Grammars as Processes

Javier Esparza

Laboratory for Foundations of Computer Science
Division of Informatics, University of Edinburgh

1 Preface

In 1999 I met Grzegorz Rozenberg in Amsterdam, while I was attending the ETAPS conference and he was taking part in a meeting. The next day I was giving a talk with the title "Grammars as Processes", and Grzegorz, who had seen it announced in the program, asked me about it. We had little time, and so I could barely sketch the contents. I think Grzegorz would have liked the talk, because it pointed out an interesting connection between two of his oldest loves, formal languages and concurrency theory, and showed how a model of computation derived from this connection has a natural application in the area of program analysis. He would have also liked to see how an abstract result obtained by Büchi in 1964 on regular canonical systems was the basis to new algorithms for the analysis of software.

This paper is a written version of the talk, and it also surveys the new results obtained since 1999. Sections 3 and 4 are taken from the Ph. D. Thesis of Richard Mayr [21], and have also been published in [22].

2 An Intuitive Introduction

Grammars count among the oldest tools in computer science. They are one of the few formal concepts that a speaker can safely use in front of an audience of computer scientists without having to introduce them. The "grammars as processes" research program proposes to look at grammars from a different point of view; not as language generators, but as generators of *behaviours* (formally captured as labelled transition systems). The purpose of this paper is to convince you, the reader, that this is an interesting thing to do, both from a theoretical and from a practical point of view.

In order to look at grammars in this way, we will wear three different pairs of glasses. The first pair is already waiting for us in Figure 1. It is called the *Sequential Model*, for reasons that will be clear soon. Let us look at a production of a left-linear grammar like

$$X \rightarrow aY$$

If we now put our glasses on, what we see is

$$X \xrightarrow{a} Y$$

We interpret X and Y as *processes*, and interpret the production as:

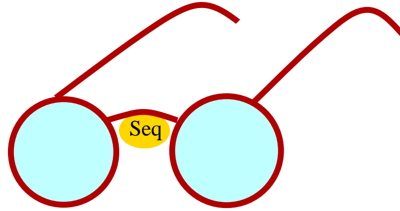

Fig. 1. The Sequential Model

Process X can perform the action a and become process Y

Readers familiar with process algebras are now raising their eyebrows, since this idea is at least 20 years old. But let us see how far can we take the analogy. The other possible productions in a left-linear grammar have the form

$$X \rightarrow a$$

If we put our glasses on, we see

$$X \xrightarrow{a} \epsilon$$

where ϵ denotes the empty word. We interpret this production as:

Process X can perform the action a and *terminate*

So through the glasses the empty word is seen as the terminated process which cannot execute any further actions.

If we now look at a context-free production like

$$X \rightarrow aYZ$$

through our glasses, what we see is

$$X \xrightarrow{a} Y \cdot Z$$

with the following interpretation:

> Process X can perform the action a and become the *sequential composition* of the processes Y and Z

This is why the glasses are called the Sequential Model. The intuition for $Y \cdot Z$ is that the process Y must execute and terminate before Z can start its execution. You may object that the usual rewriting rule for grammars does not reflect this. If $Z \rightarrow a$ is a production, then YZ can be rewritten into Ya, whereas the production $Z \xrightarrow{a} \epsilon$ should not be applicable to $Y \cdot Z$, because Y has not terminated yet. This objection is correct. In order to match the intuition, we need to change the inference rule for the application of productions. Without glasses we see the usual rule

$$\frac{X \to w}{uXv \to uwv}$$

but if we put our glasses on we see the *prefix rewriting* rule

$$\frac{X \xrightarrow{a} w}{X \cdot v \xrightarrow{a} w \cdot v}$$

Notice that in the context-free case the two rewriting rules are equivalent with respect to the generated languages: Every word generated by a context-free grammar can be generated through a leftmost derivation, and this derivation corresponds to prefix rewriting. However, they are not equivalent with respect to the generated behaviours.

Grammars with prefix rewriting have been less studied than ordinary ones, but they have a distinguished origin. They were introduced by Büchi in [4] under the name of *regular canonical systems*, as a variant of Post's canonical systems. An updated presentation can be found in Chapter 5 of Büchi's unfinished book [5].

Let us now consider a non-context-free production like

$$XY \to aZ$$

If we put our glasses on, what we see is

$$X \cdot Y \xrightarrow{a} Z$$

This has a natural interpretation in terms of *value passing* between processes. Imagine we wish to model the sequential composition of two processes, X and Y, where X somehow computes a boolean and Y passes this boolean to the environment by executing the actions t or f. If, for the sake of simplicity, we assume that X computes the boolean nondeterministically, then the system can be modelled by the productions

$$\begin{array}{ll} X \xrightarrow{a} T & T \cdot Y \xrightarrow{t} \epsilon \\ X \xrightarrow{a} F & F \cdot Y \xrightarrow{f} \epsilon \end{array}$$

The two productions on the right model passing the boolean from X to Y. The two possible executions of $X \cdot Y$ are

$$\begin{array}{c} X \cdot Y \xrightarrow{a} T \cdot Y \xrightarrow{t} \epsilon \\ X \cdot Y \xrightarrow{a} F \cdot Y \xrightarrow{f} \epsilon \end{array}$$

as expected.

It is now time to try the second pair of glasses, waiting for us to collect in Figure 2. It is the *Concurrent Model*. The Concurrent and Sequential models

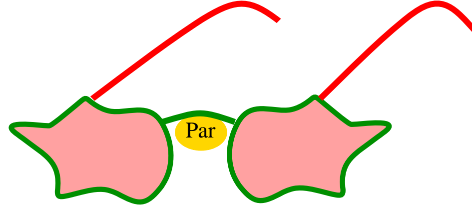

Fig. 2. The Concurrent Model

have the same effect when we look through them at linear productions. However, if we look at a context-free production like

$$X \rightarrow aYZ$$

through the new glasses, what we see is

$$X \xrightarrow{a} Y \parallel Z$$

which we interpret as:

> Process X can perform the action a and become the *parallel composition* of the processes Y and Z

We interpret $\parallel$ as a commutative operator, i.e., we consider $Y \parallel Z$ and $Z \parallel Y$ as the same term. We model concurrency as interleaving: "a and b can occur concurrently" is considered equivalent to "a and b can occur in any order, and both orders lead to the same final result." Under this proviso, we can take the following rule for applying productions in the presence of parallel composition:

$$\frac{t \xrightarrow{a} t'}{t \parallel u \xrightarrow{a} t' \parallel u}$$

If only context-free productions are allowed, processes can be created but they cannot interact, only coexist. Interaction is elegantly introduced by non-context-free productions. For instance, if we look at

$$XY \rightarrow aZV$$

through our new glasses, we see

$$X \parallel Y \xrightarrow{a} Z \parallel V$$

which we can interpret as:

> Processes X and Y can have a rendez-vous by jointly executing the action a and become the processes Z and V

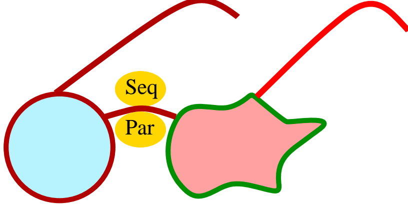

Fig. 3. The final model

The third and final model of glasses is shown in Figure 3, and combines the former two. If we put these glasses on we see productions like

$$X \xrightarrow{a} (Z \parallel V) \cdot U$$

which can be interpreted as a **parbegin–parend**: Z and V are started in parallel, and after both of them terminate— i.e., after both of them become the empty word—execution is continued with U.

Summarizing, the "grammars as processes" approach allows to model a variety of control mechanisms of programming languages in a simple way. The next section presents a formal syntax and semantics of the model.

3 Process Rewrite Systems

Let $Act = \{a, b, \ldots\}$ and $Cons = \{\epsilon, X, Y, \ldots\}$ be sets of *actions* and *process constants*, respectively. *Process terms* over Act and $Cons$ are given by

$$t ::= \epsilon \mid X \mid t_1 \cdot t_2 \mid t_1 \| t_2$$

where ϵ denotes the empty term. We introduce a syntactic equivalence relation $\equiv$ defined by the axioms

$$\begin{aligned} t_1 \cdot (t_2 \cdot t_3) &\equiv (t_1 \cdot t_2) \cdot t_3 \\ t_1 \parallel (t_2 \parallel t_3) &\equiv (t_1 \parallel t_2) \parallel t_3 \\ t_1 \parallel t_2 &\equiv t_2 \parallel t_1 \\ \epsilon \cdot t \equiv t \cdot \epsilon \equiv\ t\ &\equiv \epsilon \parallel t \equiv t \parallel \epsilon \end{aligned}$$

and by the two inference rules below, stating that $\equiv$ is a congruence.

$$\frac{t \equiv t'}{t \cdot u \equiv t' \cdot u} \qquad \frac{t \equiv t'}{t \parallel u \equiv t' \parallel u}$$

A *process rewrite system* (PRS) over $Cons$ and Act is a finite set Δ of *productions* of the form $t_1 \xrightarrow{a} t_2$, where t_1, t_2 are process terms such that $t_1 \not\equiv \epsilon$ and a is an action. The semantics of a PRS is a labelled transition system. A *labelled*

transition system is a triple (S, L, R), where S is a set of *states*, L is a set of *labels*, and $R \subset S \times L \times S$ is a *transition relation*. We denote $(s, l, s') \in R$ by $s \xrightarrow{l} s'$. The labelled transition system associated to a PRS has the set of all process terms as states, and the set Act as labels. For each action a, $t \xrightarrow{a} t'$ belongs to R if it can be derived from the productions in Δ using the following inference rules:

$$\frac{(t \xrightarrow{a} t') \in \Delta}{t \xrightarrow{a} t'} \qquad \frac{u \equiv t \quad u' \equiv t' \quad u \xrightarrow{a} u'}{u \xrightarrow{a} u'}$$

$$\frac{t \xrightarrow{a} t'}{t \cdot u \xrightarrow{a} t' \cdot u} \qquad \frac{t \xrightarrow{a} t'}{t \parallel u \xrightarrow{a} t' \parallel u}$$

Figures 4 and 5 show two PRSs together with the transition systems they generate.

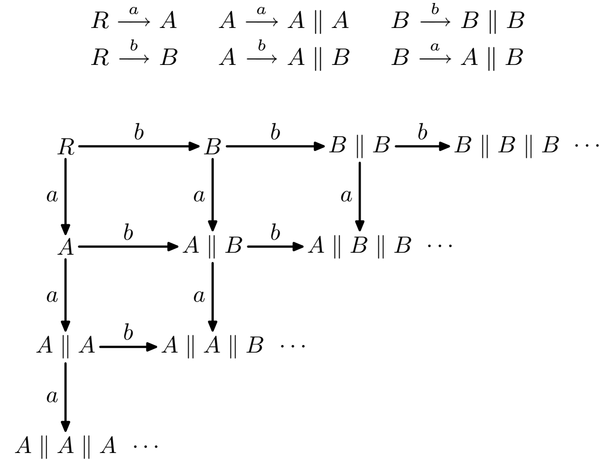

Fig. 4. An example with parallel composition

4 A process hierarchy

Grammars are customarily classified according to the Chomsky hierarchy. We introduce a similar hierarchy for PRSs, taken from [22]. We distinguish four classes of process terms:

- **1.** Terms consisting of a single process constant like X.
- **S.** Terms consisting of a single constant or a sequential composition of constants like $X \cdot Y \cdot Z$.

$$
\begin{array}{ll}
A \cdot B \xrightarrow{a} A \cdot B \cdot B & C \cdot B \xrightarrow{d} C \\
A \cdot B \xrightarrow{b} C & D \cdot B \xrightarrow{d} E \cdot B \\
A \cdot B \xrightarrow{c} D & E \cdot B \xrightarrow{d} D
\end{array}
$$

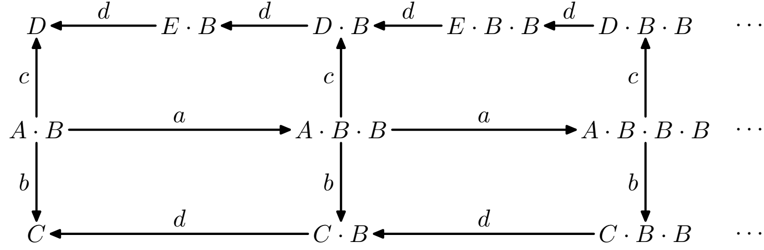

Fig. 5. An example with sequential composition

- **P.** Terms consisting of a single constant or a parallel composition of constants like $X \parallel Y \parallel Z$.
- **G.** General process terms with arbitrary sequential and parallel composition.

We obtain classes of PRSs by restricting the class of terms that can appear in the left-hand-side and in the right-hand side of a production. Given two classes of terms α and β, we denote by (α, β) the class of PRSs whose productions have α terms on the left and β terms on the right of the arrow. As in the Chomsky hierarchy, it only makes sense to consider classes in which β is at least as general as α (the remaining classes are uninteresting). Also, we do not allow the empty term in the left-hand-side of a production, but we always allow it in the right-hand-side. In this way we obtain the process hierarchy shown in Figure 6.

Basing on previous work [23], Mayr has shown that the hierarchy is strict with respect to bisimulation equivalence, i.e., no two classes generate the same transition systems up to bisimulation [22]. For instance, the transition system of Figure 5 cannot be generated (up to bisimulation) by any $(1, S)$-PRS. Incidentally, strictness does not hold for language equivalence, because the classes $(1, S)$ and (S, S) generate the same languages. So, in particular, some $(1, S)$-PRS generates the same language as the (S, S)-PRS in Figure 5, but no $(1, S)$-PRS is bisimilar to it.

4.1 Is the process hierarchy interesting?

Since the process hierarchy mimics the Chomsky hierarchy, we can ask ourselves why is the Chomsky hierarchy interesting and try to find similar arguments for the process hierarchy. The main two reasons for the relevance of the Chomsky hierarchy are probably the following:

- Its classes have interesting machine characterisations (finite and pushdown automata, Turing machines . . .), which allow to manipulate the languages generated by grammars and decide their properties.

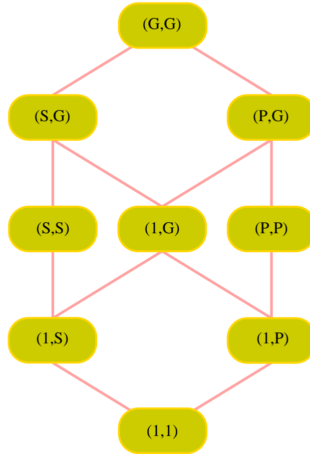

Fig. 6. The process hierarchy

– It is the undisputed formal basis for the lexicographic and syntactic analysis of programming languages, which provides an extremely important application making the hierarchy relevant outside pure theory.

We claim that two similar reasons can be used to support the relevance of the process hierarchy. In the rest of this section we show that the classes of the hierarchy have natural machine characterisations. In the next section we claim that the process hierarchy is a natural formal basis for control-flow analysis of programs. Or more daringly: In the same way that *grammars as language acceptors/generators* are an adequate basis for the syntactic analysis of programs, we claim that *grammars as processes* are an adequate basis for the analysis of their control-flow.

4.2 Machine characterisations

We start with the following proposition.

Proposition 1. *The labelled transition systems generated by* (S, S)*-PRSs are exactly those generated by pushdown automata (up to relabelling of states).*

For this to make sense we must first define the transition system generated by a pushdown automaton. The states of the transition system are all pairs (q, w), where q is a control state of the automaton and w is a word of stack symbols. The labels are the letters of the input alphabet. The transition relation contains $(q, w) \xrightarrow{a} (q', w')$ whenever this is a transition allowed by the transition function of the automaton.

One direction of the proof of Proposition 1 is simple. It is easy to see that a pushdown automaton can be encoded as a subclass of (S, S)-PRS in which the left-hand-sides of productions have length at most 2: If $(q, s_1 s_2 \dots s_n) \in$

$\delta(q, a, s)$, where δ is the transition function of the automaton, add a production $q \cdot s \xrightarrow{a} q \cdot s_1 \cdot s_2 \cdot \ldots \cdot s_n$ to the PRS. This shows that the transition systems generated by pushdown automata can also be generated by $S, S)$-PRS. The other direction is a bit more complex. Caucal shows in [7] that any (S, S)-PRS is equivalent—generates the same transition system up to relabelling of states—to an (S, S)-PRS in which the left-hand-sides of productions have length at most 2.

The following proposition has an even simpler proof.

Proposition 2. *The labelled transition systems generated by (P, P)-PRSs are exactly those generated by labelled (place/transition) Petri nets.*

Given a labelled place/transition Petri net, construct a PRS as follows. Take a constant for each place, and for each a-labelled transition having places $p_1, \ldots, p_n$ in its preset and places $p'_1, \ldots, p'_m$ in its postset add a production

$$p_1 \parallel \ldots \parallel p_n \xrightarrow{a} p'_1 \parallel \ldots \parallel p'_n$$

to the PRS.

The class $(1, S)$ corresponds to pushdown automata with one single control state. It is also equivalent to the class of context-free processes [6] and to Basic Process Algebra [2], the core of the ACP process algebra. $(1, P)$ is equivalent to the class of labelled Petri nets in which every transition has exactly one input place, and to a fragment of CCS called Basic Parallel Processes [15]. $(1, G)$ has been studied under the name Process Algebra [6]. Finally, the class $(1, 1)$ generates finite transition systems, and so it is equivalent to finite automata when seen as transition systems.

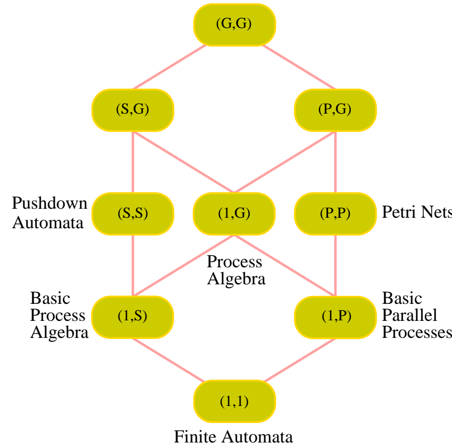

Fig. 7. Machine characterisations of the classes

These correspondences allow to describe PRSs as a process algebra combining the sequential composition operator of ACP and the synchronisation mechanism of Petri nets, with the goal of keeping a strong link to computational machines.

5 The process hierarchy and control-flow analysis

The thesis of this section is that PRSs constitute a very adequate tool for control-flow analysis of programs. On the one hand, they provide a simple semantics to sophisticated control mechanisms, like procedures, spawning of processes, multiple threads, and synchronisation. On the other hand, as shown in the last section, they are closely related to computational models for which a large collection of analysis and verification techniques have been developed. So PRSs are a natural interface between theoretical results and software analysis.

In the area of program analysis it is customary to represent the flow of control of programs in terms of flow-graphs. We use flow-graphs with program instructions attached to the edges. Figure 8 shows the flow-graph of a program consisting of two procedures, Π_0 and Π_1. Π_0 is the main procedure. The flow-

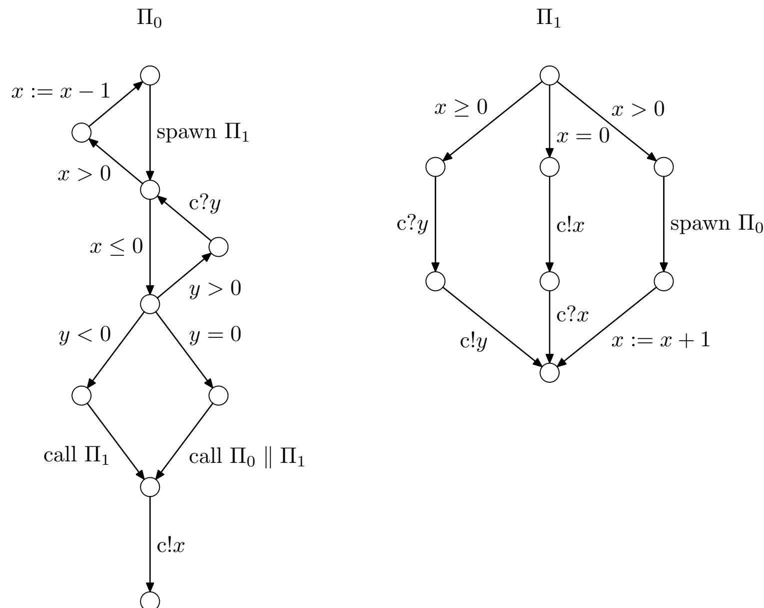

Fig. 8. A flow-graph

graph contains procedure calls (call Π_1), parallel procedure calls (call $\Pi_1 \parallel \Pi_1$), a process creation primitive, (spawn Π_0, spawn Π_1) and the well-known primitives $c?x$ and $c!expr$ for rendez-vous communication. As usual, $c!expr$ denotes sending the value of *expr* through the channel c, while $c?x$ denotes receiving a value

through channel c which is assigned to the variable x. Sending and receiving must occur together in a single atomic action. In general, we assume that the graph corresponding to a procedure has a unique distinguished initial node. In figures, this is always the topmost node.

Control-flow analysis abstracts from data. This means that we are allowed to take a conditional branch in a flow-graph even if the condition attached to it does not hold. In this way we add to the *real* executions of the program *spurious* executions. The semantics of the flow-graph is the set of all executions, real or spurious. Any property holding of all executions of the program will also hold, in particular, of its real executions.

Flow-graphs of plain while programs have one single component (the main procedure) whose edges are labelled with assignments or boolean conditions. In this case, the executions of a flow-graph are just the maximal paths of the graph, where a path is maximal if it is infinite or it is finite and its last node has no successors. In the presence of procedures, the flow-graph has several components, and a denotational definition of the set of executions becomes more complicated, since "jumps" to procedures and correct nesting of calls and returns have to be accounted for. Papers on program analysis usually define the executions of these flow-graphs (if at all) with the help of an inlining operator, which replaces an edge corresponding to a procedure call by the graph of the called procedure. If procedures are recursive or mutually recursive, inlining does not terminate, but a "limit" infinite flow-graph can still be defined as a suitable fixpoint, and executions are then defined as the maximal paths of this limit. This is already somewhat involved. If parallelism and synchronisation are added, a denotational definition becomes more complicated; in fact, papers dealing with these issues usually provide no formal definition at all. For instance, recent papers like [25, 24] give the semantics in English.

In contrast, it is remarkably simple to give an operational definition of the set of all executions. Just convert the flow-graph into a PRS with a distinguished *initial process constant*, and declare the executions to be the maximal paths of the associated transition system which start at this constant. The conversion proceeds as follows. The set of process constants is the set of nodes of the flow-graph (although we use lowercase for the nodes and uppercase for the constants for clarity). The initial constant is the one corresponding to the initial node of the main procedure. The productions of the PRS are described in Figure 9. For each edge labelled by an assignment or a boolean condition, we generate the production shown in the upper left part of the figure (the figure shows the case of an assignment). For each edge labelled by a procedure call, we generate the production shown in the upper right, where I denotes the constant corresponding to the initial node of procedure Π. The productions corresponding to a parallel procedure call and a spawn operation are shown in the middle row of the figure, where $I_1, \dots I_n$ denote the initial nodes of the procedures $\Pi_1, \dots, \Pi_n$. Notice that the production for the parallel call correctly models that all the parallel processes must terminate before execution can be resumed; this is due to the fact that a parallel composition of processes is equivalent to ϵ only if all its

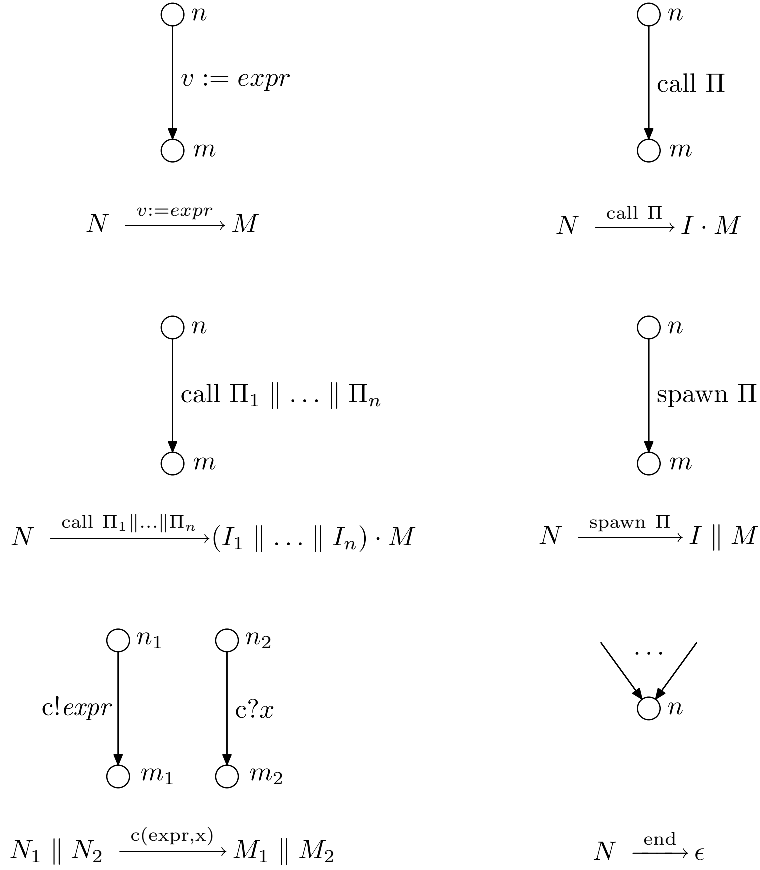

Fig. 9. PRS semantics of flow-graphs

components are equivalent to ϵ. For each (unordered) *pair* of edges labelled by an input and an output on the same channel, we generate the production shown in the lower left of the figure. Finally, for each node n without successors (end node of a procedure), we generate the production $N \xrightarrow{\text{end}} \epsilon$, as shown in the lower right.

If we consider a hierarchy of programs corresponding to more and more sophisticated control structures, we observe that the translations into PRSs match corresponding classes of the process hierarchy. Flat while-programs translate into the class $(1, 1)$. If we add procedures, we move to the class $(1, S)$. Through the addition of parallel threads through primitives for parallelism and process creation we jump to $(1, G)$. Finally, synchronisation between threads requires to use (P, G). If procedures are allowed to return a value from a finite domain, (sometimes called *multiple exit* procedures in other models, since the exit taken by the

procedure is used to encode the value), we obtain the classes (S, S), (S, G), and (G, G), respectively.

5.1 The PRS communication policy: Pros and cons

The presence of procedures leads to the distinction between global and local variables. Local variables are incarnated anew whenever the procedure is called; their values must be stored before the call and restored after the call is completed.

If we add procedures to a language with communication through channels, we find a similar distinction between global and local *channels*. Consider the flow-graph of Figure 10.

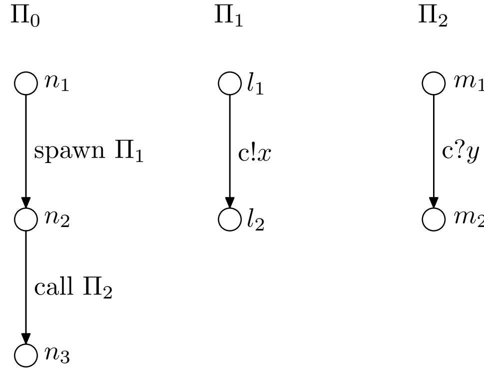

Fig. 10. The difference between global and local channels

Π_0 first spawns the process Π_1 and then calls the process Π_2. Should Π_1 and Π_2 be able to communicate or not? This depends on our assumptions on the nature of channel c. If the channel is globally defined, then communication can take place. On the contrary, if the channel is locally defined in, say, Π_1, then its existence is only known to Π_1 itself, to the processes Π_1 was called in parallel with (none in our example), and to the processes spawned by these processes (again, none). Communication is then forbidden.

The assumption underlying the PRS-semantics of flow-graphs is that *all channels are local*. Let us see why. The PRS-semantics of the flow-graph of Figure 10 is given by

$$\begin{array}{lcl} N_1 \xrightarrow{\text{spawn } \Pi_1} (N_2 \parallel L_1) & \quad & L_1 \parallel M_1 \xrightarrow{c(x,y)} L_2 \parallel M_2 \\ N_2 \xrightarrow{\text{call } \Pi_2} M_1 \cdot N_3 & & L_2 \xrightarrow{\text{end}} \epsilon \\ N_3 \xrightarrow{\text{end}} \epsilon & & M_2 \xrightarrow{\text{end}} \epsilon \end{array}$$

The only possible execution is

$$N_1 \xrightarrow{\text{spawn } \Pi_1} (N_2 \parallel L_1) \xrightarrow{\text{call } \Pi_2} (M_1 \cdot N_3) \parallel L_1$$

The production $L_1 \parallel M_1 \xrightarrow{c(x,y)} L_2 \parallel M_2$ is not applicable because this requires $L_1 \parallel M_1$ or $M_1 \parallel L_1$ to be a subterm of $(M_1 \cdot N_3) \parallel L_1$, which is not the case. In general, a production with a left-hand-side of the form $(X \parallel Y)$ cannot be applied to a term of the form $(X \cdot t \parallel Y \cdot t')$ if $t \neq \epsilon$ or $t' \neq \epsilon$. Intuitively, if we say that Y is a child of X in $Y \cdot X$ and a sibling of X in $X \parallel Y$, then a constant is only allowed to communicate with its siblings, but not with its aunts or nieces. Communication with, say, aunts, can be allowed by adding productions of the form $(M_1 \cdot N_3) \parallel L_1 \xrightarrow{c(x,y)} (M_2 \cdot N_3) \parallel L_2$. In a similar way we could allow communication with grand-aunts etc.. However, communication with arbitrary ancestors is not possible, since it requires an infinite number of productions.

Global channels have the same advantages and disadvantages as global variables. Global variables can be useful and reduce the size of code, but they can also be dangerous, since badly programmed procedures may have undesired side-effects which modify the values of global variables, when they should not. Similarly, global channels can be useful, but they allow any process to communicate with any other, whether that is intended or not.

However, global channels have another specific disadvantage: They make flow-graphs Turing-powerful, which implies that all interesting verification problems are undecidable. Even though we have not formally defined the concept of global channel, let us formulate this as a theorem and sketch a proof.

Proposition 3. *Flow-graphs with global communication channels are Turing-powerful.*

Proof. The proof shows that counter machines, a Turing-powerful model (see for instance Section 8.5.3 of [16]), can be simulated by flow-graphs with global channels. Counter machines consist of counters and a finite control. The control can increase or decrease the counters by one unit, and it can test if they are zero (i.e., it can decide which action to perform next according to whether a counter is zero or not). We simulate a counter by means of the flow-graph shown in Figure 11. The channels are *inc*, *dec*, *zero*, and *nonzero* with the obvious meanings: increase the counter, decrease, and test for zero. Since the values sent or received during communication are irrelevant, we write *inc*? instead of *inc*?x, and similarly for the other channels. The number stored in the counter is simulated by the number of calls of the procedure C which have not terminated yet. Observe that a communication on channel *zero* is only possible if this number is 0. The control of a particular counter machine can be easily simulated by a flow-graph which communicates with the counters through their channels (the names of the channels need renaming so that the sets of channels of any two counters are disjoint). The final flow-graph consists of the procedures C_{0i}, C_i for all the counters, the procedure for the finite control, say *Control*, and a main procedure which just calls $C_{01}, \ldots, C_{0n}$, and *Control* in parallel.

Globality of channels is essential for this proof: If channels are local, the *Control* procedure can communicate with C_0 but not with C, and so in particular the counter cannot exceed the value 1.

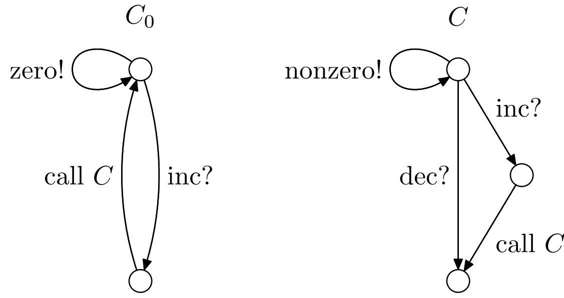

Fig. 11. A flow-graph simulating a counter

In contrast to this result, there are interesting verification problems which are decidable for flow-graphs with local channels. We prove this by translating the flow-graphs into PRSs and applying known results. In particular, we have the following theorem:

Proposition 4. *Given a flow-graph with local channels and one of its nodes, the existence of a computation that visits this node is a decidable problem.*

Proof. We first translate the flow-graph into a PRS. The node, say n, which we wish to visit becomes a process constant N. The initial node of the main procedure becomes another constant, say I. We add to the PRS a production $N \xrightarrow{yes} \epsilon$, where *yes* is a new action not appearing anywhere else in the PRS. Computations that visit n correspond to paths in the transition system of the PRS that start at I and contain an edge labelled by *yes*. Mayr has shown that the existence of these paths can be reduced to solving a number of instances of the word problem for Petri nets and for pushdown automata, both of which are known to be decidable [22].

6 Flow-graph analysis using PRSs

We have shown that PRSs provide a simple semantics to sophisticated control mechanisms, like procedures, spawning of processes, multiple threads, and synchronisation. We still have to show that the theoretical results behind the process hierarchy can be applied to control flow analysis. This has been the subject of a number of papers, whose results we now survey.

(S, S)*-PRSs:* The class of PRSs most studied so far is (S, S), corresponding to pushdown automata, and including the PRSs modelling flow-graphs with procedure calls. Given a PRS Δ over sets *Cons* and ACt, we say that a process term t' is *reachable* from another term t if the transition system of the PRS contains a path leading from t to t'. The following theorem is the heart of all the work done on this class:

Theorem 1. *Let Δ be a (S, S)-PRS over a set Cons of constants, and let $L \subseteq Cons^*$ be a regular set. The set of terms reachable from L (the* successors *of L) is regular, and an automaton recognising this set can be effectively constructed from an automaton recognising L.*[1] *The same result holds for the set of terms from which L can be reached (the predecessors of L).*

This result has an interesting history. It was first proved by Büchi in 1964, where he expressed it in terms of regular canonical systems [4]; it can also be found in Chapter 5 of his unfinished book [5]. The result has been rediscovered at least twice: Book and Otto prove it for monadic rewriting systems in [3], while Caucal proves it for prefix rewriting [7].

In [12] it was shown that this theorem leads to new solutions for a variety of classical control-flow analysis problems, like comptuting the sets of live variables or very busy expressions. In order to convey the main idea behind the solutions, let us sketch the procedure for a somewhat simpler problem, namely that of determining if a program variable v needs to be initialised. This is the case if some execution of the flow-graph uses the value of v (i.e., executes an assignment with v on the right-hand-side or checks a boolean condition involving v) before v is assigned a value.

Since we have given flow-graphs a PRS semantics, this problem can now be reformulated in PRS terms. We assume that the set *Act* contains two subsets of actions, a set *Use* containing the actions that use the variable v, and a set *Def* containing the actions that assign values to it. Let C_{Use} be the set of process constants to which productions with actions in *Use* can be applied. Then, v needs to be initialised if the transition system of the PRS contains a path of the form

$$I \xrightarrow{a_1} t_1 \xrightarrow{a_2} t_2 \ldots t_{n-1} \xrightarrow{a_n} N \cdot t'$$

where I is the constant corresponding to the initial node of the main procedure, $a_1, \ldots, a_n \in (Act - Def)$, and $N \in C_{Use}$.

Now, we proceed as follows. We first compute the set of process terms reachable from I by means of productions whose actions belong to $Act - Def$. This set can be infinite, but by Theorem 1 we can compute a finite automaton that recognises it. It is then easy to check if the automaton accepts some term of the form $N \cdot t'$.

In order to make Theorem 1 the basis of competitive systems for flow-graph analysis it is necessary to develop efficient algorithms for the computation of the automata. This was done in [11] for (S, S)-PRSs in which the left-hand-side of a production has length at most 2. The time and space complexity of the procedures must be analysed in detail. Let n_C and n_Δ be the number of constants and productions of the PRS, and let n_Q, n_δ be the number of states and transitions of the automaton accepting L. Then, an automaton recognising all successors of L can be computed in $O(n_C n_\Delta (n_Q + n_\Delta) + n_C n_\delta)$ time and space. An automaton recognising all predecessors of L can be computed in $O(n_Q^2 n_\Delta)$ time and $O(n_Q n_\Delta + n_\delta)$ space. For (S, S)-PRSs which model flow-graphs, we

[1] Here we are identifying the term $X_1 \cdot \ldots \cdot X_n$ and the word $X_1 \ldots X_n$.

can measure the complexity in terms of the numbers n_P and n_E of procedures and edges, respectively. An automaton recognising all successors of L can be computed in $O(n_E(n_Q+n_P)+n_\delta)$ time and space, and an automaton recognising all predecessors in $O(n_Q^2 n_E)$ time and $O(n_Q n_E + n_\delta)$ space.[2]

Paper [11] contains experimental results on an implementation, not only of the algorithms for the computation of predecessors and successors, but also of a model-checker for (S,S)-PRSs based on them. The checker allows to uniformly solve analysis problems by formulating them in the temporal logic LTL.

$(1,G)$-PRSs: The class $(1,G)$, which includes the PRS models of flow-graphs with procedure calls, parallelism and process creation, but without synchronisation, has also been studied. Lugiez and Schnoebelen observed in [20] that Büchi's result, which holds for the class $(1,S)$ as a subclass of (S,S), could be generalised to $(1,G)$ in a very elegant way. In the case of $(1,S)$, terms are made out of constants and sequential composition, and so a set of terms can be identified with a set of words over the alphabet of process constants. This is no longer possible for $(1,G)$, because terms now contain both sequential and parallel composition. But a set of terms can still be identified with a set of trees, namely the syntax trees of the terms.[3] So it makes sense to speak of a regular set of terms as a set of terms recognised by a tree automaton. Lugiez and Schnoebelen show that Büchi's result still holds: Given a $(1,G)$-PRS and a regular set of terms L, the sets of predecessors and successors of L are regular, and tree automata recognising them can be effectively computed from a tree automaton recognising L. However, just as for the (S,S) class, it is necessary to provide efficient algorithms for the computation of the tree automata. This was done in [14], and the paper also shows how to apply the algorithms to common program analysis problems. However, in this case the algorithms have not been efficiently implemented yet.

It is interesting to note that more involved analysis problems are undecidable for flow-graphs with procedure calls and parallelism, but without synchronisation. Müller-Olm and Seidl have recently shown that this is the case for optimal slicing [24].

(P,G)-PRSs: The class (P,G), which includes the PRS models of flow-graphs which procedure calls, parallelism, and synchronisation through local channels has been less studied. Mayr shows in [21] that the reachability problem and a generalisation of it, the reachable property problem, are decidable. These results can be used to prove decidability of some control-flow problems. Proposition 4

[2] These are results for procedures which do not return a value. See [1] for a more detailed analysis in a similar model.

[3] There is a technical subtlety here. In the definition of PRS we work with a structural congruence stating that parallel composition is associative and commutative. This is important in the presence of synchronisation: Otherwise the production $X \parallel Y \xrightarrow{a} Z$ would be applicable to $(X \parallel Y) \parallel V$ but not to $X \parallel (Y \parallel V)$. However, in the class $(1,G)$ the left-hand-side of a production is always a single constant, and this problem does not appear. It is then possible to work directly with terms instead of with equivalence classes of terms.

shows that the reachability of program points is decidable, and the same holds for other simple problems like Possible Rendezvous (determining if two given statements $c?x$ and $c!y$ may synchronise with each other), or May Happen in Parallel (determining if two given statements may ever execute in parallel).

These results are in sharp contrast with undecidability results recently presented in the literature. Ramalingam shows in [25] that even the simplest analysis problems, like reachability of a point in a program, are undecidable for a set of Ada communicating procedures each of which may call itself recursively. In Ada, a concurrent program is a statically specified set of tasks. Each task is essentially a collection of sequential and possibly recursive procedures, with a distinguished main procedure. The only concurrency primitive he uses for the undecidability proof is rendezvous communication: a task T may *invoke* an *entry* $S.E$ belonging to a task S; S may *accept* this entry. So, in our language of flow-graphs, if T wants to communicate with S, then it does so by means of a channel named S, which is globally and statically defined before the computation starts. So, apart from technicalities, Ramalingam's result is the one we sketched in Proposition 3. Notice that neither process creation nor mutual recursion are required for the proof: It suffices to have three processes, two of which call themselves recursively. However, global channels are essential. Notice that Proposition 4 shows decidability for a model with local channels only, but allowing for process creation and mutual recursion with unrestricted interplay between the two.

The complexity of analysis problems for the (P, G) class is necessarily high. The reason is that it properly includes the class (P, P), which corresponds to place/transition Petri nets. Since all interesting analysis problems for Petri nets are know to be EXPSPACE-hard, there is no hope of finding algorithms with a good worst-case complexity. However, a bad worst-case complexity does not necessarily mean that analysis algorithms for this class will not be useful in practice. For instance, there has been recent success in the analysis of a model called broadcast protocols [13], which also contains Petri nets as special case. Delzanno has implemented an algorithm and used it to automatically verify safety properties of many different cache-coherence protocols in a few seconds [9]. Further work on the (P, G) class is needed.

7 History and related work

The history of process rewrite systems and its applications to program analysis is not easy to reconstruct, because the researchers involved in it have communicated intensely and have had a variety of motivations.

The idea of using grammars to describe transition systems can be traced back to Caucal [7], who uses pushdown automata to define and study a class of graphs. Caucal presents pushdown automata as (S, S)-PRSs.

A grammar-based view of Petri nets is presented by Huynh in [17, 18]. He calls his model "commutative grammars", and studies both the context-free and the general case, which correspond to the classes (P, P) and $(1, P)$, respectively. However, he still looks at this grammars mostly as language generators. Other

authors have studied these classes as generators of transition systems, with the classes presented through their machine characterizations. For instance, in [8] $(1, P)$ is presented as a subset of the CCS process algebra, while in [15, 10] it is presented as a subclass of Petri nets.

The idea of integrating different classes of transition systems into a unified model by means of a Chomsky-like hierarchy can also be traced back to [7]. However, only the sequential case is considered in his paper. Moller introduces two different hierarchies in [23], one for the sequential and one for the parallel case. The process hierarchy as presented here, allowing for mixed terms with both sequential and parallel combinators, was introduced by Mayr in his Ph.D. Thesis [21] and later published in [22].

The decidability and complexity of verification problems for the classes of the hierarchy has been the subject of many papers. Surveys (a bit outdated) can be found in [23] and [6].

The idea of applying PRSs to model the semantics of flow-graphs was first presented in [12]. However, it builds upon ideas by Steffen [29]—later refined by Steffen and Schmidt [27, 26]—on the relationship between program analysis and model-checking.

The program analysis algorithms of Section 6 can be seen as a development of Büchi's result on canonical systems, but also as an automata-theoretic approach to program analysis problems. A different approach based on fixpoint techniques has been introduced and developed by Knoop, Müller-Olm, Seidl, Steffen, and others (see for instance [19, 28]). This approach predates the automata-theoretic. Today the two approaches influence each other.

8 Conclusions

We have introduced Process Rewrite Systems, a process algebra having strong connections with well known computational models like pushdown automata and Petri nets. PRSs have not been defined by an individual or a few individuals; instead, they are the result of a synthesis process in which results were obtained by a large number of people, and then a model able to encompass them was defined. We have argued that PRSs are an excellent interface between strong theoretical results and applications, with program analysis playing a distinguished rôle.

Grzegorz, I hope you enjoyed the talk!

References

1. R. Alur, K. Etessami, and M. Yannanakis. Analysis of recursive state machines. In *Proceedings of CAV '01*, LNCS 2102:207–220, 2001.
2. J. C. M. Baeten and W. P. Weijland. *Process Algebra. Cambridge Tracts in Theoretical Computer Science*, vol. 18, Cambridge University Press, 1990.
3. R. Book and F. Otto. *String-rewriting Systems*. Springer-Verlag, 1993.
4. J. R. Büchi. Regular canonical systems and finite automata. *Arch. Math. Logik Grundlagenforschung*, 6:91–111, 1964.
5. J. R. Büchi and D. Siefkes. *Finite Automata, Their Algebras and Grammars*. Springer-Verlag, 1988.

6. O. Burkart and J. Esparza. More infinite results. *EATCS Bulletin*, 62:138–159, 1997.
7. D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61–86, November 1992.
8. S. Christensen, Y. Hirshfeld, and F. Moller. Bisimulation equivalence is decidable for all basic parallel processes. In *Proceedings of CONCUR '93*, LNCS 715:143–157, 1993.
9. G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings of CAV '00*, LNCS 1885:53–68, 2000.
10. J. Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. *Fundamenta Informatica*, 31:13–26, 1997.
11. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of CAV '00*, LNCS 1885:232–247, 2000.
12. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. *Proceedings of FOSSACS '99*, LNCS 1578:14–30, 1999.
13. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proceedings of LICS '99*, 352–359. IEEE Computer Society, 1999.
14. J. Esparza and A. Podelski. Efficient algorithms for pre and post on interprocedural parallel flow graphs. In *Proceedings of POPL '00*, 1–11, ACM Press, 2000.
15. Y. Hirshfeld. Petri nets and the equivalence problem. In *Proceedings of CSL '93*, LNCS 832:165–180, 1994.
16. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.
17. D.T. Huynh. Commutative grammars: The complexity of uniform word problems. *Information and Control*, 57(1):21–39, 1983.
18. D.T. Huynh. The complexity of equivalence problems for commutative grammars. *Information and Control*, 66(1/2):103–121, 1985.
19. J. Knoop. *Optimal Interprocedural Program Optimization*. LNCS 1428, 1998.
20. D. Lugiez and Ph. Schnoebelen. The regular viewpoint on PA-processes. In *Proceedings of CONCUR '98*, LNCS 1466:50–66, 1998.
21. R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*. Ph.D. thesis, Technische Universität München, 1998.
22. R. Mayr. Process rewrite systems. *Information and Computation*, 156(1/2):264–286, 2000.
23. F. Moller. Infinite results. In *Proceedings of CONCUR '96*, LNCS 1119:195–216, 1996.
24. M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *Proceedings of STOC '01*, 647–656, ACM Press, 2001.
25. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 2000.
26. D.A. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *Proceedings of SAS '98*, LNCS 1503:351–380, 1998.
27. D.A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Proceedings of POPL '98*, 38–48, ACM Press, 1998.
28. H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. In *Proceedings of ESOP '00*, LNCS 1782:351–365, 2000.
29. B. Steffen. Data flow analysis as model checking. In *Proceedings of TACS '91*, LNCS 526:346–364, 1991.