $\Pi(C_i, x)$ is:

$$\Pi(C_i, x) = \{r_{ik} \mid RED(r_{ik}) = (\cdot \leftarrow \alpha x), 1 \leq k \leq d_i\}$$

For every right-corner c the set of clusters having c as right-corner in at least one of their rules is:

$$CS(c) = \{C_i \mid C_i \in \mathcal{C}, \exists r_{ik}\, RED(r_{ik}) = (\cdot \leftarrow \alpha c), 1 \leq k \leq d_i, \text{ for some } i \in \{1, \ldots, |\mathcal{C}|\}\}$$

or, that is the same,

$$CS(c) = \{C_i \mid C_i \in \mathcal{C}, c \in RC_i, \text{ for some } i \in \{1, \ldots, |\mathcal{C}|\}\}$$

The two main events for scheduling are terminal node scanning and non-terminal node reduction:

- **Scanning** :

  When a terminal node of category c is scanned by operation **scan** then the set $CS(c)$ is taken and for every $C_j \in CS(c)$ the rules in $\Pi(C_j, c)$ are proposed to the matcher;

- **Reduction** :

  When a rule $r_{ik} \in C_i$ with $RED(r_{ik}) = (A \leftarrow \alpha c)$ is successfully applied then:

  - The set $\Pi(C_i, A)$ is proposed to the matcher; successful matches on this set allow transitions within the current cluster C_i;
  - For every $C_j \in CS(A)$ then the set $\Pi(C_j, A)$ is proposed to the matcher; successful matches on this set allow transitions to other clusters.

  For every rule as determined above in the two main scheduling events for which the matcher finds at least a process environment, the current SoC for r_{ik} is replicated for the application of the fired rule by means of the proper operation. This is the built-in scheduling mechanism. The control augmentations can override or extend this functioning when:

  - $RULES(MFSET(r_{ik}))$ is not empty; then the rules in this set are proposed to the matcher and they must have c as their right-corner;
  - $CLUSTERS(MFSET(r_{ik}))$ is not empty; then for every C_j in this set $\Pi(C_j, c)$ is proposed to the matcher;
  - $RULES(SSET(r_{ik}))$ is not empty; then this set replaces $\Pi(C_i, A)$, and only rules of the same cluster must be given there, i.e., $RULES(SSET(r_{ik})) \subset \Pi(C_i, A)$;
  - $CA(SSET(r_{ik}))$ is not empty; then the control actions must be applied;
  - $CLUSTERS(SSET(r_{ik}))$ is not empty; then this set replaces $CS(A)$, i.e., $CLUSTERS(SSET(r_{ik})) \subset CS(A)$;
  - $RULES(TFSET(r_{ik}))$ is not empty; then the rules in this set must have the same right-hand side of r_{ik} and they are proposed to the matcher;
  - $CA(TFSET(r_{ik}))$ is not empty; then the control actions are applied.

  When an empty rule $r_{ik} \in C_i$ with $RED(r_{ik}) = (empty \leftarrow \alpha c)$ is successfully applied no automatic scheduling is provided by the PGP unless:

  - $RULES(MFSET(r_{ik}))$ is not empty; then this set is proposed to the matcher and all the rules in it must have c as their right-corner;
  - $CLUSTERS(MFSET(r_{ik}))$ is not empty; then for every C_j in this set $\Pi(C_j, c)$ is proposed to the matcher;
  - $CA(SSET(r_{ik}))$ is not empty; then the control actions are applied;
  - $RULES(TFSET(r_{ik}))$ is not empty; then the rules in this set must have the same right-hand side of r_{ik} and they are proposed to the matcher;
  - $CA(TFSET(r_{ik}))$ is not empty; then the control actions are applied.

If no transitions are possible, i.e., no matches found and no control provided in the rule, then the next operation is a **scan** of the input, with a single replication of the current SoC.

3 Parse Graph Structure

The Parse Graph Structure (PGS) is built by the parser when grammar rules are applied. If $w = w_1 \ldots w_n$ is the input string, the initial PGS is composed by the set of terminal nodes $<0,\$>,<1,w_1>,\ldots,<n,w_n>,<n+1,\$>$, where the nodes $0, n+1$ represent border markers for the sentence. The non-terminal nodes are numbered starting from $n+2$.

Definition 3.1 *A PGS is a triple (N_T, N_N, T), where: N_T is the set of the terminal nodes numbers $\{0, 1, \ldots, n, n+1\}$; N_N is the set of the non-terminal nodes numbers $\{k \mid k \geq n+2\}$; T is the set of the subtrees built in the PGS.*

Thus the elements of N_N and N_T are numbers identifying nodes of the PGS, and their structure along with the structure of the subtrees in T are described later. The following definitions introduce some relations existing among the nodes in a PGS.

Definition 3.2 *If $k \in N_N$ then the node $i \in N_T$ labeling w_i at the beginning of the clause covered by k is said to be the* **left corner leaf** *of k, denoted as $lcl(k)$. If $k \in N_T$ then $lcl(k) = k$.*

Definition 3.3 *If $k \in N_N$ then the node $j \in N_T$ labeling w_j at the end of the clause covered by k is said to be the* **right corner leaf** *of k, denoted as $rcl(k)$. If $k \in N_T$ then $rcl(k) = k$.*

Definition 3.4 *If $k \in N_N$ then the node $h \in N_T$ that immediately follows the right corner leaf of k, $rcl(k)$, is said to be the* **anchor leaf** *of k, denoted as $al(k)$, and $al(k) = h = rcl(k)+1$. If $k \in N_T - \{n+1\}$ then $al(k) = k+1$.*

Definition 3.5 *If $k \in N_T - \{0\}$ then the* **set of the anchored nodes** *of k, denoted as $an(k)$, is the set of the nodes such that their anchor leaf is k, that is, the only terminal node $k-1$ and some other non-terminal nodes j such that $al(j) = k$. Thus, $an(k) = \{j \mid j \in N_N, al(j) = k\} \cup \{k-1\}$.*

From this last definition it follows that, at the initial parsing time, for every $k \in N_T - \{0\}$, $|an(k)| = 1$ and $an(k) = \{k-1\}$.
The nodes are labeled by a number and also by a category symbol, for non-terminal nodes, and by a set of category symbols, named interpretations, for terminal nodes. In fact terminal nodes are attached to lexical entries that, in general, may have more than one interpretation.

Definition 3.6 *If $k \in N_T$ and it has p interpretations coming from the lexical item w_k then the set of its interpretations is $ints(k) = \{(0, c_0), (1, c_1), \ldots, (p-1, c_{p-1})\}$ where c_i are terminal categories. If $k \in N_N$ it has only one interpretation, thus: $ints(k) = \{(0, A) \mid$ for some non-terminal category $A\}$.*

At this point it is possible to define the structure of the set of the subtrees T of the PGS.

Definition 3.7 *If $k \in N_T$ then the subtree $T(k)$ rooted in k is represented by the 4-tuple $< lcl(k), rcl(k), an(k), ints(k) >$. If $k \in N_N$ then the subtree $T(k)$ rooted in k is represented by $< lcl(k), rcl(k), sons(k), \{(0, A)\} >$, where A is a non-terminal category; $sons(k)$ is the set of the direct descendants of k: $sons(k) = \{s_1, \ldots, s_p \mid s_i = (h, int), h \in N_T \cup N_N, int \in \{0, 1, \ldots\}$, for $i = 1, \ldots, p\}$.*

Definition 3.8 *A* **node-interpretation identifier** *is a pair* (h, int) *such that* $h \in N_T \cup N_N$ *refers to* $T(h)$*, and* $int \in \{0, 1, \ldots\}$ *refers to* $(int, c) \in ints(h)$*, for some category* c*. Henceforth,* int *represents a variabile on the ordered set* $\{0, 1, \ldots\}$ *numbering in this order the interpretations of a terminal node.*

Definition 3.9 *If* V_N, V_T *are the sets of the non-terminal and terminal symbols of the grammar, respectively, the function* **cat** $: (N_N \cup N_T) \times \{0, 1, \ldots\} \rightarrow (V_N \cup V_T) \cup \{Nil\}$ *returns the category of a node-interpretation identifier in the following way:*

cat(k,int): if (int,c)∈ints(k) then return c else return Nil.

In the following, whenever the interpretation number is not necessary for the purpose of the description or it is clear from the context, it will be omitted from the list of the arguments.

Summarizing, from the preceding definitions the initial PGS for a sentence $w = w_1, \ldots, w_n$ is:

$N_T = \{0, 1, \ldots, n, n+1\}$
$N_N = \{\}$
$T = \{T(0), T(1), \ldots, T(n), T(n+1)\}$

and

$T(0) = < 0, 0, \{\}, \{(0, \$)\} >$
$T(i) = < i, i, \{i-1\}, \{(0, w_i)\} >$ for $i = 1, \ldots, n$
$T(n+1) = < n+1, n+1, \{n\}, \{(0, \$)\} >$

where all the terminal nodes have been considered with one interpretation.
The parser starts with this PGS reducing new nodes from the already existing ones. If for some $k \in N_N$, its subtree is:

$T(k) = < lcl(k), rcl(k), sons(k), \{(0, A)\} >$

and $sons(k) = \{(h_1, int_1), \ldots, (h_p, int_p)\}$ are the direct descendants of k such that $(int_i, c_i) \in ints(h_i)$ for $i = 1, \ldots, p$, then the node k has been reduced from $sons(k)$ by some grammar rule whose reduction rule has the form $(A \leftarrow c_1 \ldots c_p)$, and the following holds:

$lcl(k) = lcl(h_1)$
$rcl(h_1) = lcl(h_2) - 1$
$rcl(h_2) = lcl(h_3) - 1$
$\ldots$
$rcl(h_{p-1}) = lcl(h_p) - 1$
$rcl(h_p) = rcl(k)$

The following definition follows:

Definition 3.10 *If $\{s_1, \ldots, s_p\}$ is a set of nodes in the PGS, then the subtrees $T(s_1), \ldots, T(s_p)$ are said to be* **adjacent** *when $rcl(s_i) = lcl(s_{i+1}) - 1$, or, alternatively, $al(s_i) = lcl(s_{i+1})$, for $i = 1, \ldots, p-1$.*

4 Process Grammar

Definition 4.1 *A* **Process Grammar** *G is a 5-tuple $(V_T, V_N, V_S, \mathcal{C}, F)$ where:*

- *V_T is the set of terminal symbols; V_N is the set of non-terminal symbols;*
- *V_S is the set of the special symbols; $S, empty \in V_S$, where S is the root symbol of G, and empty an empty category;*
- *$\mathcal{C} = \{C_1, \ldots, C_{|\mathcal{C}|}\}$ is the set of the clusters of G, and every cluster C_i contains a set of rules, $C_i = \{r_{i1}, \ldots, r_{id_i}\}$, such that every rule is of the format as defined in Section 2.3;*
- *$F = \{RA, ELPE, ASR, \ldots\}$ is the set of functions the rules can call in their augmentations.*

The scheduler and the matcher are directly connected with the structure of the PG. To the end of introducing these two main components of the PGP it is necessary to give some definitions regarding both the structure of PGS and the PG organization.

4.1 More about PGS, the Matcher, and the Scheduler

The purpose of the matcher is to find a set of adjacent subtrees matching a right-hand side of a reduction rule. Let $(A \leftarrow c_1\, c_2\, c_3)$ be a reduction rule. Since the RC strategy is at the base of rule access, the matcher starts with a node matching the category c_3. Afterwards, it has to sail in the PGS through adjacent subtrees by using the sets of the anchored nodes in the terminal nodes. It follows from Definition 3.10 that if $k \in N_N$ then the subtrees adjacent to $T(k)$ are given by $an(lcl(k))$, whereas if $k \in N_T$ then the adjacent subtrees are given by $an(k)$. From this remark it is possible to give the following definition.

Definition 4.2 *If (N_T, N_N, T) is a PGS for an input sentence w, with $|w| = n$, then the* **adjacency tree** *for the PGS is built as follows:*

- *$n+1$ is the root of the adjacency tree;*
- *for every $k \in N_T - \{0,1\} \cup N_N$, the sons of k are the nodes in $an(lcl(k))$, unless $an(lcl(k)) = \{0\}$.*

Any move from a node k to one of its sons h in the adjacency tree represents a move from a subtree $T(k)$ to one of its adjacent subtrees $T(h)$ in the PGS. During a matching process, this means that a constituent of the right-hand side has been consumed, and that the match process is finished when the first symbol is matched. The adjacency tree also provides further useful information for optimizing the search during a match. For every node k, if the longest path from k to a leaf is considered, its length is an upper bound

for the length of the right-hand side still to consume (supposed that k has been matched), and since the sons of k are the nodes in $an(lcl(k))$ the longest path is always given by the sequence of the terminal nodes from the node 1 through the node $lcl(k)-1$. Thus its length is just $lcl(k)-1$.

Property 4.1 *If (N_T, N_N, T) is a PGS, $(\cdot \leftarrow c_1 \ldots c_p)$ is a reduction rule, and $T(k) \in T$ such that $cat(k) = c_p$, then:*

- *the string $c_1 \ldots c_p$ is* **matchable** *iff $p \leq lcl(k)$;*
- *for $i = p, \ldots, 1$, c_i is* **partially matchable** *to a node $h \in (N_T \cup N_N)$ iff $cat(h) = c_i$ and $i \leq lcl(h)$.*

Property 4.1 along with the adjacency relation provide a method for an efficient navigation within the PGS among the subtrees. This navigation is performed by the matcher as visiting the adjacency tree in a pre-order fashion. It is easy to see that a pre-order visit of the adjacency tree scans all possible sequences of the adjacent subtrees in the PGS; also, Property 4.1 provides a shortcut for avoiding useless moves when matchable conditions do not hold. When a match ends the matcher returns one or more sets of nodes satisfying the following conditions.

Definition 4.3 *A set $RSet = \{(n_1, int_1), \ldots, (n_p, int_p)\}$ is a match for a string $c_1, \ldots, c_p$ iff $cat(n_i, int_i) = c_i$, for $i = 1, \ldots, p$, and $T(n_i)$ is adjacent to $T(n_{i+1})$, for $i = 1, \ldots, p-1$. The set RSet is called a* **reduction set**.

The adjacency tree shows the hypothetical search space for searching the reduction sets in a PGS, thus it is not a representation of what memory is actually required to store the useful data for such a search. A more suitable representation is given in the following definition.

Definition 4.4 *If (N_T, N_N, T) is a PGS, an* **adjacency directed graph**, *or* **adjacency digraph**, *is represented by means of the lists of the anchored nodes and by the pointers in non-terminal nodes to the left corner leaf as follows:*

- *for any $k \in N_T$, k has outgoing arcs directed to the nodes in $an(k)$;*
- *for any $k \in N_N$, k has one outgoing arc directed to $lcl(k)$.*

Storing the adjacency information only in terminal nodes it is possible to have just a single pointer in non-terminal nodes towards the left corner leaf where this information is contained. If $D = |N_T| + |N_N|$ is the dimension of the PGS then there are only $|N_T|$ adjacency lists and $|N_N|$ single pointers, instead of D adjacency lists. Considering the serial-serial PGP, it is possible that for every right-corner c a set of rules can be accessed and proposed to the parser if for that set of rules the matchability property holds. Afterwards, it is possible that many of these rules can be applied and it results that the set of the anchored nodes stored in the left corner leaf k of the right-corner can be such that $|an(k)| \geq 2$, and in general $|an(k)| \geq 1$. For the parallel-parallel PGP there is an upper-bound since the PGS is replicated for every fired rule, and should contain just one analysis without ambiguities. Therefore the following property holds.

Property 4.2 *If (N_T, N_N, T) is a PGS built in the parallel-parallel PGP then for every $k \in N_T$ $1 \leq |an(k)| \leq 2$.*

What the matcher returns is a reduction set that must be finally set up by the scheduler as a process environment for the new SoC where the rule is applied.

Definition 4.5 *Let $\{(n_1, int_1), \ldots, (n_p, int_p)\}$ be a reduction set for $RED(r_{ik}) = (A \leftarrow c_1 \ldots c_p)$, $h \in N_N$ be the new node for A such that $T(h)$ is the new subtree created in the PGS, and $(0, A) \in ints(h)$, then the* **process environment** *for r_{ik}, denoted by $ProcEnv(r_{ik})$, is:*

$ProcEnv(r_{ik}) = \{(h, 0), (n_1, int_1), \ldots, (n_p, int_p)\}$

If $RED(r_{ik}) = (empty \leftarrow c_1 \ldots c_p)$ then:

$ProcEnv(r_{ik}) = \{(n_1, int_1), \ldots, (n_p, int_p)\}$

How the scheduling mechanism works has been already introduced in Section 2, but due to the definition of real state and the statement of Property 4.1 it is needed a slight revision of the mechanism. One of the main conditions for a rule to be fired is that it must be really active. Here are some definitions regarding the state of the rules.

Definition 4.6 *$ST(r)$ is the* **original state** *of r.*
If $ST(r) = active$ then r is said to be **originally active**.
If $ST(r) = inactive$ then r is said to be **originally inactive**.

The original state cannot be modified; on the contrary, any rule has a run-time state, named real state, that can be modified through two functions: RD, standing for rule disable, and RE, standing for rule enable. These two functions do not change the original state, instead they change a mask flag associated to the state of a rule, named $MSK(r)$, and stored in the current SoC; this means that the real state is not global, but local to the SoCs. Beside this, these two functions can work only on the rules in the cluster running in the current SoC. If the active state is coded as a binary digit 1, and the inactive state as a binary digit 0, the mask flag is set at the initial time to 0 indicating no mask, and the real state of r is given by the original state. When $MSK(r) = 1$, then a mask is on and it masks the original state, thus the real state must be the negation of $ST(r)$. Therefore, the real state of a rule is given by the exor between $ST(R)$ and $MSK(R)$.

Definition 4.7 *The* **real state** *of a rule r is given by $(ST(r)\, exor\, MSK(r))$.*
If $(ST(r)\, exor\, MSK(r)) = 0$ then r is said to be **really inactive**.
If $(ST(r)\, exor\, MSK(r)) = 1$ then r is said to be **really active**.

Definition 4.8 *The function RE is in F and it is defined as follows: if $r_{ik} \in C_i$ and C_i is the active running cluster in the current SoC then*

`RE(`r_{ik}`):` **if** `(ST(`r_{ik}`) exor MSK(`r_{ik}`))=0` **then** `MSK(`r_{ik}`):=NOT(MSK(`r_{ik}`))`

Definition 4.9 *The function RD is in F and it is defined as follows: if $r_{ik} \in C_i$ and C_i is the active running cluster in the current SoC then*

`RD(`r_{ik}`):` **if** `(ST(`r_{ik}`) exor MSK(`r_{ik}`))=1` **then** `MSK(`r_{ik}`):=NOT(MSK(`r_{ik}`))`

Definition 4.10 *If $G = (V_T, V_N, V_S, \mathcal{C}, F)$ is a PG, then RC_i is the set of the right-corners in C_i, for $i = 1, \ldots, |\mathcal{C}|$, and*

$$\mathcal{RC} = \bigcup_{1 \leq i \leq |\mathcal{C}|} RC_i$$

Definition 4.11 *For every $x \in \mathcal{RC}$ then $Lm(x)$ is the length of the longest right-hand side having x as right-corner.*

Definition 4.12 *For every $x \in \mathcal{RC}$ then:*

$CS(x) = \{C_i \mid C_i \in \mathcal{C}, x \in RC_i$, *for some* $i \in \{1, \ldots, |\mathcal{C}|\}\}$

Definition 4.13 *For every $i = 1, \ldots, |\mathcal{C}|$, for every $x \in RC_i$:*

$P(C_i, x, l) = \{r_{ik} \mid RED(r_{ik}) = (\cdot \leftarrow \alpha\ x), 1 \leq |\alpha\ x| \leq l\}$ *for* $l = 1, \ldots, Lm(x)$

At this point it is possible to define which sets of rules can be accessed for scheduling, using also the result given by Property 4.1, in the two main scheduling events:

- **Scanning** :

  When a node $h \in N_T$ has been scanned and $k = min\{h, Lm(cat(h))\}$ then the scheduler has to access the clusters in $CS(cat(h))$, and for every cluster $C_j \in CS(cat(h))$ the rules in $P(C_j, cat(h), k)$ whose real state is active are accessed. Therefore:

Definition 4.14 *Let $h \in N_T$ be a node scanned, then the schedulable rules are:*

$\Pi(C_j, cat(h), k) = \{r_{ik} \in P(C_j, cat(h), k) \mid k = min\{h, Lm(cat(h))\}$, r_{ik} *really active*$\}$

for every $C_j \in CS(cat(h))$.

- **Reduction** :

  Whenever a node $h \in N_N$ has been reduced by a rule $r_{ik} \in C_i$ such that $RED(r_{ik}) = (A \leftarrow \alpha)$, $cat(h) = A$, $k = min\{lcl(h), Lm(A)\}$, then the scheduler has to access the rules in $P(C_i, A, k)$ whose real state is active. Therefore:

Definition 4.15 *Let $h \in N_N$ be a node reduced by $r_{ik} \in C_i$, $cat(h) = A$, then the schedulable rules of the same cluster C_i are:*

$\Pi(C_i, A, lcl(h)) = \{r_{ik} \in P(C_i, A, k) \mid k = min\{lcl(h), Lm(A)\}$, r_{ik} *really active*$\}$

and the schedulable rules in other clusters are taken in $\Pi(C_j, A, lcl(h))$, for every cluster $C_j \in CS(A)$.

4.2 ELPE States and Two Algorithms for the Generation of PG Control Rules

ELPE states provide an alternative functioning for the PGP, where the parallelism of the analyses can be realized in an extended scheduling mechanism through the control rules. Their use does not increase the recognition power, but there are some practical advantages:

- As described in the following two Sections it is possible to generate a set of PG control rules from a grammar specified in a formal way; this can work with regular right-linear and context-free grammars;
- The generation mechanism is particularly well suited for regular right-linear grammars, where all the work is actually made in the ELPE states, even though it can be also used for context-free grammars;
- When working on a PG generated from a regular right-linear grammar the PGP works as a parallel non-deterministic finite state automaton;
- Both the cases of generation are two examples of how the control can be worked out in the PGP without affecting the writing of the grammar rules. Once the control rules are generated, it is possible to insert all the necessary operations dealing with the syntactic/semantic analyses in the grammar rules since the control matters are already solved.

4.2.1 Generation of PG Control Rules from Regular Right-Linear Grammar

In this Section it is shown how it is possible to generate a set of PG control rules corresponding to a regular grammar, in such a way the PGP works as a parallel non-deterministic finite state machine. In particular, the kind of regular grammars considered for the generation are the right-linear grammars; this is not a restriction as any left-linear grammar can be transformed into an equivalent right-linear grammar, and viceversa.

Definition 4.16 *G is a right-linear grammar with terminal symbols set V_T, non-terminal symbols set V_N, and start symbol S, denoted by $G = (V_T, V_N, P, S)$, if each rule in the set P has one of the forms:*

$A \rightarrow \alpha,\ A \in V_N, \alpha \in V_T^+$
$A \rightarrow \alpha\, B,\ A, B \in V_N, \alpha \in V_T^+$

The process is in two main steps: construction of a finite state automaton corresponding to the right-linear grammar; and generation of the PG control rules from the transitions of the finite state automaton. The finite state automaton constructed from the regular grammar is not the standard corresponding automaton for it, but a different one that has some restrictions on the kinds of transitions. This automaton is called Rule-State Machine (RSM) as its states will identify rules of the PG.

Definition 4.17 *M is a* **Rule-State Machine** *with rule-state set R, alphabet Σ, transition set T, initial rule-state set I, final rule-state set F, denoted by $M = (R, \Sigma, T, I, F)$ if each transition rule in T has the form:*

- $(r_\alpha\ \beta \rightarrow r_\beta)$, $r_\alpha, r_\beta \in R$, $\alpha, \beta \in \Sigma^+$
- $(\alpha \rightarrow r_\alpha)$, *for every* $r_\alpha \in I$, $\alpha \in \Sigma^+$

such that:

- *if* $(r_\alpha\ \beta \rightarrow r_\beta) \in T$ *then* $\beta \neq \epsilon$*; that is* $\epsilon \notin \Sigma$*;*
- *if* $(r_{\alpha_1}\ \beta_1 \rightarrow r_\beta), (r_{\alpha_2}\ \beta_2 \rightarrow r_\beta) \in T$ *then* $\beta = \beta_1 = \beta_2$*; that is, if a set of transitions to a rule-state can be made from many sources then the string that allows the transitions is the same for each rule.*

A configuration or instantaneous description of a RSM M is a string of the form $[c\gamma\alpha]r_\alpha\beta\delta$, where $c \in \Sigma$, $r_\alpha \in R$, $[c\gamma\alpha]$ is the input string recognized so far, $\beta\delta$ is the remaining input string. If there is a transition rule $(r_\alpha\ \beta \rightarrow r_\beta) \in T$ then this rule can be applied to the configuration:

$[c\gamma\alpha]r_\alpha\beta\delta \rightarrow [c\gamma\alpha\beta]r_\beta\delta$

Given an input string over Σ as a regular expression $c\alpha$, the initial configuration for $c\alpha$ is $[c]r_c\alpha$, being r_c the initial rule-state. A given regular expression $c\alpha$ is accepted when a final configuration $[c\alpha]r_\varphi$ is derived for some $r_\varphi \in F$, $c\alpha = c\alpha_1\varphi$. The language accepted by M, denoted $L(M)$, is the set of input strings accepted by M:

$L(M) = \{w \in \Sigma^+ \mid w = c\alpha, [c]r_c\alpha \Rightarrow^*_M [c\alpha]r_\varphi \text{ for some } r_\varphi \in F\}$

The algorithm for generating a set of PG control rules from a right-linear grammar is in two steps: construction of a RSM M from a right-linear grammar G; generation of the PG control rules from M.

- **First Step** :

  From the right-linear grammar $G = (V_T, V_N, P, S)$ construct the RSM $M = (R, \Sigma, T, I, F)$ as follows:

  1. For each $(S \rightarrow \alpha) \in P$ then $(\alpha \rightarrow r_\alpha) \in T$, $r_\alpha \in F$, $r_\alpha \in I$;
  2. For each $(S \rightarrow \alpha\ A) \in P$ then $r_\alpha \in F$, $r_\alpha \in RuleSet(S)$;
  3. For each $(B \rightarrow \alpha) \in P$, $B \neq S$, then $(\alpha \rightarrow r_\alpha) \in T$, $r_\alpha \in I$, $r_\alpha \in RuleSet(B)$;
  4. For each $(B \rightarrow \alpha\ A) \in P$, $B \neq S$, then $r_\alpha \in RuleSet(B)$;
  5. For each $(B \rightarrow \alpha\ A) \in P$, such that $r_{\beta_1}, \ldots, r_{\beta_k} \in RuleSet(A)$ then $(r_{\beta_1}\ \alpha \rightarrow r_\alpha), \ldots, (r_{\beta_k}\ \alpha \rightarrow r_\alpha) \in T$.

- **Second Step** :

  From the RSM M derive a set of PG rules as follows:

  - For each $r_\alpha \in R - F$ then a rule r_α exists such that $RED(r_\alpha) = (empty\ (\alpha))$;
  - For each $r_\alpha \in R$ such that $(r_\alpha\,\alpha \rightarrow r_\alpha), (r_\alpha\beta_i \rightarrow r_{\beta_i}) \in T, \alpha \neq \beta_i$, for $i = 1, \ldots, k$, then (ELPE $r_\alpha r_{\beta_1}$),...,(ELPE $r_\alpha r_{\beta_k}$)$\in CTA(r_\alpha)$;

- For each $r_\alpha \in R$ such that $(r_\alpha\ \beta_i \rightarrow r_{\beta_i}) \in T, \alpha \neq \beta_i$, for $i = 1, \ldots, k$, then (ELPE r_{β_i})$\in CTA(r_\alpha)$, $i = 1, \ldots, k$;
- For each $r_\alpha \in I$ such that $(\alpha \rightarrow r_\alpha) \in T$, $r_\alpha \in F$, and $(r_\alpha\ \beta_i \rightarrow r_{\beta_i}) \in T$, $i = 1, \ldots, k$, then the rules r_{B_α}, r_α exist such that: $RED(r_{B_\alpha}) = (S(\alpha))$, $ST(r_{B_\alpha}) = inactive$, $RED(r_\alpha) = (empty(\alpha))$, $ST(r_\alpha) = active$, and (RA r_{B_α}),(ELPE r_{β_i})$\in CA(SSET(r_\alpha))$, for $i = 1, \ldots, k$;
- For each $r_\alpha \in I$ such that $(\alpha \rightarrow r_\alpha) \in T$, $r_\alpha \in F$, and there are not outgoing arcs from r_α, that is $(r_\alpha\ \beta \rightarrow r_\beta) \notin T$, for any β, then $RED(r_\alpha) = (S(\alpha))$, and $ST(r_\alpha) = active$;
- For each $r_\alpha \in R$ such that $(r_\alpha\ \varphi \rightarrow r_\varphi) \in T$, $r_\varphi \in F$, and there are not outgoing arcs from r_φ, that is $(r_\varphi\ \beta \rightarrow r_\beta) \notin T$ for any β, then a rule r_φ exists such that $RED(r_\varphi) = (S(\varphi))$, and $ST(r_\varphi) = inactive$;
- For each $r_\alpha \in R$ such that $(r_\alpha\ \varphi \rightarrow r_\varphi) \in T$, $r_\varphi \in F$, and at least a transition $(r_\varphi\ \beta \rightarrow r_\beta) \in T$ exists then a rule r_{B_φ} exists such that $RED(r_{B_\varphi}) = (S(\varphi))$, $ST(r_{B_\varphi}) = inactive$, and (RA r_{B_φ})$\in CA(TFSET(r_\varphi))$.

4.2.2 Generation of PG Control Rules from Context-Free Grammar

In this Section it is shown a possible method for the generation of a PG from a context-free grammar. The process is similar to that described in Section 4.2.1, the only difference is that a RSM is not sufficient and it is necessary to generate a transition graph, very similar in the structure to the RSM but without a recognition device for G (it is not as simple as for the RSM to give such a device, and it is useless for the current discussion), that can have more kinds of transitions, and whose states must be denoted so that they carry information about the rule-state and the possibly non-terminal node constructed. The transition graph then gives all information needed in specifying the control information to be put in the control rules of the PG.

Definition 4.18 *G is a context-free grammar with terminal symbols set V_T, non-terminal symbols set V_N, and start symbol $S \in V_N$, denoted by $G = (V_T, V_N, P, S)$, if each rule in the set P is of the form:*

$A \rightarrow \alpha$, $A \in V_N$; $\alpha \in (V_T \cup V_N)^+$

The rules in P are numbered, thus if $|G| = |P|$, then:

$P = \{r_1, \ldots, r_{|G|}\}$ *and every rule in P is denoted as $r_i : (A \rightarrow \alpha)$, for $i = 1, \ldots, |G|$.*

Definition 4.19 *$\mathcal{T}$ is a* **transition graph** *with rule-state set R, alphabet Σ, transition set T, initial set I, final rule-state set F, denoted by $\mathcal{T} = (R, \Sigma, T, I, F)$, if:*

- $\Sigma = (V_T \cup V_N)$
- $I = (\mathcal{RC} \cap V_T)$
- $F = \{r_i \mid r_i/S \in R\}$

and each transition rule in T has one of the forms:

- $(x \rightarrow r_i/A)$, $x \in V_T$; $r_i \in P$; $A \in V_N$; $r_i/A \in R$
- $(x \rightarrow r_x)$, $x \in V_T$; $r_x \in R$
- $(r_j/B\ B \rightarrow r_i/A)$, $r_i, r_j \in P$; $A, B \in V_N$; $r_j/B, r_i/A \in R$
- $(r_j/B\ c \rightarrow r_i/A)$, $r_i, r_j \in P$; $A, B \in V_N$; $c \in \Sigma$; $r_i/A, r_j/B \in R$
- $(r_j/B\ c \rightarrow r_c)$, $r_j \in P$; $B \in V_N$; $c \in \Sigma$; $r_j/B, r_c \in R$
- $(r_d\ c \rightarrow r_i/A)$, $r_i \in P$; $c, d \in \Sigma$; $A \in V_N$; $r_d, r_i/A \in R$
- $(r_d\ c \rightarrow r_c)$, $c, d \in \Sigma$; $r_d, r_c \in R$

The algorithm for the generation of a PG from a context-free grammar is again in two steps: construction of a transition graph $\mathcal{T}$ from the context-free grammar G; generation of the PG control rules from the transition set T of $\mathcal{T}$.

- **First Step** :

  From a context-free grammar $G = (V_T, V_N, P, S)$ construct the transition set T for the transition graph $\mathcal{T} = (R, \Sigma, T, I, F)$ as follows:

  1. For each $r_i : (A \rightarrow x) \in P$, $x \in V_T$, then $(x \rightarrow r_i/A) \in T$;
  2. For each $r_i : (A \rightarrow B) \in P$, $B \in V_N$, build $RuleSet(B) = \{r_j \mid r_j : (B \rightarrow \gamma) \in P\}$; then for each $r_j \in RuleSet(B)$ $(r_j/B\ B \rightarrow r_i/A) \in T$;
  3. For each $r_i : (A \rightarrow c_1 \ldots c_k\ x) \in P$, $c_1, \ldots, c_k \in (V_T \cup V_N), k \geq 1, x \in V_T$, then $(x \rightarrow r_x), (r_x\ c_k \rightarrow r_{c_k}), \ldots, (r_{c_2}\ c_1 \rightarrow r_i/A) \in T$;
  4. For each $r_i : (A \rightarrow c_1 \ldots c_k\ B) \in P$, $c_1, \ldots, c_k \in (V_T \cup V_N), k \geq 1, B \in V_N$, build $RuleSet(B) = \{r_j \mid r_j : (B \rightarrow \gamma) \in P\}$; then for each $r_j \in RuleSet(B)$ $(r_j/B\ c_k \rightarrow r_{c_k}), (r_{c_k}\ c_{k-1} \rightarrow r_{c_{k-1}}), \ldots, (r_{c_2}\ c_1 \rightarrow r_i/A) \in T$; if $k = 1$ then $(r_j/B\ c_1 \rightarrow r_i/A) \in T$.

- **Second Step** :

  From the transitions in T derive a PG as follows:

  - For each $(x \rightarrow r_i/A) \in T$ then a rule r_i exists such that $RED(r_i) = (A\,(x))$, and $ST(r_i) = active$;
  - For each $(x \rightarrow r_x) \in T$ then a rule r_x exists such that $RED(r_x) = (empty\,(x))$, and $ST(r_x) = active$;
  - For each $(r_j/B\ B \rightarrow r_i/A) \in T$ then a rule r_i exists such that $RED(r_i) = (A(B))$, $ST(r_i) = active$, and $r_i \in RULES(SSET(r_j))$;
  - For each $(r_j/B\ c \rightarrow r_i/A) \in T$ then a rule r_i exists such that $RED(r_i) = (A\,(c))$, $ST(r_i) = inactive$, and either (ELPE r_i)$\in CA(SSET(r_j))$, or (ELPE r_i)$\in CTA(r_j)$;
  - For each $(r_j/B\ c \rightarrow r_c) \in T$ then a rule r_c exists such that $RED(r_c) = (empty\,(c))$, $ST(r_c) = inactive$, and (ELPE r_c)$\in CA(SSET(r_j))$;

- For each $(r_d\, c \rightarrow r_i/A) \in T$ then a rule r_i exists such that $RED(r_i) = (A\,(c))$, $ST(r_i) = inactive$, and either (ELPE r_i)$\in$ $CA(SSET(r_d))$, or (ELPE r_i)$\in$ $CTA(r_d)$;
- For each $(r_d\, c \rightarrow r_c) \in T$ then a rule r_c exists such that $RED(r_c) = (empty\,(c))$, $ST(r_c) = inactive$, and either (ELPE r_c)$\in$ $CA(SSET(r_d))$, or (ELPE r_c)$\in$ $CTA(r_d)$.

5 Process Grammar Processor

The architecture of the parallel-parallel PGP is based on a set of independent asynchronous processes that communicate one each other through a message passing protocol. These processes are almost the same of those illustrated in Figure 1 for the serial-parallel PGP; the difference is that now the transitions are not based on stacks conditions, since there are no stacks anymore, but on the scheduling mechanism provided by the PGP and by the control rules, and the transitions are performed through the SoCs that are the messages the processes communicate one each other. Figure 9 shows the processes and their communication links. The only data sent on the links are the SoCs containing the information needed by the receiving process so that the next operation can be performed. Every operation must also perform a sort of pre-scheduling if some control augmentations are not empty; in any case, they always have to prepare a process descriptor for the scheduler in order to allow a correct scheduling. The architecture in Figure 9 shows how a unique scheduler and a unique matcher can be a bottleneck for the overall system since their task is the most expensive having to manage the access to PG, the PGS, and also the communications with the other processes. However, this is not the only possible architecture, and specific design of how the processes have to manage communications and processing also depends on the kind of parallel machine used for the implementation; an alternative architecture could have a scheduler connected to every operation so that the load due to the scheduling operations can be shared among the processes.

Definition 5.1 *A* **process descriptor** *is a 6-tuple PD=[Cluster, Rule, Node, RedSet, CSet, RuleSet] where Cluster is the current cluster used by the SoC, Rule is the rule involved; Node is a node-interpretation identifier of the node where the matching starts from (corresponds to the right-corner); RedSet is the reduction set found for Rule; CSet is a cluster set; RuleSet is a rule set. CSet, and RuleSet are used when the control augmentations are involved. There are 5 standard forms for a PD:*

- **[-,-,Node,-,-,-]***:*

  $Op =$ **scan**, *then schedule as in Definition 4.14;*

- **[Cluster,-,Node,-,-,-]***:*

  Schedule as in Definition 4.15; this kind of PD is sent to the scheduler when a reducing operation has completed its task, i.e., when either $Op =$ **reduce**, *or* $Op =$ **h** $-$ **reduce**, *or* $Op =$ **elpe** $-$ **reduce**;

- **[Cluster,Rule,Node,-,-,-]***:*

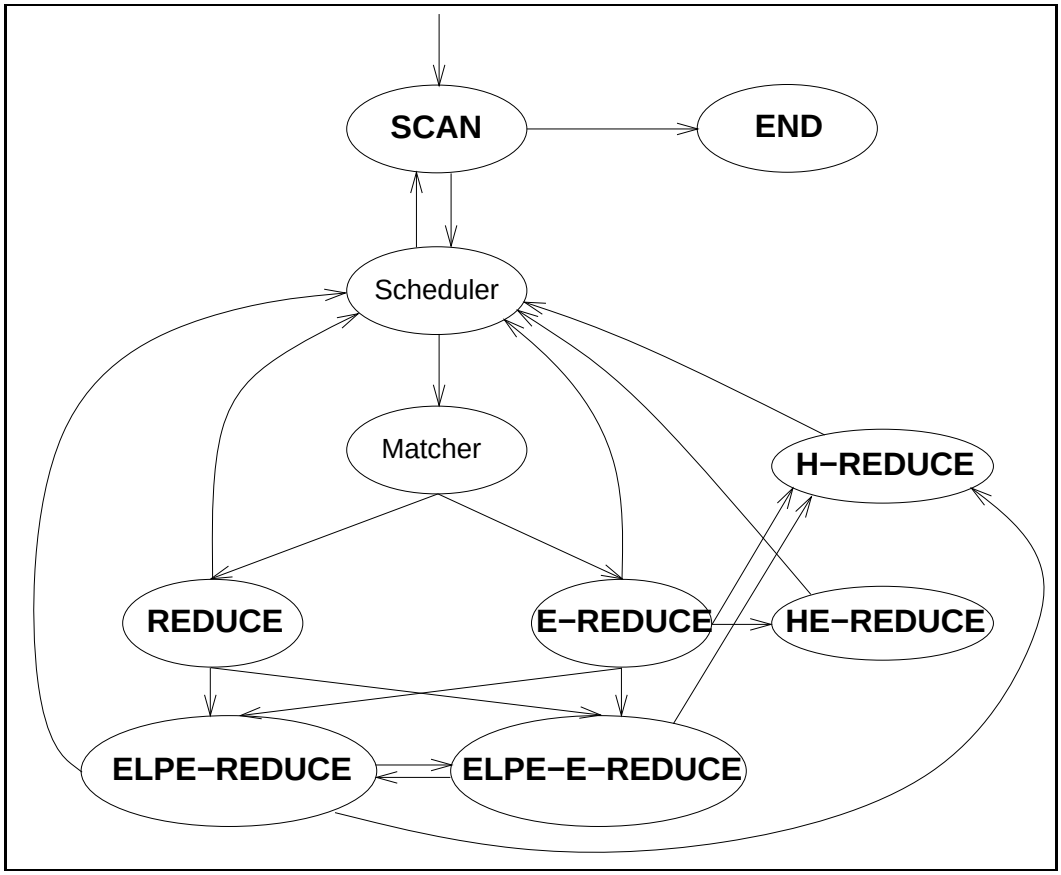


Figure 9: Processes and their connections in the parallel-parallel PGP.

Match $RED(Rule)$ starting from Node; this kind of PD is sent to the matcher from the scheduler;

- **[Cluster,Rule,Node,RedSet,-,-]***:*

  This kind of PD is sent to a reducing operation when Rule must be fired on a process environment determined by the reduction set found;

- **[Cluster,Rule,-,RedSet,-,-]***:*

  The next operation is **he-reduce** *if Rule is empty, otherwise* **h-reduce***; this kind of PD is generally due to a call to RA.*

Furthermore, when the control augmentations are involved the operations must prepare also some PDs depending on the clusters or rules specified in those slots. There are 5 possible forms when control augmentations are used:

- **[Cluster,Rule,Node,-,-,RuleSet]***:*

  In the last applied rule Rule $RULESET(MFSET(Rule))$ is not empty; then $RuleSet = RULES(MFSET(Rule))$, such that all the rules in this set must be in the cluster Cluster, and $cat(Node)$ is their right-corner;

- **[Cluster,Rule,Node,RedSet,-,RuleSet]***:*

  In the last applied rule Rule $RULES(SSET(Rule))$ is not empty; then $RuleSet = RULES(SSET(Rule))$, such that all the rules in this set must be in the cluster Cluster and they replace $\Pi(Cluster, A, \cdot)$, where A is the left-hand side of $RED(Rule)$;

- **[Cluster,Rule,Node,-,CSet,-]***:*

  In the last applied rule Rule $CLUSTERS(MFSET(Rule))$ is not empty; then $CSet = CLUSTERS(MFSET(Rule))$, such that for every cluster in this set all the rules $\Pi(C, cat(Node), \cdot)$ can be proposed;

- **[Cluster,Rule,Node,RedSet,CSet,-]***:*

  In the last applied rule Rule $CLUSTERS(SSET(Rule))$ is not empty; then $CSet = CLUSTERS(SSET(Rule))$ and replaces $CS(A)$, where A is the left-hand side of $RED(Rule)$;

- **[Cluster,Rule,Node,RedSet,-,RuleSet]***:*

  In the last applied rule Rule $RULES(TFSET(Rule))$ is not empty; then $RuleSet = RULES(TFSET(Rule))$, and all the rules in this set must have the same right-hand side of Rule; thus, the process environment for all of them is $ProcEnv(Rule)$.

Process descriptors provide information to the scheduler for scheduling when they come from the operations, and to the operations for rule application when they come from the scheduler. They are combined along with a State of Computation where some additional information allows the firing of the rule whose firing data are specified in the PD.

Definition 5.2 *The* **State of Computation** *is a 8-tuple SoC=[Op, PD, pt, pn, PGS, RealStates, Queue, ProcEnv], where: Op is the operation to be performed; PD is the process descriptor; pt, and pn are the pointers to the last terminal node scanned and the last non-terminal node reduced in the PGS; the PGS is the local parsing structure; RealStates is a table storing which rules have the real state not equal to the original state; Queue is a queue of PDs for reductions in the same context, e.g., multiple reductions through a call to RA in the same context; ProcEnv is the process environment for the rule in PD.*

For a sentence $w = w_1 \dots w_n$ the computation starts from an initial SoC:

[**scan**,[-,-,-,-,-,-],0,$n+1$,PGS,RealStates=empty,Queue=empty,ProcEnv=empty]

and terminates whenever a SoC of the following form is generated:

[**end**,[-,-,-,-,-,-],n,pn,PGS,RealStates,Queue=empty,ProcEnv=empty]

It is possible that many SoCs of this form are generated, thus providing multiple analyses for the input sentence.
Whenever a SoC is of the first form above for scanning then the operation **scan** must receive that SoC, and then has simply to increase the terminal node pointer, to put the node interpretation identifier in the PD, and send the SoC to the scheduler[2]:

```
scan:
loop
    receive SoC;
        /* SoC=[scan,[-,-,-,-,-,-],pt,pn,PGS,...,Queue=empty,ProcEnv=empty]*/
    if pt=n
        then  send-to-end([end,[-,-,-,-,-,-],pt,pn,PGS,...])
        else  send-to-scheduler([scan,[-,-,pt+1,-,-,-],pt+1,pn,PGS,...]);
until job-done;
```

Let suppose now that the scheduler and the matcher have arranged the data needed for firing a standard rule in the **reduce** operation. Then the scheduler can send a SoC of the following form:

[**reduce**,[Cluster,Rule,Node,RedSet,-,-],pt,pn,PGS,...,ProcEnv]

to the **reduce** operation that would work as follows:

[2]The following code is not intended to describe in a full detail the task of the operations, since aspects of how they should be loaded and started, or how they should stop, are implementation details that are not faced here.

```
reduce:
loop
   receive SoC; /*SoC=[reduce,[Cluster,Rule,Node,RedSet,-,-],pt,pn,PGS,...,ProcEnv]*/
   if RedSet≠ Ø
      then if gr-apply(TEST(Rule))
                then  add-subtree(pn,ProcEnv);
                      gr-apply(ACT(Rule));
                      cr-apply(SSET(Rule));
                else  gr-apply(RATF(Rule));
                      cr-apply(TFSET(Rule));
      else gr-apply(RAMF(Rule));
           cr-apply(MFSET(Rule));
until job-done;
```

Here, and in all other operations, the function `cr-apply` must deal with:

- The application of control actions, and consequently the handling of the scheduling functions as ELPE or RA;
- The creation of new PDs based on the other control augmentation slots as in Definition 5.1, and their transmission in new SoCs to the scheduler.

In the case the scheduler has set up the data needed for the firing of an empty rule in the operation **e-reduce**, a SoC of the following form should be sent to it:

[**e-reduce**,[Cluster,Rule,Node,RedSet,-,-],pt,pn,PGS,...,ProcEnv]

and the operation should work as follows:

```
e-reduce:
loop
   receive SoC; /*SoC=[e-reduce,[Cluster,Rule,Node,RedSet,-,-],pt,pn,PGS,...,ProcEnv]*/
   if RedSet≠ Ø
      then if cr-apply(CTA(Rule))
              then if gr-apply(TEST(Rule))
                      then  gr-apply(ACT(Rule));
                            cr-apply(SSET(Rule));
                      else  gr-apply(RATF(Rule));
                            cr-apply(TFSET(Rule));
              else cr-apply(TFSET(Rule));
      else gr-apply(RAMF(Rule));
           cr-apply(MFSET(Rule));
until job-done;
```

The other operations work in an analogous way, with some further processing in **elpe-reduce**, and **elpe-e-reduce** for the handling of the extension of the process environment.

6 Parsing Tools

In the description of the PGP given so far some parsing tools have been already introduced and it has been explained how they interact with the parsing process and the parsing control. Some of them can be integrated further with other tools that are described in this Section: the pre-scheduling mechanism; the use of feature structures; the message passing mechanism.

6.1 Pre-Scheduling

The mechanism of pre-scheduling concerns with the preference for the application of some rules. Three kinds of pre-scheduling can be defined, but in general it works as follows: when a rule r pre-schedules a rule r' then the parser is advised that r' is pending for application, and, when the conditions for firing it occur, it must be scheduled. Let consider an example for the language $a^n b^n c^n$. If the substring a^n is recognized by the rules:

$r_{a1} : (A \leftarrow a)$
$r_{a2} : (A \leftarrow Aa)$

then it could be possible to have a pre-scheduling function called both in r_{a1} and r_{a2} such that the rules for recognizing b^n would be pre-scheduled:

$r_{b1} : (B \leftarrow b)$
$r_{b2} : (B \leftarrow Bb)$

If the pre-scheduling is for just one application of every rule then the number of rules that r_{a1} and r_{a2} pre-schedule is just equal to the number of a's scanned in the string. For completing the parsing the same must be then done by r_{b1} and r_{b2}.
It is clear that there must be a $1-1$ correspondence between the pre-scheduling and the pre-scheduled rules to have a correct parse working for the language above. What has been left unspecified so far is how the application order of the rules is handled and which strategy of pre-scheduling can be implemented.

6.1.1 Pre-Scheduling Strategies

It is possible to define various strategies of pre-scheduling, where for strategy it is meant how one or more rules designated for scheduling will be applied, that is, the order and the number of times. Three different ways of pre-scheduling are defined that cause three different ways of scheduling and application:

- Stack-order pre-scheduling: the rules are pre-scheduled in stack order and are applied once;
- Queue-order pre-scheduling: the rules are pre-scheduled in queue order and are applied once;
- Unordered pre-scheduling: the rules are pre-scheduled and applied either once or as many times as possible not following an order of activation.

The information about the pre-scheduled rules in the current SoC should be inherited by the future SoCs generated after the pre-scheduling rule; thus, if pre-scheduling is required, the SoC structure should be extended with the appropriate data structures for supporting the different pre-scheduling strategies. A common constraint for whatever pre-scheduling strategy is that the rules involved must be always really-inactive to avoid automatic scheduling by the scheduler. Let now examine the different strategies.

- **Stack-Order Pre-Scheduling** :

  Stack-order pre-scheduling is generated by the function sPS as (sPS $r_1 \ldots r_k$), where the rules are considered in stack order for application, that is, starting from the last listed one: $r_k, r_{k-1}, \ldots, r_1$. Whenever a rule r_i is scheduled the next schedulable rule becomes r_{i-1}, so just one application of the rules must be performed; a strong condition on the success of the parsing is that the stack of the pre-scheduled rules must be empty when parsing ends;

- **Queue-Order Pre-Scheduling** :

  Queue-order pre-scheduling is generated by the function qPS as (qPS $r_1 \ldots r_k$), where the rules are considered in queue order for application, that is, starting from the first listed one: $r_1, r_2, \ldots, r_k$. Whenever a rule r_i is scheduled the next schedulable rule becomes r_{i+1}, so just one application of the rules must be performed; a strong condition on the success of the parsing is that the queue of the pre-scheduled rules must be empty when parsing ends;

- **Unordered Pre-Scheduling** :

  Unordered pre-scheduling is generated by the function uPS as (uPS $r_1 \ldots r_k$), where the rules are scheduled in a not *a priori* defined order, as it must depend on the schedulable events that occur during parsing. Thus, the order of application is that whenever a rule r_i is scheduled there is not a next candidate, but the whole set of rules is always the set of the candidates. If a counter is associated to a rule than the scheduling can be pre-determined for a limited number of times.

Example 6.1 *Let take again the language $a^n b^n c^n$ for this example, and consider some more constraints for parsing a string in this language: using the pre-scheduling the parser should reduce one non-terminal node for every substring a^n, b^n, c^n. The grammar below does the job in the following way:*

- *The rule $r1$ determines the border between the a's and the b's; then it starts an extension firing ra, and fires $r2$ at the same time;*

- *The extension started with ra ends immediately if there is just one a, otherwise it continues through $ra1$, that extends the coverage of the node A reduced by ra, and q-pre-schedules $rb1$;*

- *$r2$, fired by $r1$ on the first b, reduces the node B, and q-pre-schedules $r3$;*

- *$rb1$ is fired for every b in the string if the correct number has been q-pre-scheduled by $ra1$, extends the coverage of node B reduced by $r2$, and q-pre-schedules $rc1$;*

- *r3 is fired on the first c and reduces the node C;*
- *rc1 is fired for every c in the string if the correct number has been q-pre-scheduled by rb1, and extends the coverage of the node C reduced by r3;*
- *rs is fired when its process environment covers all the sentence and the pre-scheduling queue is empty.*

The previous functioning for a correct string causes a sequence of activations of the rules as given by the following string:

$r1\ ra\ ra1^{n-1}\ r2\ rb1^{n-1}\ r3\ rc1^{n-1}\ rs$

For any string not in the language the parsing fails either because the pre-scheduling queue is not empty or because the sentence cannot be covered completely.

(**dgr**
  :RNAME *r1*
  :RED *(empty (a b)))*
(**dcr**
  :RNAME *r1*
  :ST *active*
  :SSET *(:CA ((RA ra r2))))*

(**dgr**
  :RNAME *ra*
  :RED *(A (a)))*
(**dcr**
  :RNAME *ra*
  :ST *inactive*
  :SSET *(:CA ((ELPE ra1))))*

(**dgr**
  :RNAME *ra1*
  :RED *(empty (a))) ;* $\epsilon \leftarrow a[A]$
(**dcr**
  :RNAME *ra1*
  :ST *inactive*
  :SSET *(:CA ((ASR A a) (ELPE ra1) (qPS rb1))))*

(**dgr**
  :RNAME *r2*
  :RED *(B (b)))*
(**dcr**
  :RNAME *r2*
  :ST *inactive*
  :SSET *(:CA ((qPS r3))))*

```
(dgr
   :RNAME    rb1
   :RED      (empty (B b)))
(dcr
   :RNAME    rb1
   :ST       inactive
   :SSET     (:CA ((ASR B b) (qPS rc1))))

(dgr
   :RNAME    r3
   :RED      (C (c)))
(dcr
   :RNAME    r3
   :ST       inactive)

(dgr
   :RNAME    rc1
   :RED      (empty (C c)))
(dcr
   :RNAME    rc1
   :ST       inactive
   :SSET     (:CA ((ASR C c))))

(dgr
   :RNAME    rs
   :RED      (S (A B C)))
(dcr
   :RNAME    rs
   :ST       active
   :CTA      ((covered-sentence-p)))
```

It is interesting to note that cross dependencies occurring in sentences as:

John Mary Paul are a boy a girl a dog, respectively

can be easily solved using this method.

6.2 Feature Structures, and Message Passing

Feature structures (FS) are the memory of the process environment. Each node created and available to a process through the process environment has a FS where to store intermediate and final data of a parsing. There is a set of functions that can be used in the actions for handling this set of data; they are not described here, but a description of them is in [8]. FSs are defined as a set of attribute-value pairs and are built bottom-up in a compositional way, but it can also be possible to allow a long-distance connection (in a non-compositional way)

between rules working in different sides of a sentence and transmitting a set of a FS from a rule to another one. This mechanism, named **message passing**, makes possible to transmit a message, consisting of a feature structure or a set of feature structures, from the process environment of a fired rule, to another rule that will be fired next. For a more detailed discussion about message passing see [5]. Therefore, the message passing is asynchronous, and it can be used also whenever a rule pre-schedules another one, when a FS to be passed to the pre-scheduled rule is specified. The set of FS functions also allows an easy way for implementing Lexical-Functional Grammars and the construction of unification algorithms that can be used within the grammar rule augmentations.

References

[1] Aho, A., V., and Ullman, J., D. (1972). *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing.* Prentice Hall. Englewood Cliffs, NJ.

[2] Dassow, J., and Păun, G. (1989). *Regulated Rewriting in Formal Language Theory.* EATCS. Vol. 18. Springer-Verlag, Berlin.

[3] Grishman, R. (1976). A Survey of Syntactic Analysis Procedures for Natural Language. *American Journal of Computational Linguistics.* Microfiche 47. 2-96.

[4] Kay, M. (1980). Algorithm Schemata and Data Structures in Syntactic Processing. In Grosz, B., J., et al. (Eds.), *Readings in Natural Language Processing.* Morgan Kaufmann, Los Altos, CA. 35-70. Also CSL-80-12, Xerox PARC, Palo Alto, CA.

[5] Marino, M. (1989). A Framework for the Development of Natural Language Grammars. In *Proceedings of First International Workshop on Parsing Technologies.* CMU, Pittsburgh, PA. 350-360.

[6] Marino, M. (1990). Bottom-Up Parsing Extending Context-Freeness in a Process Grammar Processor. In *Proceedings of 28th Annual Meeting of the ACL.* Pittsburgh, PA.

[7] Meersman, R., and Rozenberg, G. (1978). Cooperating Grammar Systems. In Winkowski, J. (Ed.), *Proceedings of 7th Mathematical Foundations of Computer Science.* 364-373.

[8] *PGDE: Process Grammar Development Environment User Manual.* AITech s.n.c., Pisa, Italy.

[9] Shann, P. (1989). The Selection of a Parsing Strategy for an On-Line Machine Translation System in a Sublanguage Domain. A New Practical Comparison. In *Proceedings of First International Workshop on Parsing Technologies.* CMU, Pittsburgh, PA. 264-276.

[10] Slocum, J. (1981). A Practical Comparison of Parsing Strategies. In *Proceedings of 19th Annual Meeting of the ACL.* Stanford, CA. 1-6.

[11] Wirén, M. (1987). A Comparison of Rule-Invocation Strategies in Context-Free Chart parsing. In *Proceedings of the 3rd Conference of the European Chapter of the ACL.* Copenhagen, Denmark. 226-233.