Triangle: Language, Literature, Computation, n. 8, 2012
Publicacions Universitat Rovira i Virgili · ISSN: 2013-939X
https://revistes.urv.cat/index.php/triangle

Owicki-Gries Theory: A Possible Way of Relating Grammar Systems to Concurrent Programs

María Adela Grando

Research Group on Mathematical Linguistics
Rovira i Virgili University
Tarragona, Spain
E-mail: mariaadela.grando@estudiants.urv.es

Summary. The aim of this paper is to show how grammar systems and concurrent programs might be viewed as related models for distributed and cooperating computation. We argue that it is possible to translate a grammar system into a concurrent program, where the Owicki-Gries theory and other tools available in the programming framework can be used. The converse translation is also possible and this turns out to be useful when we are looking for a grammar system that can generate a given language.

In order to show this we use tools from concurrent programming theory to prove that $L_{cd} = \{a^n b^m c^n d^m \mid n, m \geq 1\}$ can be generated by a non-returning Parallel Communicating grammar system with three regular components. We show that this strategy can be helpful in the construction of grammar systems that generate strings in less time and more efficiently. We also discuss the absence of strategies in the concurrent programming theory to prove that L_{cd} can be generated by any Parallel Communicating grammar system with two regular components.

1 Introduction

At the beginning of computation theory, classic computing devices were centralized: that is, the computation was accomplished by one central processor. But in contemporary computer science distributed computing systems that consist of multiple communicating processors play a major role because they have various advantages: efficiency, fault tolerance, scalability in the relation between price and performance, etc.

Since 1960, when the concept of *concurrent programming* [6] was introduced, a huge variety of topics related to parallelism and concurrency have been defined and investigated: for example, operating systems, machine architectures, communication networks, circuit design, protocols for communication and synchronization, distributed algorithms, logics for concurrency, automatic verification and model checking. The same trend has been observed in classic formal language and automata theory as well. At first, grammars and automata modelled classic computing devices of one agent or processor, so a language was generated by one grammar or recognized by one automaton. Inspired by different models of distributed systems in Artificial Intelligence, *grammar systems theory* [4] has been developed as a grammatical theory for distributed and parallel computation. More recently, similar approaches have been reported for systems of automata [12].

In the concurrent programming framework *Owicki-Gries theory* [13], the first complete programming logic for the formal development of concurrent programs and other programming strategies was developed to help programmers analyse and design multiprograms. We argue that grammar system theory can benefit from these tools. For example: *given a grammar system one can prove that it generates a specific language* by direct reasoning or one can translate the grammar system into a multiprogram and prove the same statement by some strategies of programming developed in the Owicki-Gries theory. We exemplify this with the language $\{a^n b^n c^n \mid n \geq 0\}$. Furthermore, we propose another approach to solve problems of the following type: *given a language specification find a grammar system that generates the given language*. The strategy widely used so far is as follows: first, propose a grammar system and then prove by means of language theory that the proposed grammar system does indeed generate the given language. We give three examples of how the Owicki-Gries logic of programming could help us to *simultaneously* obtain a grammar system that generates the given language and the proof that it really generates it. This new approach might be of great benefit for the grammar

systems theory. The strategy consists of translating the problem of finding grammar system Γ of a certain type that generates a language L, into the problem of finding a multiprogram $\mathcal{P}$. $\mathcal{P}$ will have as many programs $Prog_i$ as the grammar system Γ has grammars and will have to be correct with respect to the specification:

$$\{(w_1 = S_1) \wedge (w_2 = S_2) \wedge ... \wedge (w_n = S_n) \wedge n \geq 1\}\ \mathcal{P}\ \{w_1 \in L\}.$$

Then this multiprogram will be translated back into the grammar system Γ, the whole behavior of Γ being similar to that of $\mathcal{P}$. Actually, the language generated by Γ is included in L, but for the examples we present here equality is reached, as detailed reasonings prove.

Here we show how to apply this strategy for a well-known non-context-free language, namely $L_{cd} = \{a^n b^m c^n d^m \mid n, m \geq 1\}$. In [2] it was proved that L_{cd} can be generated by a Nonreturning Centralized Parallel Communicating grammar system with four context free components ($L_{cd} \in CPC_4(CF)$). In [8] we improved this result showing that $L_{cd} \in NPC_3(REG)$, based on a similar strategy. Here we show the proof of this result. We also exemplify with the language $\{xcxc \mid x \in \{a, b\}^*\}$ the use of the Owicki-Gries strategy to get more efficient grammar systems when combined with some other programming techniques used to improve parallelism.

Finally we show how the concurrent programming framework can benefit from grammar system theory to get negative results; in the first one we have no strategy to deal with negative results of the type: *a given language cannot be generated by any grammar system of a specified type.* This kind of problems has to be analyzed in the grammar system framework, with the tools available there.

2 Grammar System Theory: Models

2.1 Cooperative Distributed (CD) grammar systems

Grammar System theory started in 1988 by introducing CD grammar systems for modelling syntactic aspects of the blackboard model of problem solving [3]. It is a finite set of (usually generative) grammars which cooperate in deriving words of a common language. At any moment in time there is exactly one sentential form in generation. The component grammars generate the string by turns, under a cooperation protocol, called the derivation mode. In this

model the cooperating grammars represent independent cooperating problem solving agents which jointly solve a problem. They modify the contents of a global database, called the blackboard, which is used for storing information on the problem solving process. In blackboard architectures the agents communicate with each other only through the blackboard: that is, there is no direct communication among them.

We do not give here the formal definition of this model, but the reader is referred to [4] for all the formal concepts related to all grammar systems models mentioned in this paper.

We fix the notation for the class of languages generated by homogeneous Cooperative Distributed grammar systems. We denote them as $CD_n(f)$ where $n \in \mathbf{N}$ is the maximun number of grammar components with context free productions and $f \in \{t, *\} \cup \{\leq k, = k, \geq k \mid k \geq 1\}$ is the mode of derivation.

2.2 Networks of language processors

Networks of language processors form an essential area in the theory of grammar systems. Language processors, that is grammars or other language determining devices, are located in nodes of a network (a virtual graph). Each processor works on its own sentential form (on its own collection of sentential forms) and informs the others about its activity by communicating strings which can be data and/or programs. Rewriting and communication take place alternately, and the system functions (usually) in a synchronized manner.

The difference between CD grammar systems and these architectures is that while in the first case the grammars generate a common string, in the second case each of them operates on its own string. There are several important models in the area, of which we are interested in two: Parallel Communicating (PC) grammar systems and Parallel Communicating grammar systems with Communication by Command (CCPC).

Parallel communicating grammar systems were introduced in [15] as a grammatical representation of the so-called "classroom model" of problem solving, which is a modification of the blackboard model.

We denote by $PC_n(Y)$ the class of languages generated by non-centralized Parallel Communicating grammar systems with at most n components, each component with productions of type Y, where: $n \in \mathbf{N}$ and $Y \in \{FIN, REG, CF, CS, RE\}$. When the PC grammar system is centralized, non-returning non-centralized and non-returning centralized the prefixes C, N and NC, respectively, are added.

We denote by $CCPC_n(Y)$ the class of languages generated by Parallel Communicating grammar systems with Communication by Command with at most n components, each component with productions of type Y, where $n \in \mathbf{N}$ and $Y \in \{FIN, REG, CF, CS, RE\}$.

3 Programming

3.1 Sequential programming

A sequential program consists of a number of declarations and a sequence of instructions or actions. The actions take place one after another. That is, an action does not begin until the preceding one has ended. Because a sequential program has a sequence of actions we consider a program as a transformer of states or predicates [9], where a state $\{P\}$ describes the relationships between the variables of the systems and their values by the predicate P . Each action $\mathcal{S}$ transforms the current state of the system, called precondition of $\mathcal{S}$, to the state $\{Q\}$ which is called postcondition.

A *Hoare triple* is a sequence $\{P\}\,\mathcal{S}\,\{Q\}$, where:

– $\mathcal{S}$ is an action or instruction,
– $\{P\}$ is a state representing the precondition of $\mathcal{S}$,
– $\{Q\}$ is a state representing the postcondition of $\mathcal{S}$.

Its operational interpretation is as follows: $\{P\}\,\mathcal{S}\,\{Q\}$ is a correct Hoare triple if and only if it is true that each terminating execution of $\mathcal{S}$ that starts from a state satisfying P is guaranteed to end up in a state satisfying Q. More precisely, if $\{P\}\,\mathcal{S}\,\{Q\}$ holds and $\mathcal{S}$ starts in a state satisfying P, we can be sure that $\mathcal{S}$ either terminates in a state satisfying Q or does not terminate at all. Consequently, a program ought to be annotated in such a way that each action carries a precondition. In other words, from a logical perspective a sequential program may be viewed as a sequence of Hoare triples.

We can now formulate the concept of *local correctness* of a predicate Q in a program. We distinguish two cases:

– If Q is the initial predicate of the program, it is locally correct whenever it is implied by the precondition of the program as a whole. We may also say that Q satisfies the hypothesis of the problem which is to be solved.

– If Q is preceded by $\{P\}\,\mathcal{S}$, i.e. by atomic action $\mathcal{S}$ with precondition P, it is locally correct whenever $\{P\}\,\mathcal{S}\,\{Q\}$ is a correct Hoare-triple.

A sequential program is *partially locally correct* if all its predicates are locally correct and the last predicate satisfies the requirements of the problem solved, provided that it halts. A sequential program is *totally locally correct* if it is partially correct and always halts.

3.2 Concurrent programming

Concurrent execution or multiprogramming means that various sequential programs run simultaneously. Actions change the state of the multiprogram, so the critical question now is what happens if two overlapping actions change the same state of the multiprogram in a conflicting manner.

Now we are ready to formulate what we call the core of the **Owicki-Gries theory**. We consider a multiprogram annotated in such a way that the annotation provides a precondition for the multiprogram as a whole and a precondition for each action in each individual program. Then, by Owicki and Gries, *a multiprogram is correct* whenever each individual predicate is correct, i.e.:

– *locally correct* as described above and

– *globally correct*: a predicate Q in a multiprogram $\mathcal{M}$ is globally correct whenever for each $\{P\}\,\mathcal{S}$, i.e. for each action $\mathcal{S}$ with precondition P, taken from a program of $\mathcal{M}$, $\{P \vee Q\}\,\mathcal{S}\,\{Q\}$ is a correct Hoare-triple.

4 How to Relate Grammar Systems with Programming

4.1 How can grammar systems benefit from programming?

In this section we exemplify possible ways in which grammar systems can benefit from the Owicki-Gries theory and from some strategies of proof used in the formal analysis of concurrent programs.

Example 1. We introduce the grammar system $\Gamma_1 \in CD_2(=2)$ defined in this way:

$$\Gamma_1 = (\{a,b,c\}, (\{S,A,A',B,B'\}, \emptyset, P_1, =2),\\ (\{S,A,A',B,B'\}, \{a,b,c\}, P_2, =2), S)$$

where:

$P_1 = \{S \to S, S \to AB, A' \to A, B' \to B\}$,

$P_2 = \{A \to aA'b, B \to cB', A \to ab, B \to c\}$.

We transcribe the proof taken from [5] that all the derivations in Γ_1 are of the form:

$$S \underset{P_1}{\Longrightarrow} S \underset{P_1}{\Longrightarrow} AB \underset{P_2}{\Longrightarrow} aA'bB \underset{P_2}{\Longrightarrow} aA'bcB' \underset{P_2}{\Longrightarrow} aAbcB \underset{P_2}{\Longrightarrow} \ldots\ldots\ldots$$

$$\ldots\ldots\ldots \underset{P_2}{\Longrightarrow} a^n A'b^n c^n B' \underset{P_2}{\Longrightarrow} a^n Ab^n c^n B \underset{P_2}{\Longrightarrow} a^{n+1} b^{n+1} c^{n+1}$$

for some $n \geq 0$. Hence $L_{=2}(\Gamma) = \{a^n b^n c^n \mid n \geq 0\}$

To show that the previous sequence of derivation is correct, and that it is the only possible sequence of derivation, we analyze all possible cases by applying the technique of analysis by cases.

We have to start from S. Only P_1 can be used. Applying the rule $S \to S$ twice changes nothing, so eventually we shall perform the step

$$S \underset{P_1}{\Longrightarrow} S \underset{P_1}{\Longrightarrow} AB$$

From now on, S will never appear again. Only P_2 can be applied to AB. If we use the nonterminal rules, we get: $AB \underset{P_2}{\Longrightarrow} aA'bB \underset{P_2}{\Longrightarrow} aA'bcB'$

In general, from a string of the form $a^i Ab^j c^k B$ (initially we have $i = j = k = 0$), we can obtain $a^i Ab^j c^k B \underset{P_2}{\Longrightarrow} a^{i+1} A'b^{j+1} c^{k+1} B'$

To such a string we have to apply P_1 again so we get:

$$a^{i+1} A'b^{j+1} c^{k+1} B' \underset{P_2}{\Longrightarrow} a^{i+1} Ab^{j+1} c^{k+1} B$$

This is the only possibility of using P_1. However, P_2 can be applied to a string $a^i Ab^j c^k B$ in the $= 2$ mode also using only one nonterminal rule (replacing either A or B by A' or B', respectively), and one terminal rule (removing the remaining symbol A or B). To a string containing only one nonterminal (which is different from S), none of the two components can be applied. Consequently, we have to use, in turn, the first component and the nonterminal rules of the second one, and we have to finish the derivation by using the terminal rules of P_2.

Now we present another alternative to solve the problem introduced above. We show that it is possible to automatically translate a grammar system to a concurrent program, and we use the Owicki-Gries theory to give our proofs.

We use the CD grammar system given above to exemplify this. So first we make the automatic translation of Γ_1 to a concurrent program:

$$
\begin{array}{l}
P : \{Begin\ Main\ Program\} \\
\quad \{declaration\ of\ variables\} \\
\quad w := S; \\
\quad P : \{w = S\} \\
\quad do\ belong(w, non\_terminals) \rightarrow \\
\qquad Prog_1 \quad \| \quad Prog_2 \\
\quad od \\
\quad Q : \{w \in \{\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n \mid \mathbf{n} \geq \mathbf{0}\}\} \\
\quad \{End\ Main\ Program\}
\end{array}
$$

For this example the translation generates programs $Prog_1$ and $Prog_2$, one program $Prog_i$ for each grammar G_i, and a global variable w whose initial value is S has been introduced to represent the current sentential form that all programs $Prog_i$ can access and modify. The fact that in the grammar system Γ_1 derivation finishes when a string of terminals is generated is modelled here by the cyclic instruction "*do Condition* $\rightarrow$ *Instructions od*" that iterates while the *Condition* is satisfied, in this case while string w contains non terminals. The set of productions P_i of each G_i are represented by the alternative construction called "if" that executes an assignment to the right of one arrow if one of the predicates to the left of the arrow is true. More than one predicate to the left of the arrows can be true, and in this case one is chosen non deterministically.

And the $= 2$-mode of derivation of G_1 and G_2 is preserved by adding the variable *cont*, to count the number of derivations performed on the sentential form. If the number of derivations is different from two, the programs abort. Also the symbols $<$ and $>$ enclose programs $Prog_1$ and $Prog_2$ to denote that these programs are considered atomic instructions. This means that once the processor is assigned to the program, it can not be released or reassigned to another program before its execution finishes.

$Prog_1$: $\{program\ for\ G_1\}$
$cont : int;$
$< cont := 0;$
$do\ (w = xSy \vee w = xA'y \vee w = xB'y) \wedge cont \neq 2 \rightarrow$
$if\ w = xSy \rightarrow w := xSy;\ cont := cont + 1;$
$w = xSy \rightarrow w := xABy;\ cont := cont + 1;$
$w = xA'y \rightarrow w := xAy;\ cont := cont + 1;$
$w = xBy \rightarrow w := xB'y;\ cont := cont + 1;$
fi
$od;$
$if\ (cont \neq 2) \longrightarrow abort\ >$
$\{end\ program\ Prog_1\}$

$Prog_2$: $\{program\ for\ G_2\}$
$cont : int;$
$< cont := 0;$
$do\ (w = xAy \vee w = xBy) \wedge cont \neq 2 \rightarrow$
$if\ w = xAy \rightarrow w := xaA'by;\ cont := cont + 1;$
$w = xBy \rightarrow w := xcB'y;\ cont := cont + 1;$
$w = xAy \rightarrow w := xaby;\ cont := cont + 1;$
$w = xBy \rightarrow w := xcy;\ cont := cont + 1;$
fi
$od;$
$if\ (cont \neq 2) \longrightarrow abort\ >$
$\{end\ program\ Prog_2\}$

So we have automatically generated each program $Prog_i$ for each grammar G_i where each program $Prog_i$ modifies the global variable w in the same way as each grammar G_i modifies the current sentential form and preserves the mode of derivation of G_i.

What remains to be done is to prove that when the programs $Prog_1$ and $Prog_2$ that we have defined run concurrently, they behave like Γ_1. According to the Owicki-Gries theory that we have introduced, this is equivalent to proving the *global correctness* of the multiprogram with respect to the precondition $\{w = S\}$ and postcondition $\{w \in \{a^n b^n c^n \mid n \geq 0\}\}$.

The analysis needed for this proof is similar to the one we showed above but in the programming framework. But for some problems, like this one, the Owicki-Gries theory also contemplates the possibility of using the so-called *System Invariant* strategy. To apply this strategy we need to find a predicate that remains invariant throughout all the computation and that synthesizes the behavior of the multiprogram and in case we find it we reduce the number of proofs to a linear size.

Definition 1. *By definition a relation I is a system invariant whenever:*

- it holds initially, i.e. is implied by the precondition of the multiprogram as a whole,

- it is maintained by each individual atomic statement $\{Q\}\mathcal{S}$ of each individual component, i.e. whenever for each such $\{Q\}\mathcal{S}$, $\{I \wedge Q\}\mathcal{S}\{I\}$ is a correct Hoare-triple.

For the previous program $\mathcal{P}$ we can give this system invariant:

$$\begin{aligned}\text{Inv } \mathbf{I} : [&w = S \vee (w = a^n A' b^n c^n B' \wedge n \geq 0) \vee (w = a^n A b^n c^n B \wedge n \geq 1) \vee \\ &\vee (w = a^n A' b^n c^n \wedge n \geq 1) \vee (w = a^n b^n c^n B' \wedge n \geq 1) \vee \\ &\vee (w = a^n b^n c^n \wedge n \geq 1)]\end{aligned}$$

And proving that the predicate I is invariant is equivalent to proving that:

1. $(w = S) \rightarrow I$,
2. $\{I\} Prog_1 \{I\}$,
3. $\{I\} Prog_2 \{I\}$,
4. $I \wedge (\mathcal{P} \text{ terminates}) \longrightarrow (w = a^n b^n c^n \wedge n \geq 1)$.

While the proof of 1 and 4 is trivial, the proof of 2 and 3 requires an analysis by cases checking that $Prog_1$ and $Prog_2$ always rewrite strings satisfying the invariant in new strings that satisfy the invariant. For reasons of space we do not provide the proof here.

With this example we have shown that it is possible to automatically translate a CD grammar system Γ to a concurrent program $\mathcal{P}$. In this way the problem of proving that Γ generates a language L is transformed into the problem of proving that the program $\mathcal{P}$ obtained from the translation is correct with respect to the precondition $\{w = S\}$ and the postcondition $\{w \in L\}$. So since as well as analysis by cases we now have the *global correctness* strategy from the Owicki-Gries theory to prove that a grammar system generates a language. And for some problems, like this example, it is also possible to prove global correctness through the system invariant strategy.

From this example we can point out some advantages of the system invariant strategy over the analysis by cases technique:

- Once the invariant predicate has been proposed the number of proofs to be made is *linear*, instead of the *exponential* number of proofs needed with analysis by cases.

- With analysis by cases we can capture the overall behavior of a system by a general sequence of derivations *including detailed information,* such as how grammars interact, which productions they apply, how they change the sentential form, etc. When we apply the system invariant technique we capture the overall behavior of the system by an invariant that shows all the possible values of the sentential form and hides information that the previous technique gives. So we can say that the invariant system captures the overall behavior in a *more abstract way.*
- With analysis by cases, apart from showing the shape that any sequence of derivation should have, we need to prove that this is the only possible sequence of derivations, and add an explanation in *natural language.* Like the system invariant technique we need to prove in the framework of predicate calculus that each program $Prog_i$ preserves the invariant. This is done by *formal proofs.*

Another benefit of the Owicki-Gries logic of programming is that it can be used to prove certain matters related to dynamic aspects of a grammar such as: reachability of a configuration, absence of progress because of circularity (in the case of PC grammar systems with communication by query), deadlock, etc.

Another advantage of the Owicki-Gries logic of programming is that it can help us to simultaneously construct the grammar systems that a given language specification generates and the proof that it generates (see [1]). This is a great improvement, because we did not have any techniques in the framework of grammar systems, to help us to solve this kind of problem. We give two examples of the use of this strategy. The first example is taken from [8]:

Theorem 1. $L_{cd} \in NPC_3(REG)$

Proof. We want to find a non-returning, non-centralized grammar system Γ with regular components that generates L_{cd}. This problem is transformed into the equivalent problem of finding a multiprogram $\mathcal{P}$ that behaves like Γ and is correct with respect to the specification:

$$\{(w_1 = S_1) \wedge (w_2 = S_2) \wedge ..... \wedge (w_n = S_n) \mid n \geq 1\}\ \mathcal{P}\ \{w_1 \in L_{cd}\}.$$

The problem remains the same, but we use different tools to solve it: instead of *induction* and *analysis by cases* available in the framework of grammar systems we use *Logic, the Owicki-Gries theory* and *programming strategies* from the programming framework.

The strategy we followed for this proof is frequently used for the development of programs. It is called *refinement of the problem* and consists of:

(I) First, start with an outline of the solution, which identifies the basic principle by which the input can be transformed into the output. Define pre- and post- conditions for each of the subproblems that are identified as part of the solution for the whole problem.

For our problem we propose this idea:

$\{(w_1 = S_1) \wedge (w_2 = S_2) \wedge ..... \wedge (w_n = S_n)\}$

Subproblem 1*: (Rewrite)^p, with $p \geq 1$*

$\{(w_1 = S_1) \wedge .... \wedge (w_i = a^p S_i) \wedge .... \wedge (w_j = c^p S_j) \wedge .... \wedge (w_n = S_n) \wedge (p \geq 1)\}$

Subproblem 2*: (Rewrite; Communication)$^+$*

Find a way to stop the productions of *a's* and *c's*, through synchronization by communication.

$\{(w_1 = a^r N_1) \wedge ..... \wedge (w_k = c^r N_2) \wedge ..... \wedge (w_n = S_n) \wedge (r \geq 1) \wedge (N_1, N_2 \in N)\}$

Subproblem 3*: (Rewrite)^m, with $m \geq 1$*

$$\left\{ \begin{array}{l} (w_1 = a^r b^m Q_k) \wedge ..... \wedge (w_k = c^r d^{m-1} N_3) \wedge ..... \wedge (w_n = S_n) \wedge \\ (r, m \geq 1) \wedge (Q_k \in K) \wedge (N_3 \in N) \end{array} \right\}$$

Subproblem 4*: Communication*

$\{(w_1 = a^r b^m c^r d^{m-1} N_3) \wedge (r, m \geq 1) \wedge (N_3 \in N)\}$

Subproblem 5*: Rewrite*

$\{(w_1 = a^r b^m c^r d^m) \wedge (r, m \geq 1)\}$

or equivalently

$\{w_1 \in \{a^r b^m c^r d^m \wedge r \geq 1 \mid m \geq 1\}\}$

(II) Now we make the outline indicated more precise, refine the subproblems by trying to simultaneously find the instructions that solve the subproblems and the proof of its local correctness. We also discuss the difficulties we might have when proving overall correctness.

In the refinement of subproblems 1, 2, 3, 4 and 5 we proposed three programs $Prog_1$, $Prog_2$ and $Prog_3$. These programs make up the multiprogram P, run simultaneously and behave like a non-returning, non-centralized grammar system with regular productions. With the subproblems we have identified in the step above, they behave locally correctly.

In the case of Subproblem 1 we propose this refinement:

$\{(w_1 = S_1) \wedge (w_2 = S_2) \wedge (w_3 = S_3)\}$

Subproblem 1*: Rewrite^n, with $n \geq 1$*

$Prog_1$ rewrites $n-1$ times S_1 to aS_1 and then rewrites S_1 to aA, $Prog_2$ rewrites $n-1$ times S_2 to cS_2 and then rewrites S_2 to cB and $Prog_3$ rewrites $n-1$ times S_3 to S_3, until it decides to finish the production of a's and c's, rewriting S_3 to Q_2.

To be sure that $w_2 = c^n B$ when $Prog_3$ introduces Q_2, $Prog_3$ should not be able to rewrite S_2, and after $Prog_2$ introduces B it should rewrite it for another nonterminal and not introduce B any more.

The reason why $w_1 = a^n A$ and $w_1 \neq a^n S_1$ is that this is the only possibility that does not lead to deadlock, as the states of the next subproblem show.

$\{(w_1 = a^n A) \wedge (w_2 = c^n B) \wedge (w_3 = Q_2) \wedge (n \geq 1)\}$

For Subproblem 2 we propose this sequence of rewriting and communications as a refinement:

$\{(w_1 = a^n A) \wedge (w_2 = c^n B) \wedge (w_3 = Q_2) \wedge (n \geq 1)\}$

Subproblem 2:

Communication

$\{w_1 = a^n A \wedge w_2 = c^n B \wedge w_3 = c^n B \wedge n \geq 1\}$

Rewrite

$Prog_1$ rewrites A to A', $Prog_2$ rewrites B to Q_1 and $Prog_3$ rewrites B to D

We do not allow any possibility other than $w_1 = a^n A' \wedge w_2 = c^n Q_1 \wedge w_3 = c^n D$.

To be sure that $w_1 = a^n A'$ after the rewriting step, we need $Prog_2$ to be defined only for A', and after $Prog_1$ introduces A' it should rewrite it to another nonterminal and not introduce A' anymore.

$\{w_1 = a^n A' \wedge w_2 = c^n Q_1 \wedge w_3 = c^n D \wedge n \geq 1\}$

Communication

$\{(w_1 = a^n A') \wedge (w_2 = c^n a^n A') \wedge (w_3 = c^n D) \wedge (n \geq 1)\}$

In the case of Subproblem 3 this is a possible refinement:

$\{(w_1 = a^n A') \wedge (w_2 = c^n a^n A') \wedge (w_3 = c^n D) \wedge (n \geq 1)\}$

Subproblem 3: *Rewrite*$^{m+1}$, *with* $m \geq 1$

$Prog_1$ rewrites A' to A" and rewrites $m-1$ times A" to bA", and then rewrites A" to bQ_3, $Prog_2$ always rewrites A' to A' and $Prog3$ rewrites D to D', then D' to D" and rewrites $m-1$ times D" to dD"

$\{(w_1 = a^n b^m Q_3) \wedge (w_2 = c^n a^n A') \wedge (w_3 = c^n d^{m-1} D") \wedge (n, m \geq 1)\}$

Refinement for Subproblem 4 and Subproblem 5 is very simple:

$\{(w_1 = a^n b^m Q_3) \wedge (w_2 = c^n a^n A') \wedge (w_3 = c^n d^{m-1} D") \wedge (n, m \geq 1)\}$

Subproblem 4: *Communication*

$\{(w_1 = a^n b^m c^n d^{m-1} D") \wedge (w_2 = c^n a^n A') \wedge (w_3 = c^n d^{m-1} D") \wedge (n, m \geq 1)\}$

Subproblem 5: *Rewrite*
$Prog_1$ rewrites D" to d
$\{(w_1 \in \{a^n b^m c^n d^m) \wedge (n, m \geq 1)\}\}$
Equivalently we propose a non-returning, non-centralized grammar system Γ with three regular components, defined in this way:

$$\Gamma = (N, K, \{a, b, c, d\}, (P_1, S_1), (P_2, S_2), (P_3, S_3))$$

where:
$N = \{S_1, S_2, S_3, A, A', A", B, D, D', D"\}$
$K = \{Q_1, Q_2, Q_3\}$
$P_1 = \{S_1 \longrightarrow aS_1, S_1 \longrightarrow aA, A \longrightarrow A', A' \longrightarrow A", A" \longrightarrow bA", A" \longrightarrow bQ_3,$
$D" \longrightarrow d\}$
$P_2 = \{S_2 \longrightarrow cS_2, S_2 \longrightarrow cB, B \longrightarrow Q_1, A' \longrightarrow A'\}$
$P_3 = \{S_3 \longrightarrow S_3, S_3 \longrightarrow Q_2, B \longrightarrow D, D \longrightarrow D', D' \longrightarrow D", D" \longrightarrow dD"\}$

(III) The last and most difficult step is to prove overall correctness.

In this case this means that we have to use the Owicki-Gries theory to show that the multiprogram $\mathcal{P}$ we constructed satisfies the next specification:

$$\{(w_1 = S_1) \wedge (w_2 = S_2) \wedge (w_3 = S_3)\}\ \mathcal{P}\ \{w_1 \in L_{cd}\}$$

Furthermore, $\mathcal{P}$ outputs the word $a^n b^m c^n d^n$ for any input formed by the pair of positive integers n, m. This is equivalent to proving that $L(\Gamma) = L_{cd}$.

According to the definition $Prog_1$, $Prog_2$ and $Prog_3$, which behave like G_1, G_2 and G_3, respectively, we propose the following invariant:

$$InvV : \left\{ \begin{array}{l} \left[\begin{array}{l} (w_1 = a^n S_1 \wedge n \geq 0) \vee (w_1 = a^n A \wedge n \geq 1) \vee (w_1 = a^n A' \wedge n \geq 1) \vee \\ \vee (w_1 = a^v b^n A" \wedge v \geq 1 \wedge n \geq 0) \vee (w_1 = a^v b^n Q_3 \wedge v \geq 1 \wedge n \geq 1) \vee \\ \vee (w_1 = a^v b^n c^g d^h D" \wedge v, n, g \geq 1 \wedge h \geq 0) \vee \\ \vee (w_1 = a^e b^f c^g d^h \wedge e, f, g, h \geq 1) \end{array} \right] \wedge \\ \wedge \left[\begin{array}{l} (w_2 = c^q S_2 \wedge q \geq 0) \vee (w_2 = c^q B \wedge q \geq 1) \vee \\ \vee (w_2 = c^q Q_1 \wedge q \geq 1) \vee (w_2 = c^q a^r A' \wedge q, r \geq 1) \end{array} \right] \wedge \\ \wedge \left[\begin{array}{l} (w_3 = S_3) \vee (w_3 = Q_2) \vee (w_3 = c^n B \wedge n \geq 1) \vee \\ \vee (w_3 = c^n D \wedge n \geq 1) \vee (w_3 = c^n D' \wedge n \geq 1) \vee \\ \vee (w_3 = c^n d^m D" \wedge n \geq 1 \wedge m \geq 0) \end{array} \right] \end{array} \right\}$$

But the Owicki-Gries theory of global correctness can be used to prove that after n rewriting, with $n \geq 1$, the only possible combination of values for

the sentential forms w_1, w_2 and w_3 that does not lead to a deadlock, is the one expressed by the state:

$$\{(w_1 = a^n A) \wedge (w_2 = c^n B) \wedge (w_3 = Q_2) \wedge (n \geq 1)\}$$

From this state it can be proved that the only valid continuation is the sequence of rewriting and communications described in step 2 of the refinement process, which reaches the state containing $\{w_1 \in \{a^n b^m c^n d^m \mid n, m \geq\}\}$

The strategy we have presented above differs from the traditional approach not in complexity, because the number of cases considered in the proofs are the same, but in the way the problem is approached. We suggest that the Owicki-Gries methodology could provide more possibilities for reasoning about problems than the strategies commonly used in the grammar system framework because:

1. It makes it possible to reason in a forward or data-driven way, as does case analysis, but also in a backward or goal-directed way. The notion of backward reasoning comes from psychology, as is pointed in [11] where this description of problem solving is given: *We may have a choice between starting with where we wish to end, or starting with where we are at the moment. In the first instance we start by analyzing the goal. We ask, "Suppose we did achieve the goal, how would things be different- what subproblems would we have solved, etc.?". This in turn would determine the sequence of problems, and we would work back to the beginning. In the second instance we start by analyzing the present situation, see the implications of the given conditions and lay-out, and attack the various subproblems in a "forward direction".*
2. Problems can be divided into subproblems because of the theorem: for any Q $\{P\}\mathcal{S}_0$; $\mathcal{S}_1\{R\} \Longleftarrow \{P\}\mathcal{S}_0\{Q\} \wedge \{Q\}\mathcal{S}_1\{R\}$, where P,R are predicates and $\mathcal{S}_0, \mathcal{S}_1$ are instructions. Also goals and subgoals are discussed in the psychology text mentioned above ([11]): *The person perceives in his surrounding goals capable of removing his needs and fulfilling his desires... And there is the important phenomenon of emergence of subgoals. The pathways to goals are often perceived as organized into a number of subparts, each of which constitutes and intermediate subgoal to be attained on the way to the ultimate goal.* These characteristics suggest that Owicki-Gries strategies are closer to how humans reason.

We give another example showing the combined use of the Owicki-Gries strategy and the so-called technique of refinement of problems.

Example 2. We are looking for a CCPC grammar system that generates this language: $\{xcxc \mid x \in \{a,b\}^*\}$

A possible division into subproblems could be:
$\{w_1 = S_1 \wedge w_2 = S_2 \wedge w_3 = S_3 \wedge w_4 = S_4\}$
Subproblem 1: (Rewrite)$^{n+1}$, $n \geq 0$
$Prog_4$ generates xX, $x \in \{a,b\}^*$, $\mid x \mid = n$.
$\{w_1 = S_1\text{'} \wedge w_2 = S_2\text{'} \wedge w_3 = S_3\text{'} \wedge w_4 = xX \wedge x \in \{a,b\}^* \wedge \mid x \mid = n \wedge X \in N\}$
Subproblem 2: Communication
$Prog_4$ communicates with $Prog_2$ and $Prog_3$ sending two copies of w_4.
$\{w_1 = S_1\text{'} \wedge w_2 = xX \wedge w_3 = xX \wedge w_4 = S_4 \wedge x \in \{a,b\}^* \wedge \mid x \mid = n \wedge X \in N\}$
Subproblem 3: (Rewrite)^p, $p \geq 0$
$Prog_2$ and $Prog_3$ replace X by c.
$\{w_1 = S_1\text{'} \wedge w_2 = xc \wedge w_3 = xc \wedge w_4 = yX \wedge x \in \{a,b\}^* \wedge \mid x \mid = n \wedge$
$\wedge X \in N \wedge y \in \{a,b\}^* \wedge \mid y \mid = p\}$
Subproblem 4: Communication
$Prog_1$ receives the content of w_2 and w_3 from $Prog_2$ and $Prog_3$.
$\{w_1 = xcxc \wedge w_2 = yX \wedge w_3 = yX \wedge w_4 = S_4 \wedge x \in \{a,b\}^* \wedge \mid x \mid = n \wedge$
$\wedge X \in N \wedge y \in \{a,b\}^* \wedge \mid y \mid = p\}$

As we can see, with the Owicki-Gries theory we can simultaneously propose a multiprogram $\mathcal{P}$ with programs $Prog_1$, $Prog_2$, $Prog_3$ and $Prog_4$ and prove its correctness with respect to the precondition $\{w_1 = S_1 \wedge w_2 = S_2 \wedge w_3 = S_3 \wedge w_4 = S_4\}$ and postcondition $\{w_1 \in \{xcxc \mid x \in \{a,b\}^*\}\}$.

Equivalently we simultaneously define a grammar system $\Gamma_3 \in CCPC_4(CF)$ and the proof that $L(\Gamma_3) = \{xcxc \mid x \in \{a,b\}^*\}$, where Γ_3 is defined in this way:

$$\Gamma_3 = (\{S_1, S_2, S_3, S_4, S_1\text{'}, S_2\text{'}, S_3\text{'}, S_4\text{'}, X\}, \{a,b,c\},$$
$$(S_1, P_1, R_1), (S_2, P_2, R_2), (S_3, P_3, R_3), (S_4, P_4, R_4))$$

where:

$P_1 = \{S_1 \to S_1\text{'}\}, \quad R_1 = \{a,b\}^* c,$
$P_2 = \{S_2 \to S_2\text{'}, X \to c\}, \quad R_2 = \{a,b\}^* X,$
$P_3 = \{S_3 \to S_3\text{'}, X \to c\}, \quad R_3 = \{a,b\}^* X,$
$P_4 = \{S_1 \to aS_4, S_4 \to bS_4, S_4 \to X\}, \quad R_4 = \emptyset.$

We do not include here the proof of global correctness of the program proposed, but encourage the reader to do it.

So far the main effort in grammar system theory has focused on finding grammar systems with the fewest possible number of grammars and more restricted productions, to show how distribution and communication can make simple components very powerful when they work together. Some studies on the computational complexity measure of PC grammar systems that considers the number of communications between grammars have been presented in [10] and [14]. This apart, the most investigated complexity measure is the number of grammars that a PC grammar system consists of, which is clearly a descriptional complexity measure. So a very important matter has been forgotten: the efficient use of time. The opposite has happened in the programming area (see [7]), where research has focused on looking for techniques to parallelize algorithms and to help programmers to design more efficient concurrent algorithms.

Although there are no recipes to follow, in some cases we can construct efficient grammar systems using some of the methodical approaches developed in the programming framework that maximize the range of options considered and that provide mechanisms for evaluating alternatives.

For example if we calculate the time that the grammar system Γ_3 defined above spends to generate a string $xcxc$ with $x \in \{a,b\}^*$ and $\mid x \mid = n$, it is $O(n)$ in the best case. If we want to improve the efficiency of Γ_3 in terms of time taken to produce a string, we can try to apply some of the strategies developed in the programming framework to design parallel algorithms. For this example we can apply so-called *functional decomposition.*

Definition 2. *(Functional decomposition) Functional decomposition is a strategy of partitioning used to the design concurrent algorithms. This approach uses computation to expose opportunities for parallel execution. Hence, the idea is to define a large number of small tasks in order to yield a fine-grained decomposition of a problem.*

Example 3. We can apply the functional decomposition strategy over Γ_3 to generate another grammar system Γ_4 that solves this problem in less time.

We focus on the computation of the string $x \in \{a,b\}^*$ and we discover that this task can be done by m grammars working simultaneously instead of only one grammar. Thus, we can reduce the time to $O(n/m)$, in the best case.

For defining Γ_3 we propose this refinement of *Subproblem 1*:
$\{w_1 = S_1 \wedge w_2 = S_2 \wedge w_3 = S_3 \wedge w_4 = S_4\}$
Subproblem 1: (Rewrite)$^{n+1}$, $n \geq 0$
$Prog_4$ generates xX, $x \in \{a,b\}^*, \mid x \mid = n$.
$\{w_1 = S_1\text{'} \wedge w_2 = S_2\text{'} \wedge w_3 = S_3\text{'} \wedge w_4 = xX \wedge x \in \{a,b\}^* \wedge \mid x \mid = n\}$

To improve efficiency we propose this other refinement of the same subproblem:
$\{w_1 = S_1 \wedge w_2 = S_2 \wedge w_3 = S_3 \wedge w_4 = S_4 \wedge .... \wedge w_{i+4} = S_{i+4} \wedge ...$
$... \wedge w_{m+3} = S_{m+3} \wedge w_{m+4} = S_{m+4} \wedge 1 \leq i \leq m-1 \wedge 1 \leq m\}$
Subproblem 1:
$(Rewrite)^{t+1}$, $t \geq 0$
$Prog_{i+4}, ..., Prog_{m+3}$ generates $x_i, ..., x_{m-1} \in \{a,b\}^*$, $1 \leq i \leq m-1$,
$1 \leq m$ and $Prog_{m+4}$ generates $x_m Y$, $x_m \in \{a,b\}^*$
$\{w_1 = S_1\text{'} \wedge w_2 = S_2\text{'} \wedge w_3 = S_3\text{'} \wedge w_4 = S_4\text{'} \wedge ... \wedge w_{i+4} = x_i \wedge$
$\wedge ... \wedge w_{m+3} = x_{m-1} \wedge w_{m+4} = x_m Y \wedge 1 \leq i \leq m-1 \wedge 1 \leq m \wedge$
$\wedge x_1, ..., x_m \in \{a,b\}^* \wedge Y \in N\}$
Communication
$Prog_4$ receives the $x_1, ..., x_{m-1} \in \{a,b\}^*$ produced by $Prog_5, ..., Prog_{m+3}$
followed by $x_m Y$, $x_m \in \{a,b\}^*$ produced by $Prog_{m+4}$
$\{w_1 = S_1\text{'} \wedge w_2 = S_2\text{'} \wedge w_3 = S_3\text{'} \wedge w_4 = x_1 ... x_m Y \wedge ... \wedge$
$\wedge w_{i+4} = S_{i+4} \wedge ... \wedge w_{m+3} = S_{m+3} \wedge w_{m+4} = S_{m+4} \wedge$
$\wedge 1 \leq i \leq m-1 \wedge 1 \leq m \wedge x_1 ... x_m \in \{a,b\}^* \wedge Y \in N\}$
$(Rewrite)^{s+1}$, $s \geq 0$
$Prog_4$ replaces Y by X.
$\{w_1 = S_1\prime \wedge w_2 = S_2\prime \wedge w_3 = S_3\prime \wedge w_4 = x_1 ... x_m X \wedge ... \wedge w_{i+4} = y_i \wedge$
$\wedge ... \wedge w_{m+3} = y_{m-1} \wedge w_{m+4} = y_m Y \wedge 1 \leq i \leq m-1 \wedge 1 \leq m \wedge$
$\wedge x_1 ... x_m \in \{a,b\}^* \wedge y_1, ..., y_m \in \{a,b\}^* \wedge Y \in N\}$

The rest is analogous to the analysis we made for Γ_3, and according to our previous analysis we get $\Gamma_4 \in CCPC_{m+4}(CF)$, $m \geq 1$ defined in this way:

$$\Gamma_4 = (\{S_1, S_2, S_3, ..., S_{m+4}, S_1\text{'}, S_2\text{'}, S_3\text{'}, ..., S_{m+4}\text{'}, X, Y\}, \{a,b,c\},$$
$$(S_1, P_1, R_1), (S_2, P_2, R_2), ..., (S_{m+4}, P_{m+4}, R_{m+4}))$$

where:

$P_1 = \{S_1 \rightarrow S_1\text{'}\}$, $R_1 = \{a,b\}^* c$,
$P_2 = \{S_2 \rightarrow S_2\text{'}, X \rightarrow c\}$, $R_2 = \{a,b\}^* X$,
$P_3 = \{S_3 \rightarrow S_3\text{'}, X \rightarrow c\}$, $R_3 = \{a,b\}^* X$,

$P_4 = \{S_4 \rightarrow S_4', Y \rightarrow X\}$, $R_4 = \{a,b\}^*Y \cup \{a,b\}^*$,
$P_{i+4} = \{S_{i+4} \rightarrow aS_{i+4}, S_{i+4} \rightarrow bS_{i+4}, S_{i+4} \rightarrow a, S_{i+4} \rightarrow b, S_{i+4} \rightarrow \lambda\}$, $R_{i+4} = \emptyset$, $1 \leq i \leq m-1$
$P_{m+4} = \{S_{m+4} \rightarrow aS_{m+4}, S_{m+4} \rightarrow bS_{m+4}, S_{m+4} \rightarrow Y\}$, $R_{m+4} = \emptyset$, $1 \leq m$

The proof of global correctness of the multiprogram proposed is left as an exercise for the reader.

We improve efficiency, because Γ_4 generates strings $xcxc$ with $x \in \{a,b\}^*$ and $\mid x \mid = n$ in $O(n/m)$, in the best case. And we show with this example that we can use strategies available in the programming framework to design grammar systems that derive strings in less time.

4.2 How can programming benefit from grammar systems?

If, for example, we want to prove in the grammar system theory that there is no grammar system with n components with a certain protocol of communication that generates a language L, we use analysis by cases and induction strategies. If we translate this problem to the programming framework, we have to prove that it is not possible to find a multiprogram $\mathcal{P}$ with n programs running concurrently, that communicate with the same protocol and which is correct with respect to this specification:

$$\{(w_1 = S_1) \wedge (w_2 = S_2) \wedge ... \wedge (w_n = S_n) \wedge n \geq 1\}\ \mathcal{P}\ \{w_1 \in L\}$$

But in the programming framework we have no strategies for reasoning in the negative way. The only strategies available in this framework are *verification*, which consists of a given multiprogram that proves its correctness with respect to a specification (example 4), and the *constructive approach*, which we have exemplified with theorem 6, examples 7 and 9 that consist of simultaneously constructing a program and the proof of its correctness with respect to a specification. Both strategies are useful for getting positive results.

The lack of strategies that can prove this kind of negative result in the programming framework makes us think of the possibility of translating them to the grammar system framework and using the tools available there to solve them.

For example, let us the take the problem of proving that there is no grammar system of any type with two regular components that can generate L_{cd}. If we get a solution for this we prove that $L_{cd} \in NPC_3(REG)$ is

the most economical solution with respect to the number of components. If we translate this problem into the programming framework, we want to prove that it is not possible to find a multiprogram $\mathcal{P}$ with two programs $Prog_1$ and $Prog_2$ running concurrently, modifying w_1 and w_2 in a right-linear way, which is correct with respect to this specification: $\{(w_1 = S_1) \wedge (w_2 = S_2)\}$ $\mathcal{P}$ $\{w_1 \in \{a^n b^m c^n d^m \mid n, m \geq\}\}$. But because there is no strategy in the programming framework to solve this kind of problems, we solve it with the tools available in the grammar system framework: namely analysis by cases. The proof of the theorem $L_{cd} \notin X_2(REG)$, for $X \in \{PC, CPC, NPC, NCPC\}$, can be read from [8].

Now if we go back to the topic of efficiency that we pointed out above, the grammar system framework concentrates on agglomerate tasks as much as possible. The aim is to get grammar systems with fewer grammars, to prove the power of communication. The opposite is the case in the programming framework where programmers try to partition programs into as many tasks as necessary to improve efficiency in time, looking for strategies to parallelize programs. So it looks like researchers are working in different directions.

But some results of the grammar system theory can benefit the concurrent programming framework. For example, this theorem that makes it possible to transform a grammar system of m grammars into a grammar system of n grammars that generate the same language:

$$CF = CD_{1,*}CF(t) = CD_{2,*}CF(t) \subset CD_{3,*}CF(t) \text{ and} \tag{1}$$

$$CD_{3,*}CF(t) = CD_{*,*}CF(t) = ET0L \tag{2}$$

There are many theorems of this kind in the grammar system theory that translated to the programming framework speak about the number of programs needed to generate a certain language (refer to [4]). This is a contribution by grammar system theory to the programming framework, where there are no results about the number of programs needed to solve a problem. It would be very interesting for the design of concurrent programs if some of these transformations were also to consider efficiency. Any results about how to transform a program $\mathcal{P}$ that has m multiprograms running concurrently into a program with n multiprograms that solves the same problem more efficiently would be a great contribution to the concurrent programming theory.

5 Conclusions and Future Work

In this paper we briefly outline and illustrate the strong relationship between two mechanisms of distributed and cooperating computations: *grammar systems* and *concurrent programming*. We show that it is possible to automatically translate a grammar system into a concurrent program and make proofs using the tools available in the programming framework. The problem used to show the translation from a grammar system to a concurrent program was a homogenous CD grammar system with $= k$-mode of derivation. But this automatic translation can also be done for all the other models of grammar systems: homogeneous CD grammar systems with the other modes of derivations, hybrid CD grammar systems and networks of language processors.

The traditional approach to the problem of finding a grammar system generating a given language is: first propose a grammar system and then find a proof that it generates the language. In this paper we present a new approach, taken from the programming framework. It consists of simultaneously finding the grammar system that generates a given language and a proof that the grammar system found generates it. We think that it would be interesting to study this approach in more detail, and try to apply it to other well-known languages. We could even try to find other programming strategies, apart from the strategy of refinement of problems shown here, that could be useful in solving problems related to grammar system theory.

Until now not much attention has been paid to the time taken to generate a language with a grammar system, while in the programming framework the efficiency issue has been the main topic of research in recent years. We propose to follow some of the methodical approaches developed in the programming framework to construct more efficient grammar systems.

Moreover we can think about how programming theory can benefit from grammar system theory. The lack of strategies in the programming framework to prove negative results of the type: $\mathcal{L} \neq L(\Gamma)$ for a language $\mathcal{L}$ and any grammar system Γ, makes us think that such problems might be solved by translating them into the grammar system framework where they can be solved using the tools available there.

It is our opinion that this work opens up possibilities for further research and that it seems worthwhile to continue in this direction.

Acknowledgements

This work has been made possible thanks to the research grant "Programa Nacional para la formación del profesorado universitario", from the Spanish Ministry of Education, Culture and Sports.

References

1. Burns, A. and G. Davies (1993). *Concurrent Programming*. Wokingham: Addison-Wesley.
2. Chitu, A. (1997). PC grammar systems versus some non-context free constructions from natural and artifical languages. In Gh. Păun and A. Salomaa (eds.), *New Trends in Formal Languages. Control, Cooperation, and Combinatorics*, pp. 278–287. Berlin: Springer.
3. Cshuhaj-Varjú, E. and J. Dassow (1990). On cooperating/distributed grammar systems. *Journal of Information Processing and Cybernetics EIK*, 26(1-2): 49–63.
4. Cshuhaj-Varjú, E., J. Dassow, J. Kelemen and Gh. Păun (1994). *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*. London: Gordon and Breach.
5. Dassow, J., Gh. Păun and G. Rozemberg (1997). Grammar systems. In G. Rozenberg and A. Salomaa (eds.), *Handbook of Formal Languages*, volume 2, pp. 155–214. Berlin: Springer.
6. Dijkstra, D. and W. Edsger (1976). *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation.
7. Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley.
8. Grando, A. and V. Mitrana (2007). Can PC grammar systems benefit from concurrent programming? *Fundamenta Informaticae*, 76(3): 325–336.
9. Gries, D. (1981). *The Science of Programming*. Berlin: Springer.
10. Hromkovič, J., J.V. Kari and L. Kari (1994). Some hierarchies for the communication complexity measures of cooperating grammar systems. *Theoretical Computer Science*, 127(1): 123-147.
11. Krech, D. and R. S. Cruthfield (1958). *Elements of Psycology*, p. 383. New York: Knopf.
12. Martín-Vide, C. and V. Mitrana (2000). Parallel communicating automata systems- a survey. *Korean J. Comput. Appl. Math.*, 7(2): 237-257.
13. Owicki, S. and D. Gries (1976). An Axiomatic Proof Technique for Parallel Programs 1. *Acta Informatica*, 6: 319–340.

14. Pardubská, D. (1994). On the power of communication structure for distributive generation of languages. In G. Rozenberg and A. Salomaa (eds.), *Developments in Language Theory. At the Crossroads of Mathematics, Computer Science and Biology*, pp. 419-429. London: World Scientific.
15. Păun, Gh. and L. Sântean (1989). Parallel communicating grammar systems: the regular case. *Annals of the University of Bucharest, Mathematics-Informatics Series*, 38: 55–63.
16. Rozenberg, G. and A. Salomaa (eds.) (1997). *Handbook of formal languages*. Berlin: Springer.