Masaryk University
Faculty of Informatics

Models of Infinite-State Systems with Constraints

Master's Thesis

Jan Strejček

April 2001

Declaration

I declare that this thesis was written by myself and all presented results are my own if not stated otherwise.

Some of the material has been published in [Str00a, Str00b].

Acknowledgment

First of all, I would like to thank Antonín Kučera for his encouragement, valuable discussions and reading the draft. I thank Mojmír Křetínský for his constant willingness to listen, discuss and help.

Warm thanks to Adriana for reading the draft of my thesis and simply being with me.

Thanks are also due to all members of our ParaDiSe laboratory staff for their tolerance and to all people which help me with my awful English. Finally, I want to thank Uni, the Etruscan goddess of the cosmos.

Abstract

We extend a widely used concept of rewrite systems with a mechanism for computing with partial information in a form similar to the one used in concurrent constraint programming. We present how this extension changes the expressive power of rewrite systems classes which are included in Mayr's PRS hierarchy [May97b]. The new classes (fcBPA, fcBPP, fcPA, fc-PAD, fcPAN, fcPRS) are described and inserted into the hierarchy.

Key words

concurrency, process rewrite system, bisimulation equivalence, language expressibility, partial information, constraint system

Contents

Chapter 1

Introduction

Various principles of communication between processes and sharing information in general, are traditional fields of study in theoretical computer science. Computing with partial information in connection with the idea of concurrency is an extensively studied problem in this research area as it corresponds to many situations occurring in a real word.

One of the most successful applications of the ideas of concurrency and computing with partial information has led to Concurrent Constraint Programming (CCP) presented by Saraswat [Sar89] and consequently studied also by Rinard, Panangaden, de Boer, Palamidessi and others (see Bibliography for more details). In CCP processes work concurrently with a shared *store*, which is seen as a constraint on the values that variables can represent. In any state of the computation, the store is given by the constraint established until that moment. CCP provides two primitive operations to deal with the store, *tell* and *ask*. The execution of the *tell* operation adds a constraint to the current store (*tell* can be executed under the condition that the store remains *consistent*, i.e. there must exist some valuation of variables which satisfies the constraint in the store). The *ask* action can be seen as a test on the store – it can be executed only if the current store is strong enough to *entail* a specified constraint. If this is not the case, then the process suspends (waiting for the store to accumulate more information by contributions of the other processes). The execution of *ask* itself leaves the store unchanged, while the execution of *tell* action can only add information to the store. Thus the store evolves monotonically during the computation, i.e. the set of possible values for variables shrinks.

Operational semantics of concurrent systems is traditionally modelled by labelled transition systems. For CCP such an operational semantics was given by Saraswat in [Sar89]. Caucal [Cau92] presents an elegant classification of transition systems using families of sequential rewrite systems defined by restrictions on rewrite rules related to Chomsky hierarchy. Caucal's classification has been generalised by Moller [Mol96] to both parallel and sequential rewrite transition systems. Moller's approach was generalised by Mayr [May97b], he defines the dynamics for rewrite systems using sequential and parallel composition together. The resulting model is called *process rewrite systems*.

We transfer some principles of CCP to process rewrite systems. Previously, we have introduced an analogous modification of purely sequential and purely parallel rewrite systems in [Str00a, Str00b]. In both cases, the aim is to characterise the changes of expressive power of these systems. The mechanism of rewrite systems is extended with the store, which can contain some partial information. We keep talking about constraints (as the theory around CCP) although we do not specify the shape of partial information as sharply as CCP does. We add two constraints to every standard rewrite rule. The rule can be applied only if the actual store is strong enough to entail the first constraint. The second constraint is added to the store when the extended rule is used (the rule is applicable under condition that the store keeps consistent). After application of the rule the store contains the same or more information, thus we say that the store is monotonic. Extended process rewrite systems are called *process rewrite systems with finite constraint systems*.

The comparison between the original process rewrite systems and process rewrite systems with added finite constraint system gives some interesting results. At first, the expressive power of finite state systems, pushdown processes, and Petri nets does not change by adding the store. A more interesting result is that the expressive power of process rewrite systems corresponding to transition systems of classes BPA, BPP, PA, PAD, and PAN strictly increases, hence some new classes of transition systems are obtained in this way.

1.1 Plan of the thesis

The rest of the thesis is structured as follows. The next chapter recalls various definitions widely used in concurrency theory. In Chapter 3 we summarise Mayr's results, especially the definition of process rewrite system and we also present the hierarchy of such systems obtained by imposing various restrictions on the form of rewrite rules. In Chapter 4 we define the notion of process rewrite systems with finite constraint systems and we present the fcPRS-hierarchy. The new classes fcBPA, fcBPP, fcPA, fcPAD, fcPAN, and fcPRS are introduced in this chapter. The strictness of the fcPRS-hierarchy is proven in Chapter 5 focused on the new classes. The thesis closes with a chapter summarising our results and pointing out some directions for future research.

Chapter 2

Basic definitions

In this chapter we recall the notions of labelled transitions systems, language generated by such system, and bisimulation equivalence.

2.1 Labelled transition systems

Concurrent systems are traditionally modeled as edge-labelled directed graphs, whose nodes represent the states which can be entered by a system, and whose edges are labelled with atomic actions. An edge leading from a node s_1 to a node s_2 that is labelled with an action a represents the fact that if the system is in the state s_1, then it can do action a and will be in the state s_2 afterwards.

A precise definition is given below.

Definition 2.1. *A* labelled transition system (LTS) $\mathcal{L}$ *is a tuple* $(S, Act, \longrightarrow, \alpha_0)$*, where*

- S *is a set of* states *or* processes*,*
- Act *is a set of* atomic actions *or* labels*,*
- $\longrightarrow \subseteq S \times Act \times S$ *is a* transition relation*, written* $\alpha \stackrel{a}{\longrightarrow} \beta$ *instead of* $(\alpha, a, \beta) \in \longrightarrow$*,*
- $\alpha_0 \in S$ *is a distinguished* initial state.

A state $\alpha \in S$ *is* terminal *(or* deadlocked*, written* $\alpha \not\longrightarrow$*) if there is no* $a \in Act$ *and* $\beta \in S$ *such that* $\alpha \stackrel{a}{\longrightarrow} \beta$*.*

Our notion of a labelled transition system differs from the standard definition of a (nondeterministic) finite-state automaton (as for example the one given in [HU79]) in two aspects. First, both the set of states and the set of actions can be infinite. Second difference is an absence of final states as we do not distinguish between successful and unsuccessful termination.

The transition relation $\longrightarrow$ can be homomorphically extended to finite sequences of actions $\sigma \in Act^*$ so as to write $\alpha \stackrel{\varepsilon}{\longrightarrow} \alpha$ and $\alpha \stackrel{a\sigma}{\longrightarrow} \beta$ whenever $\alpha \stackrel{a}{\longrightarrow} \gamma \stackrel{\sigma}{\longrightarrow} \beta$ for some state γ. The set of states α such that $\alpha_0 \stackrel{\sigma}{\longrightarrow} \alpha$ for the initial state α_0 and some $\sigma \in Act^*$ is called the set of *reachable* states.

If an LTS is finite then it can be finitely described. In computer science and also in other domains, there are many situations corresponding to infinite transition systems (e.g. algorithms working on arbitrarily large natural numbers). Formal models like Petri nets, pushdown automata and process algebras are able to describe certain classes of infinite transition systems in a finite way. As we shall see in Chapter 3, the class of transition systems definable by process rewrite systems is even larger than all mentioned classes.

2.2 Language equivalence and bisimilarity

An important question in the realm of concurrency theory is to determine when two transition systems are to be considered "the same". It turned out that the isomorphism is a too strong equivalence. A plethora of finer eqivalences was defined by many people in eighties, an overview of these equivalences was compiled by van Glabbeek [vG90]. We define just two of them, language equivalence and bisimulation equivalence.

Given a labelled transition system $\mathcal{L}$ with the initial state α_0, we can define its *language* $L(\mathcal{L})$ to be the language generated by its initial state α_0, where the language generated by a state is defined in the usual way as the set of all sequences of labels associated with transitions leading from the given state to a terminal state.

Definition 2.2. *The* language *generated by the labelled transition system $\mathcal{L}$ is the set $L(\mathcal{L}) = L(\alpha_0)$, where*

$$L(\alpha) = \{w \in Act^* \mid \alpha \stackrel{w}{\longrightarrow} \beta \text{ for some terminal state } \beta\}.$$

States α and β of the system $\mathcal{L}$ are language equivalent, *written $\alpha \sim_L \beta$, iff they generate the same language, i.e. $L(\alpha) = L(\beta)$.*

Language equivalence is generally taken to be too coarse in the framework of concurrency theory. The second presented equivalence, *bisimulation equivalence*, is perhaps the finest behavioural equivalence studied. Bisimulation equivalence was defined by Park [Par81] and used by Milner [Mil80, Mil89] in his work on CCS. Its definition is as follows.

Definition 2.3. *A binary relation $\mathcal{R}$ on states of labelled transition system is a* bisimulation *iff whenever $(\alpha, \beta) \in \mathcal{R}$ we have that*

- *if $\alpha \stackrel{a}{\longrightarrow} \alpha'$ then $\beta \stackrel{a}{\longrightarrow} \beta'$ for some β' with $(\alpha', \beta') \in \mathcal{R}$,*

- *if* $\beta \xrightarrow{a} \beta'$ *then* $\alpha \xrightarrow{a} \alpha'$ *for some* α' *with* $(\alpha', \beta') \in \mathcal{R}$.

α *and* β *are* bisimulation equivalent *or* bisimilar*, written* $\alpha \sim \beta$*, iff* $(\alpha, \beta) \in \mathcal{R}$ *for some bisimulation* $\mathcal{R}$*.*

This definition can be extended to states in different transition systems by putting them "side by side" and considering them as a single transition system. The binary relation $\sim$ defined above is called *bisimulation equivalence* as it is an eaquivalence and even the largest bisimulation.

Bisimulation equivalence has an elegant characterisation in terms of certain two-player games presented by Stirling [Sti95].

Chapter 3

Process rewrite systems (PRS)

In this chapter we summarise (and slightly modify) the first part of Mayr's paper titled "Process Rewrite Systems" [May97b].

The process rewrite systems represent a very general term rewriting formalism that covers many widely known models like Basic Parallel Processes (BPP), context-free processes (BPA), pushdown processes, process algebras (PA), Petri Nets, and provides an unified view of these models.

3.1 Process terms

The process terms are the cornerstone of process rewrite systems. They correspond to states of transition systems described by process rewrite systems.

Definition 3.1. *Let $Const = \{X, Y, Z, \cdots\}$ be a countably infinite set of* process constants*. The* process terms *that describe states of the system have the form*

$$t = \varepsilon \mid X \mid t_1.t_2 \mid t_1 \| t_2,$$

where ε is the empty term, $X \in Const$ is a process constant (used as an atomic process in this context), "$\|$" means parallel composition, and "." means sequential composition. Let $\mathcal{T}$ be the set of process terms.

We always work with equivalence classes of terms modulo commutativity and associativity of parallel composition and modulo associativity of sequential composition. Also we define that $\varepsilon.t = t = t.\varepsilon$ and $t\|\varepsilon = t$.

Although we have declared that sequential composition is associative, when we look at terms we think of it as left-associative. So when we say that a term t has the form $t_1.t_2$, we mean that t_2 is either a single constant or a parallel composition of process terms.

Definition 3.2. *The* size *of a process term is the number of occurrences of constants in it plus the number of occurrences of operators in it.*

$$\begin{array}{rcl} size(\varepsilon) & = & 0 \\ size(X) & = & 1 \\ size(t_1.t_2) & = & size(t_1) + size(t_2) + 1 \\ size(t_1 \| t_2) & = & size(t_1) + size(t_2) + 1 \end{array}$$

Definition 3.3. *The set $Const(t)$ is the set of all constants that occur in a process term t.*

$$\begin{array}{rcl} Const(\varepsilon) & = & \emptyset \\ Const(X) & = & \{X\} \\ Const(t_1.t_2) & = & Const(t_1) \cup Const(t_2) \\ Const(t_1 \| t_2) & = & Const(t_1) \cup Const(t_2) \end{array}$$

3.2 Definition of PRS

Now we are ready to introduce the syntax of *process rewrite systems*. The semantics will be explicitly given a bit later, when we show how to associate a labelled transition system to each process rewrite system.

Definition 3.4. *Let $Act = \{a, b, \cdots\}$ be a countably infinite set of* atomic actions. *A* process rewrite system (PRS) Δ *is a pair (R, t_0), where*

- *R is a finite set of* rewrite rules, *which are of the form $t_1 \stackrel{a}{\longrightarrow} t_2$*[1]*, where $t_1, t_2 \in \mathcal{T}$ are process terms and $a \in Act$ is an atomic action,*
- *$t_0 \in \mathcal{T}$ is an* initial state.

Slightly abusing our notation, we usually write $(t_1 \stackrel{a}{\longrightarrow} t_2) \in \Delta$ instead of $(t_1 \stackrel{a}{\longrightarrow} t_2) \in R$, where $\Delta = (R, t_0)$.

For a given Δ with the initial state t_0, the set $Const(\Delta)$ is naturally defined as the set of all constants that occur in rewrite rules or initial state.

$$Const(\Delta) = Const(t_0) \cup \bigcup_{(t_1 \stackrel{a}{\longrightarrow} t_2) \in \Delta} (Const(t_1) \cup Const(t_2))$$

Similarly, the set $Act(\Delta)$ is the set of all actions that occur in rewrite rules of Δ.

$$Act(\Delta) = \bigcup_{(t_1 \stackrel{a}{\longrightarrow} t_2) \in \Delta} \{a\}$$

[1] There is an ambiguity around the expression $t_1 \stackrel{a}{\longrightarrow} t_2$ as it can be seen as a rewrite rule as well as an element of a transition relation. Fortunately, the actual meaning is always determined by the context.

Due to the finiteness of Δ, the sets $Const(\Delta)$ and $Act(\Delta)$ are both finite.

Each process rewrite system induces an unique labelled transition system that represents its dynamics. A formal definition follows.

Definition 3.5. *Let $\Delta = (R, t_0)$ be a process rewrite system. The LTS $\mathcal{L}$ induced by Δ has the form $(S, Act(\Delta), \longrightarrow, t_0)$, where*

- $S = \{t \in \mathcal{T} \mid Const(t) \subseteq Const(\Delta)\}$ *is the set of states,*
- *transition relation $\longrightarrow$ is defined as the least relation that satisfies the inference rules*

$$\frac{(t_1 \xrightarrow{a} t_2) \in \Delta}{t_1 \xrightarrow{a} t_2}, \qquad \frac{t_1 \xrightarrow{a} t_1'}{t_1 \| t_2 \xrightarrow{a} t_1' \| t_2}2, \qquad \frac{t_1 \xrightarrow{a} t_1'}{t_1.t_2 \xrightarrow{a} t_1'.t_2},$$

  where $t_1, t_2, t_1' \in \mathcal{T}$,

Since the set of rewrite rules in Δ is finite, the generated LTS is finitely branching. (For some classes of systems (e.g. Petri nets) the branching-degree is bounded by a constant that depends on Δ. For other classes (e.g. PA) the branching-degree is finite at every state, but it can get arbitrarily large.) On the other hand, the generated transition system can be infinite.

We often speak about "process rewrite system" meaning "labelled transition system generated by process rewrite system".

The definition of process rewrite systems is more general than the definition of rewrite systems presented by Caucal [Cau92] which takes into account only systems with sequential composition, and also than the one given by Moller [Mol96] which takes into account only purely sequential and purely parallel rewrite systems and which was also used to form the hierarchy of standard process classes. Process rewrite systems provide a general and unifying framework which naturally subsumes all of the abovementioned formalisms.

3.3 PRS-hierarchy

Many common models of systems fit into the scheme of process rewrite systems. In this section we characterise some interesting subclasses of rewrite systems. At first, we need to define some classes of process terms.

Definition 3.6. *We distinguished four* classes of process terms.

"1" *Terms consisting of a single process constant like X.*

[2]Note that parallel composition is commutative and, thus, the inference rule for parallel composition also holds with t_1 and t_2 exchanged.

"S" *Terms consisting of a single constant or a sequential composition of process constants like $X.Y.Z$.*

"P" *Terms consisting of a single constant or a parallel composition of process constants like $X\|Y\|Z$.*

"G" *General process terms with arbitrarily nested sequential and parallel compositions like $(X.(Y\|Z))\|W$.*

We also let $\varepsilon \in S, P, G$, but $\varepsilon \notin 1$.

The relationship between these classes of process terms is easy to see: $1 \subsetneq S$, $1 \subsetneq P$, $S \subsetneq G$, and $P \subsetneq G$. S and P are incomparable and $S \cap P = 1 \cup \{\varepsilon\}$.

The expressiveness of a rewrite system depends on what kind of terms are on the left-hand side and the right-hand side of rewrite rules in Δ. Thus the subclasses of process rewrite systems are characterised by the classes of terms allowed on the left-hand and the right-hand side of the rewrite rules.

Definition 3.7. *Let $\alpha, \beta \in \{1, S, P, G\}$. A process rewrite system $\Delta = (R, t_0)$ is an (α, β)-*PRS *if $t_0 \in \beta$ and for every rewrite rule $(l \xrightarrow{a} r) \in \Delta$ the term l is in the class α and $l \neq \varepsilon$ and the term r is in the class β (and can be ε iff $\beta \neq 1$). A (G, G)-PRS is simply called* PRS.

It does not have much sense to consider those (α, β)-PRS where α is more general than β or incomparable to β (for example, $\alpha = G$ and $\beta = S$), because the rule $t_1 \xrightarrow{a} t_2$ can generate a transition from a state t only if the term t_1 is a subterm of t. But when the initial state t_0 is taken from the same class as terms which appear at right sides of rewrite rules, the reachable states are of the class β. Only those rules whose left-hand side is taken from the class β (or some subclass) are applicable to reachable states. Thus, we restrict our attention to such (α, β)-PRS where $\alpha \subseteq \beta$.

Without the loss of generality it can be assumed that the initial state t_0 of a (α, β)-PRS is a single constant. There are only finitely many terms $t_1, t_2, \ldots, t_n$ such that $t_0 \xrightarrow{a_i} t_i$. If t_0 is not a single constant then we can achieve this by introducing a new constant X_0 and new rules $X_0 \xrightarrow{a_i} t_i$ and declaring X_0 to be the initial state (the modified system is still (α, β)-PRS).

If a system Δ belongs to a class (α, β)-PRS (where $\alpha \subseteq \beta$), then the set of states of LTS generated by Δ consists only of process terms t of the class β that satisfy $Const(t) \subseteq Const(\Delta)$.

Figure 3.1 shows a graphical description of the hierarchy of (α, β)-PRS, simply called *PRS-hierarchy*. Many of these (α, β)-PRS correspond to widely known models like Petri nets, pushdown processes, context-free processes, and others.

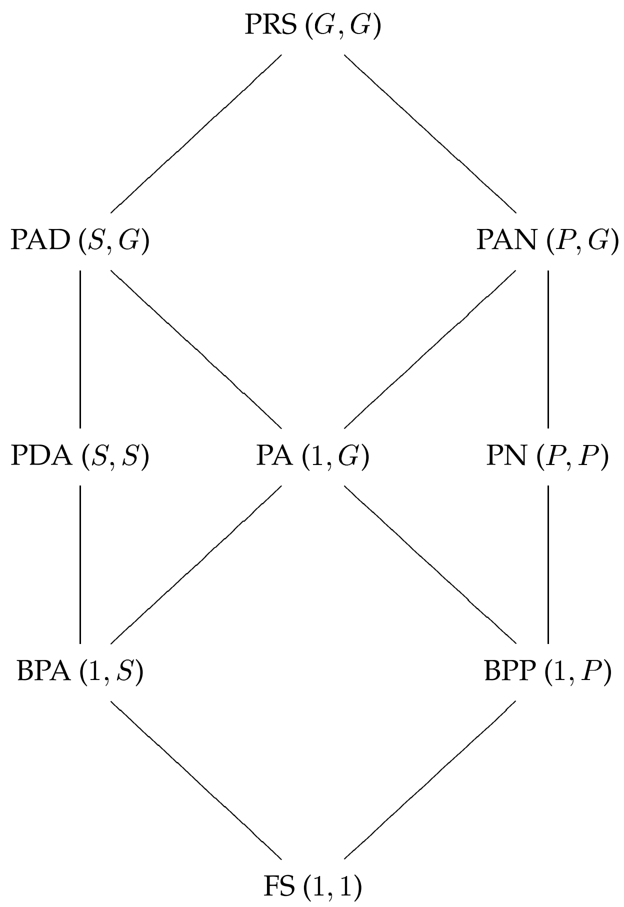


Figure 3.1: The PRS-hierarchy

1. $(1, 1)$-PRS are equivalent to finite-state systems (FS). Every process constant corresponds to a state and the state space is bounded by $|Const(\Delta)|$. Every finite-state system can be encoded as a $(1, 1)$-PRS.

2. $(1, S)$-PRS are equivalent to Basic Process Algebra processes (BPA) defined in [BK85], which are the transition systems associated with Greibach normal form (GNF) context-free grammars in which only left-most derivations are allowed.

3. It is easy to see that pushdown automata can be encoded as a subclass of (S, S)-PRS (with at most two constants on the left-hand side of rules). Caucal [Cau92] showed that any unrestricted (S, S)-PRS can be presented as a pushdown automaton (PDA), in the sense that the transition systems are isomorphic up to the labelling of states. Thus (S, S)-PRS are equivalent to pushdown processes (which are the pro-

cesses described by pushdown automata).

4. (P, P)-PRS are equivalent to Petri nets (PN). Every constant corresponds to a place in the net and the number of occurrences of a constant in a term corresponds to the number of tokens in this place. This is because we work with classes of terms modulo commutativity of parallel composition. Every rule in Δ corresponds to a transition in the net.

5. $(1, P)$-PRS are equivalent to communication-free nets, the subclass of Petri nets where every transition has exactly one place in its preset [BE97]. This class of Petri nets is equivalent to Basic Parallel Processes (BPP) [Chr93].

6. $(1, G)$-PRS are equivalent to PA-processes, Process Algebras with sequential and parallel composition, but no communication (see [BK85] for details).

7. (P, G)-PRS are called *PAN-processes* in [May97a]. It is the smallest common generalisation of Petri nets and PA-processes and it strictly subsumes both of them (e.g., PAN can describe all Chomsky-2 languages while Petri nets cannot).

8. (S, G)-PRS is the smallest common generalisation of pushdown processes and PA-processes. They are called *PAD* (PA + PDA) in [May98].

9. The most general case is (G, G)-PRS (here simply called PRS). PRS have been introduced in [May97b]. They subsume all of the previously mentioned classes.

From our point of view, standard process classes like FS, BPA, BPP, PDA, and PN are considered more generally then for example in [Mol96]. The difference corresponds to the different definition of labelled transition system. For us, every terminal state is successful ("final" in the terms of [Mol96]) and we do not assume that the set of final states is finite.

3.4 Intuition behind the PRS-hierarchy

In this section we explain the intuition behind the design of (α, β)-PRS rules and the respective restricted subclasses of PRS.

If parallel composition is allowed on the right-hand side of rules, then there can be rules of the form $t \xrightarrow{a} t_1 \| t_2$. This means that it is possible to create processes that run in parallel. The rule can be interpreted that, by action a, the process t splits into two independent processes t_1 and t_2.

If sequential composition is allowed on the right-hand side of rules, then there are rules of the form $t \xrightarrow{a} t_1.t_2$. The interpretation is that the process t calls a subroutine t_1 and behaves like the process t_2. It resumes its execution after the subroutine t_1 terminates.

If arbitrary process terms are allowed on the right-hand side of rules then both parallelism and subroutines are possible.

If parallel composition is allowed on the left-hand side of rules, then there are rules of the form $t_1 \| t_2 \xrightarrow{a} t$. This can be interpreted as synchronisation or communication of the parallel processes t_1 and t_2. This is because this action can only occur if both t_1 and t_2 change in a certain defined way.

If sequential composition is allowed on the left-hand side of rules, then there can be rules of the form $t_1'.t_2 \xrightarrow{a} t'$ and $t_1''.t_2 \xrightarrow{a} t''$. The intuition is that the process t called a subroutine t_1 and behaves like t_2 by a rule $t \xrightarrow{a} t_1.t_2$. In its computation the subroutine may reach a state t_1' or t_1''. Now one of these rules is applicable. This means that the result of the computation of the subroutine affects the behaviour of the caller when it becomes active again, since the caller can become t' or t''. The interpretation is that the subroutine returns a value to the caller when it terminates.

If arbitrary process terms are allowed on the left-hand side of rules then both synchronisation and value-passing by subroutines are possible.

3.5 Strictness of the PRS-hierarchy

There is a natural question about the strictness of the PRS-hierarchy. With respect to language expressibility this is not the case. For example, both BPA and PDA define exactly the (ε-free) context-free languages. The situation is different if we ask about strictness with respect to bisimulation equivalence.

It has been proven by Burkart, Caucal, Steffen, and Moller [BCS96, Mol96] that the classes FS, BPP, BPA, PDA, PA, and PN are all different with respect to bisimulation equivalence. For PAD, PAN, and PRS it was demonstrated by Mayr [May97b] using the two rewrite systems below.

Example 3.8. *Consider the following PDA system given as (S, S)-PRS with initial state $U.X$.*

$$\begin{array}{lll}
U.X \xrightarrow{a} U.A.X & U.A \xrightarrow{a} U.A.A & U.B \xrightarrow{a} U.A.B \\
U.X \xrightarrow{b} U.B.X & U.A \xrightarrow{b} U.B.A & U.B \xrightarrow{b} U.B.B \\
U.X \xrightarrow{c} V.X & U.A \xrightarrow{c} V.A & U.B \xrightarrow{c} V.B \\
U.X \xrightarrow{d} W.X & U.A \xrightarrow{d} W.A & U.B \xrightarrow{d} W.B \\
V.X \xrightarrow{e} V & V.A \xrightarrow{a} V & V.B \xrightarrow{b} V \\
W.X \xrightarrow{f} W & W.A \xrightarrow{a} W & W.B \xrightarrow{b} W
\end{array}$$

The system described in Example 3.8 can produce a sequence $\sigma \in \{a, b\}^*$ and then either c followed by σ in the reversed order and finally action e, or d followed by σ in the reversed order and finally action f. Such a system cannot be bisimilar to any PAN system.

Example 3.9. *Consider following Petri net given as (P, P)-PRS with initial state $X \| A \| B$.*

$$\begin{array}{ll} X \xrightarrow{g} X\|A\|B & Y\|A \xrightarrow{a} Y \\ X \xrightarrow{c} Y & Y\|B \xrightarrow{b} Y \\ X\|A \xrightarrow{d} Z & Y\|A \xrightarrow{d} Z \\ X\|B \xrightarrow{d} Z & Y\|B \xrightarrow{d} Z \end{array}$$

The system shown in Example 3.9 can do the action g n-times ($n \geq 0$), then the action c followed by an arbitrary sequence $\sigma \in \{a, b\}^*$ such that the action a occurs $(n + 1)$-times in σ and so does the b. From every non-terminal state the system can also do the action d leading to a deadlocked state. This Petri net cannot be described by any PAD process with respect to bisimulation equivalence.

The systems given by previous two examples prove that the classes PAN and PAD are incomparable. The strictness of the PRS-hierarchy is now obvious.

Chapter 4

PRS with finite constraint systems (fcPRS)

In this chapter we extend the process rewrite systems with finite constraint systems. This extension of rewrite systems provides a way of keeping a sort of global information which is accessible to all parallel threads. It is quite surprising that this extension (which is not so powerful as an extension with a general finite-state control unit which gives Turing power even to PA class) increases the expressive power of classes like PAN and PAD.

The extension is inspired by the idea of common store used in Concurrent Constraint Programming.

4.1 Constraint systems

The state space and possible evolution of the store used by PRS with finite constraint system are described by a constraint system, i.e. a set of constraints with a structure of an algebraic lattice.

Definition 4.1. *A* constraint system *is a bounded lattice* $(C, \leq, \wedge, tt, ff)$*, where* C *is the set of* constraints, $\leq$ *is an ordering on this set,* $\wedge$ *is the lub operation, and* tt (true), ff (false) *are the least and the greatest elements of* C *(*$tt \neq ff$*).*

In algebra, the symbol $\wedge$ usually denotes the *glb* (the greatest lower bound) operation, while *lub* (the least upper bound) operation is rather marked with symbol $\vee$. We adopted the notation used in the framework of CCP, where the lub operation (marked with $\wedge$) corresponds to logical conjunction.

Following the terminology and the notation used in CCP, instead of $\leq$ we refer to its inverse relation, denoted by $\vdash$ and called *entailment*. Formally

$$\forall m, n \in C : m \vdash n \iff n \leq m.$$

We say that a constraint m is *consistent* with a constraint n iff $m \wedge n \neq f\!f$. The state of the store cannot be $f\!f$ as we require the consistency of the store initialised to tt. We use $C^{\diamond}$ to denote $C \setminus \{f\!f\}$.

Two following examples show two constraint systems heavily used in the rest of this thesis.

Example 4.2. *Let* $C = \{tt, f\!f\}$, $\leq = \{(tt, f\!f), (tt, tt), (f\!f, f\!f)\}$. *Then* $\mathcal{C}_{\varepsilon}$ *is the trivial constraint system* $(C, \leq, \wedge, tt, f\!f)$ *depicted in Figure 4.1.*

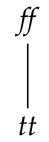


Figure 4.1: Constraint system $\mathcal{C}_{\varepsilon}$

Example 4.3. *Let* $C = \{tt, m, n, f\!f\}$, $\leq = \{(tt, f\!f), (tt, m), (tt, n), (m, f\!f), (n, f\!f)\} \cup \{(o, o) \mid o \in C\}$. *Then* $\mathcal{C}_{mn} = (C, \leq, \wedge, tt, f\!f)$ *is the constraint system depicted in Figure 4.2.*

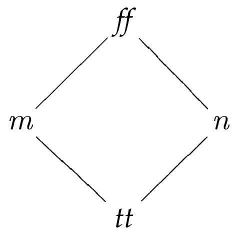


Figure 4.2: Constraint system $\mathcal{C}_{mn}$

We add one more example which can provide a better illustration of the relation between partial information, a constraint system and an evolution of the store.

Example 4.4. *The Herbrand constraint system on* $\{a, b\}$ *with variables* x, y *is diagrammatically represented in Figure 4.3.*

4.2 Definition of fcPRS

At first we define the syntax of PRS with finite constraint system. Similarly to the definition of PRS, the semantics will be given later by a precise definition of LTS generated by fcPRS.

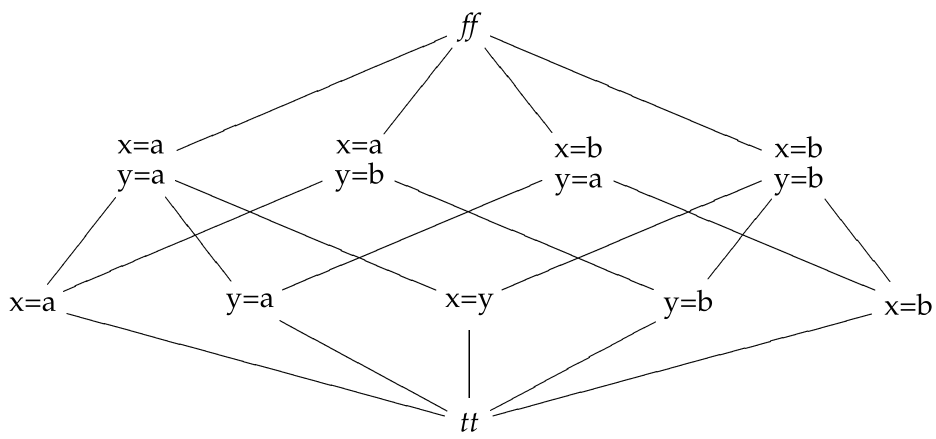


Figure 4.3: Herbrand constraint system on $\{a, b\}$ with variables x, y

Definition 4.5. *A* PRS with finite constraint system Δ *is a tuple* $(\mathcal{C}, R, t_0)$, *where*

- $\mathcal{C} = (C, \leq, \wedge, tt, ff)$ *is a finite constraint system describing the* store*; the elements of* C *represent the* states of the store,
- R *is a finite set of* rewrite rules, *which are of the form* $(t_1 \xrightarrow{a} t_2, m, n)$, *where* $t_1, t_2 \in \mathcal{T}$ *are process terms,* $a \in Act$ *is an atomic action, and* $m, n \in C^\diamond$ *are constraints,*
- t_0 *is a distinguished* initial process term.

Again, instead of $(t_1 \xrightarrow{a} t_2, m, n) \in R$ where $\Delta = (\mathcal{C}, R, t_0)$, we usually write $(t_1 \xrightarrow{a} t_2, m, n) \in \Delta$.

The definitions of $Const(\Delta)$ (which is the set of process constants used in rewrite rules) and $Act(\Delta)$ (the set of actions occurring in rewrite rules) for a given fcPRS Δ with initial process term t_0 are very similar to those which were used for PRS.

$$\begin{aligned} Const(\Delta) &= Const(t_0) \cup \bigcup_{(t_1 \xrightarrow{a} t_2, m, n) \in \Delta} (Const(t_1) \cup Const(t_2)) \\ Act(\Delta) &= \bigcup_{(t_1 \xrightarrow{a} t_2, m, n) \in \Delta} \{a\} \end{aligned}$$

Again, the sets $Const(\Delta)$ and $Act(\Delta)$ are both finite because of the finiteness of Δ.

The next definition exactly describes the labelled transition system defined by a PRS with finite constraint system.

Definition 4.6. *Let $\Delta = (\mathcal{C}, R, t_0)$ be a PRS with finite constraint system $\mathcal{C} = (C, \leq, \wedge, tt, ff)$. The labelled transition system $\mathcal{L}$ induced by Δ has the form $(S, Act(\Delta), \longrightarrow, \alpha_0)$, where*

- *$S = \{t \in \mathcal{T} \mid Const(t) \subseteq Const(\Delta)\} \times C^{\diamond}$ is the set of states,*
- *transition relation $\longrightarrow$ is defined as the least relation that satisfying the inference rules*

$$\frac{(t_1 \xrightarrow{a} t_2, m, n) \in \Delta}{(t_1, o) \xrightarrow{a} (t_2, o \wedge n)} \text{ if } o \vdash m \text{ and } o \wedge n \neq ff,$$

$$\frac{(t_1, o) \xrightarrow{a} (t_1', p)}{(t_1 \| t_2, o) \xrightarrow{a} (t_1' \| t_2, p)},$$

$$\frac{(t_1, o) \xrightarrow{a} (t_1', p)}{(t_1.t_2, o) \xrightarrow{a} (t_1'.t_2, p)},$$

  where $t_1, t_2, t_1' \in \mathcal{T}$ and $m, n, o, p \in C^{\diamond}$,
- *$\alpha_0 = (t_0, tt)$ is the initial state.*

Two important conditions contained in the first inference rule are very close to principles used in Concurrent Constraint Programming (CCP). The first one ($o \vdash m$) ensures that the rule $(t_1 \xrightarrow{a} t_2, m, n) \in \Delta$ can be used only if the actual state of the store o *entails* the constraint m (it is similar to *ask(m)* in CCP). The second condition ($o \wedge n \neq ff$) guarantees that the store keeps *consistent* after application of the rule (analogous to consistency requirement when processing *tell(n)* action in CCP). If these two conditions are satisfied, the meaning of inference rules is the same as in the case of standard process rewrite systems.

An important observation is that the state of the store (starting at *tt*) can move in a lattice $\mathcal{C}$ only in one direction, from *tt* upwards. This can be easily seen from the fact that the actual state of the store o can be changed only by applying some rewrite rule $(t_1 \xrightarrow{a} t_2, m, n) \in \Delta$ and after this application the new state of the store $o \wedge n$ always entails the old state o. Intuitively, the partial information can only be added to the store, not retracted. We say the store has a *monotonic* behaviour, or simply that the store is monotonic.

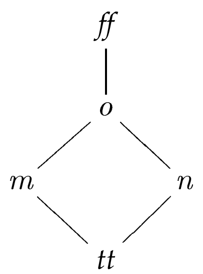
Note that when the system (with o on the store) executes a transition generated by a rewrite rule $(t_1 \xrightarrow{a} t_2, m, n) \in \Delta$ then for every subsequent state of the store p both conditions, $p \vdash m$ and $p \wedge n \neq ff$, are satisfied. The first condition $p \vdash m$ comes from the monotonic behaviour of the store. The second condition comes from the fact that the constraint n in the rule changes the store only in the first application of the rule provided o does not entail n ($o \wedge n \neq o$). All subsequent applications of this rule do not change the store (again thanks to its monotonic behaviour), i.e. for each subsequent state p of the store $p \wedge n = p$ holds. A corollary of this observation is the satisfaction of consistency condition as $p \neq ff$. In other words, once the information presented by n is added to the store, each subsequent attempt to add the same information to the store does not change the store and thus the store keeps consistent.

On the other hand, the fact that some rule is applicable (hence entailment and consistency are satisfiable) does not imply that this rule is applicable forever. The insidious point is the consistency requirement. The store can evolve to a state inconsistent with the second constraint from the rule.

4.3 Intuition behind fcPRS

The intuition behind process rewrite systems with finite constraint systems will be demonstrated on one small example.

Let Δ be a fcPRS given below with initial term $X\|Y\|Z$.



$$
\begin{array}{l}
(X \xrightarrow{a} X, tt, tt) \\
(Y \xrightarrow{b} Y, tt, tt) \\
(X \xrightarrow{x} \varepsilon, tt, m) \\
(Y \xrightarrow{y} Y, tt, n) \\
(Z \xrightarrow{c} Z, o, tt)
\end{array}
$$

At the beginning, the process X can perform the a action without changing the store. The process Y can perform b, also without any changes on the store. The process Z is deadlocked as the rule $(Z \xrightarrow{c} Z, o, tt)$ is applicable only if information on the store implies o. Furthermore, the process X can also perform x, put an information m on the store (and terminate). The process Y can do y and put n on the store. The process Y can do y again and again, but the information on the store increases only after performing the first y. This naturally corresponds to the fact that if one writes on the blackboard the same statement twice, the information written there is not doubled. When constraints m and n have been added to the store (it does not depend what constraint was written to the store first), the store is in the state o corresponding to $m \wedge n$. Then the process Z can start to perform e's.

As you can see, the meaning of constraints in rewrite rules together with the shape of used constraint system can be translated to a human language in a very straightforward way.

4.4 Relationship between PRS and fcPRS

The first information about the relationship between PRS with finite constraint system and standard PRS is provided by the following lemma.

Lemma 4.7. *Labelled transition systems defined by PRS $\Delta' = (R', t_0)$ and by fcPRS $\Delta = (\mathcal{C}_\varepsilon, R, t_0)$ are isomorphic on the assumption that $R' = \{t_1 \xrightarrow{a} t_2 \mid (t_1 \xrightarrow{a} t_2, tt, tt) \in R\}$.*

Proof. Let $\mathcal{L}$ be the transition system corresponding to the fcPRS Δ. The state of the store is tt in every state of $\mathcal{L}$ due to the shape of the trivial constraint system $\mathcal{C}_\varepsilon$ (defined in Example 4.2).

Further, for each rewrite rule $(t_1 \xrightarrow{a} t_2, tt, tt)$ the two conditions in the first inference rule are always satisfied ($tt \vdash tt$ and $tt \wedge tt = tt \neq ff$).

Now we know that the transition system $\mathcal{L}$ is of the form $(S, Act(\Delta), \longrightarrow, (t_0, tt))$, where $S = \{(t, tt) \mid Const(t) \subseteq Const(\Delta)\}$ and the transition relation $\longrightarrow$ can be alternatively defined as $(t, tt) \xrightarrow{a} (t', tt)$ iff there is a transition rule $(t_1 \xrightarrow{a} t_2, tt, tt) \in \Delta$ such that the transition $t \xrightarrow{a} t'$ can be derived from the rewrite rule $t_1 \xrightarrow{a} t_2$ using the inference rules given in Definition 3.5.

If we remove tt from the states of $\mathcal{L}$, we get an isomorphic system $\mathcal{L}'$ which corresponds to the rewrite transition system Δ'. $\square$

Roughly speaking, the lemma says that the trivial constraint system cannot hold any significant information and thus such a fcPRS is isomorphic to the corresponding standard process rewrite system. The lemma can be used in both directions, for proving that any fcPRS of the specified form has an equivalent PRS as well as for constructing a fcPRS equivalent to an arbitrary given PRS.

The following lemma defines another situation when an added constraint system cannot increase expressive power of a process rewrite system.

Lemma 4.8. *For every fcPRS $\Delta = (\mathcal{C}, R, t_0)$ with the rewrite rules of the form $(t_1 \xrightarrow{a} t_2, tt, tt)$, there is an (effectively constructible) PRS Δ' with the transition system isomorphic to the transition system of Δ.*

Proof. We may assume that $\mathcal{C} \neq \mathcal{C}_\varepsilon$ (if $\mathcal{C} = \mathcal{C}_\varepsilon$ then the lemma is a direct corollary of Lemma 4.7).

A crucial step is to observe that the state n of the store cannot be changed by any application of a rewrite rule of the form $(t_1 \xrightarrow{a} t_2, tt, tt)$ as $n \wedge tt = n$. This means that there is no transition between states (t, n) and (t', m) for any $t, t' \in \mathcal{T}$ and $m, n \in C^\diamond$, $m \neq n$.

Another important observation says that applicability of rewrite rules of the specified form does not depend on the current state of the store as necessary conditions are always satisfied because every $m \in C$ entails tt and every $n \in C^\diamond$ is consistent with tt ($n \wedge tt = n \neq ff$). Thus, if there is a transition $(t, tt) \xrightarrow{a} (t', tt)$ then there is also a transition $(t, m) \xrightarrow{a} (t', m)$ for every $m \in C^\diamond$.

The conclusion is that the transition system defined by Δ can be split into $|C^\diamond|$[1] isolated isomorphic parts. Let $\mathcal{L}_m$ denote the part with m on the store for every $m \in C^\diamond$. It is easy to see that if we change the constraint system in Δ to $\mathcal{C}_\varepsilon$, the modified fcPRS describes exactly $\mathcal{L}_{tt}$. From the Lemma 4.7 it follows that we can construct a standard PRS Δ'_{tt} with a transition graph isomorphic to $\mathcal{L}_{tt}$ (i.e. isomorphic to every $\mathcal{L}_m$). The desired process rewrite system Δ' consists of $|C^\diamond|$ copies of Δ'_{tt}. $\square$

Intuitively, the lemma says that if the power of the store is not employed by the rules (we do not add any information to the store), then (without any assumptions on the structure of constraint system) the PRS with constraint system is isomorphic to some standard PRS. The proof also says that the reachable part of the transition system defined by such a fcPRS consists of states with tt on the store.

Although adding a finite constraint system looks like quite weak extension, in the following we will demonstrate that this mechanism can increase expressibility of standard classes of process rewrite systems.

4.5 fcPRS-hierarchy

There are several possibilities how to build the hierarchy of process rewrite systems with finite constraint systems. We can divide fcPRS systems into classes with respect to their constraint systems, rewrite rules (placing restrictions on the form of process terms on the left-hand side and the right-hand side of the rules, conditions on the first and the second constraint in rules), or with respect to some combination of these aspects. We have chosen a combination of the two criterions.

The first criterion is the same as for PRS-hierarchy presented in Section 3.3, i.e. the classes of process terms allowed on the left-hand side and the right-hand side of rewrite rules used in fcPRS.

Definition 4.9. *Let $\alpha, \beta \in \{1, S, P, G\}$ and $\mathcal{C}$ be a constraint system. A fcPRS $\Delta = (\mathcal{C}, R, t_0)$ is (α, β)-*fcPRS *if the initial term $t_0 \in \beta$ and for every rewrite rule $((l, m) \xrightarrow{a} (r, n)) \in \Delta$ the term l is in the class α and $l \neq \varepsilon$ and the term r is in the class β (and can be ε). A (G, G)-fcPRS is simply called* fcPRS.

As in the case of standard PRS, we can make some additional assumptions. We consider only such (α, β)-fcPRS classes where β is more general

[1]We use $|M|$ to denote cardinality of the set M.

than (or equal to) α. Also, without the loss of generality it can be assumed that the initial term t_0 of an (α, β)-fcPRS is a single constant.

If a system Δ belongs to a class (α, β)-PRS (where $\alpha \subseteq \beta$), then the set of states of LTS generated by Δ consists only of pairs (t, m), whose process terms t are of the class β and satisfy $Const(t) \subseteq Const(\Delta)$.

The second criterion for dividing fcPRS systems is a constraint system used by fcPRS. We distinguish only between rewrite systems with the trivial constraint system $\mathcal{C}_\varepsilon$ and rewrite systems with an arbitrary constraint system.

Lemma 4.7 says that any class of (α, β)-fcPRS systems with the trivial constraint system defines the same class of labelled transition systems (up to isomorphism) as (α, β)-PRS class. That is the reason why we can apprehend all (α, β)-PRS classes as classes of fcPRS systems and include them into fcPRS-hierarchy.

We use human-readable abbreviations fcFS, fcBPA, fcBPP, fcPA, fcPDA, fcPN, fcPAD, fcPAN, and fcPRS for classes $(1, 1)$-fcPRS, $(1, S)$-fcPRS, $(1, P)$-fcPRS, $(1, G)$-fcPRS, (S, S)-fcPRS, (P, P)-fcPRS, (S, G)-fcPRS, (P, G)-fcPRS, and (G, G)-fcPRS respectively.

Figure 4.4 shows the hierarchy of fcPRS classes, simply called *fcPRS-hierarchy*. The relations depicted in the hierarchy partly result from the definition of classes. The rest of this section elucidates three equalities in the hierarchy (fcFS = FS, fcPDA = PDA, and fcPN = PN). The principle of the proofs lies in various mechanisms of stowing the content of the store in the process terms.

As the PRS-hierarchy is not strict with respect to the language equivalence, the fcPRS-hierarchy also cannot be strict on the language expressibility level. However, the fcPRS-hierarchy is strict with respect to the bisimulation equivalence (with one exception in the relation between PRS and fcPRS classes, where the situation is not clear). To prove the strictness, we need to show that the new classes differ from each other and also from the classes in PRS-hierarchy. It will be demonstrated in the next chapter which is focused on new classes.

Theorem 4.10. *Let Δ be a $(1, 1)$-fcPRS. There exists $(1, 1)$-fcPRS Δ' with the trivial constraint system $\mathcal{C}_\varepsilon$, isomorphic to Δ.*

Proof. Each state of an arbitrary fcFS consists of exactly one process constant and one constraint. Thus the actual state of the store can be held as a part of such a constant.

Let Δ be of the form $(\mathcal{C}, R, X_0)$, where $\mathcal{C} = (C, \leq, \wedge, tt, ff)$. A new fcFS Δ' is constructed as $(\mathcal{C}_\varepsilon, R', X_0^{(tt)})$, where $\mathcal{C}_\varepsilon$ is the trivial constraint system, $X_0^{(tt)}$ is the initial variable holding the initial state of the store. In R' we

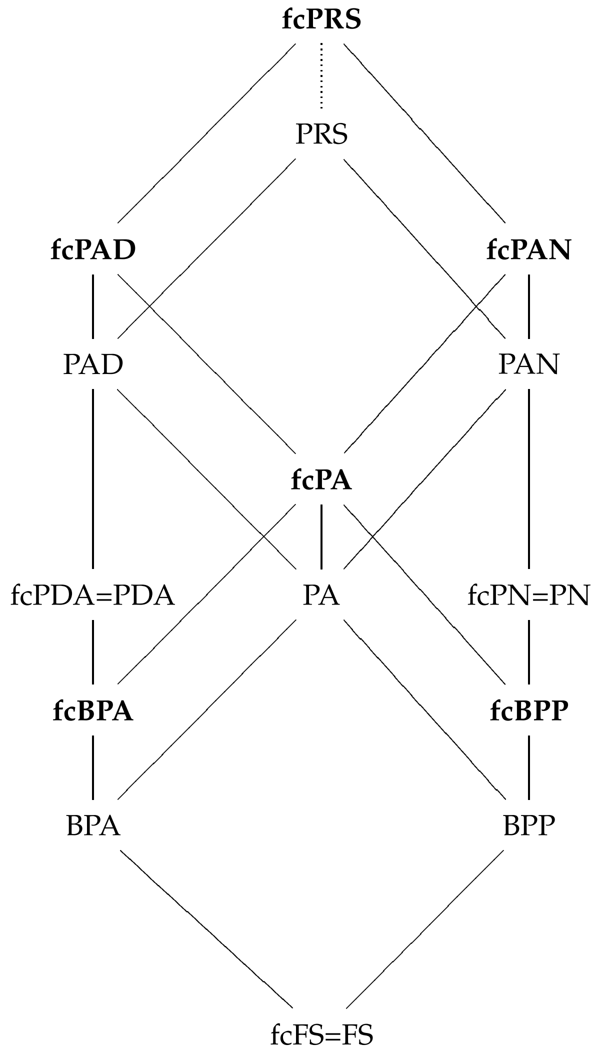


Figure 4.4: The fcPRS-hierarchy

replace every rewrite rule

$$(Y \xrightarrow{a} Z, m, n) \in R$$

by the set of rules

$$(Y^{(o)} \xrightarrow{a} Z^{(o \wedge n)}, tt, tt) \in R'$$

for every $o \in C^{\diamond}$ which satisfies the entailment condition $o \vdash m$ and the consistency condition $o \wedge n \neq ff$. In other words, the entailment and consistency conditions are always satisfied in Δ', but the power of checking for these conditions is not lost, just moved to the new set of rules. The system

Δ' is isomorphic to Δ in the way, that each state $(Y^{(m)}, tt)$ of Δ' corresponds to the state (Y, m) of Δ. $\square$

The equation fcFS = FS is a corollary of above theorem which gives fcFS $\subseteq$ FS and Lemma 4.7.

Theorem 4.11. *Let Δ be a (S, S)-fcPRS. There exists (S, S)-fcPRS Δ' with the trivial constraint system $\mathcal{C}_\varepsilon$, isomorphic to Δ.*

Proof. The idea of the proof is based on the fact that we can add special process constants corresponding to the actual states of the store, one to each state of fcPDA. Then the content of the store will be stored in such special constants, when the store keeps unused (permanently set to *tt*).

Let $\Delta = (\mathcal{C}, R, t_0)$, where the constraint system $\mathcal{C}$ is of the form $(C, \leq, \wedge, tt, ff)$. Let $S = \{S^{(m)} \mid m \in C^\diamond\}$ be the set of special process constants such that $S \cap Const(\Delta) = \emptyset$. A new fcPDA Δ' is constructed as $(\mathcal{C}_\varepsilon, R', S^{(tt)}.t_0)$, where $\mathcal{C}_\varepsilon$ is the trivial constraint system, $S^{(tt)}.t_0$ is the initial term with the special constant holding the initial state of the store. In R' we replace every rewrite rule

$$(t_1 \xrightarrow{a} t_2, m, n) \in R$$

by the set of rules

$$(S^{(o)}.t_1 \xrightarrow{a} S^{(o \wedge n)}.t_2, tt, tt) \in R'$$

for every $o \in C^\diamond$ which satisfies the entailment condition $o \vdash m$ and the consistency condition $o \wedge n \neq ff$. The new rules are constructed to abide by the entailment and consistency conditions connected with original rules. The isomorphism of Δ and Δ' is obvious as every state $(S^{(m)}.t, tt)$ of Δ' corresponds exactly to the state (t, m) of the system Δ. $\square$

Again, the equality fcPDA = PDA is a corollary of the previous theorem and Lemma 4.7. The equality fcPN = PN arises from the same lemma and the following theorem.

Theorem 4.12. *Let Δ be a (P, P)-fcPRS. There exists (P, P)-fcPRS Δ' with the trivial constraint system $\mathcal{C}_\varepsilon$, isomorphic to Δ.*

Proof. The proof is the same as the previous one if we replace every sequential composition "." by the parallel composition "$\|$". $\square$

The intuitive reasons why similar tricks cannot be done for other classes are of two kinds.

In the case of fcBPA, fcBPP, or fcPA the reason is the restriction that only process constants can occur on the left-hand side of the rules. Thus we cannot add any special process constant to every process term in states, as we cannot have two constants on the left-hand side of any rule. The content of the store also cannot be held as a part of some "original" constant Y (like in case of finite-state systems) as such a constant can be lost by a rule of the form $(Y \xrightarrow{a} \varepsilon, tt, tt)$.

In the case of fcPAN, fcPAD, and fcPRS we can add new process constants holding the content of the store. However, there is another problem. The size of a process term is unlimited and there can be more rules which are applicable on different (and far-away from each other) subterms of the process term. In fcPDA case all subterms which can be rewritten immediately are at the beginning of the term, in fcPN case we can assume the same (thanks to the commutativity of parallel composition). Thus the problem is that we do not know where this constant should be placed in a process term as we do not know which subterm will be rewritten by next transition.

Chapter 5

New classes in fcPRS-hierarchy

This chapter describes new classes of transition systems defined by the (α, β)-fcPRS formalism. The proofs that these classes differ from surrounding classes in fcPRS-hierarchy are included. The fcBPA and fcBPP classes were defined (a bit differently) in [Str00a, Str00b].

5.1 fcBPA class

This section is devoted to the class of the transition systems which can be defined by $(1, S)$-fcPRS systems. The abbreviation *fcBPA* corresponds to BPA with finite constraint system.

The fact that BPA is a subclass of fcBPA follows from Lemma 4.7. The witness of the strictness can be found in the example bellow.

Example 5.1. *Let Δ be a fcBPA of the form $(\mathcal{C}_{mn}, R, A)$, where $\mathcal{C}_{mn}$ is the constraint system from Example 4.3 and R contains the following rewrite rules.*

$$\begin{array}{ll}
(A \stackrel{a}{\longrightarrow} A.X, tt, tt) & (X \stackrel{d}{\longrightarrow} Y, m, tt) \\
(A \stackrel{c}{\longrightarrow} \varepsilon, tt, m) & (Y \stackrel{d}{\longrightarrow} \varepsilon, m, tt) \\
(A \stackrel{b}{\longrightarrow} \varepsilon, tt, n) & (X \stackrel{d}{\longrightarrow} \varepsilon, n, tt)
\end{array}$$

Behaviour of this system is represented in Figure 5.1.

The transition graph depicted on Figure 5.1 is not alphabetic (as it is not of finite multiplicity - see [CM90] for the proof and terminology). Thus this transition systems cannot be described by any BPA as the class of BPA corresponds to rooted alphabetic rewrite systems.

Another argumentation can be based on the fact proved in [BCS96], that factorisation of any BPA with respect to the bisimulation equivalence is a regular graph (i.e. there exists some graph grammar generating the graph). Figure 5.2 represents a non-regular factorisation of the transition system depicted on Figure 5.1.

$$
\begin{tikzcd}
& (\varepsilon,m) & (Y,m) \arrow[l, "d"'] & (X,m) \arrow[l, "d"'] & (Y.X,m) \arrow[l, "d"'] & (X.X,m) \arrow[l, "d"'] & \cdots \arrow[l] \\
{} \arrow[r] & (A,tt) \arrow[u, "c"] \arrow[rr, "a"] \arrow[d, "b"] & & (A.X,tt) \arrow[u, "c"] \arrow[rr, "a"] \arrow[d, "b"] & & (A.X.X,tt) \arrow[u, "c"] \arrow[r] \arrow[d, "b"] & \cdots \\
& (\varepsilon,n) & & (X,n) \arrow[ll, "d"'] & & (X.X,n) \arrow[ll, "d"'] & \cdots \arrow[l]
\end{tikzcd}
$$

Figure 5.1: Transition system described in Example 5.1

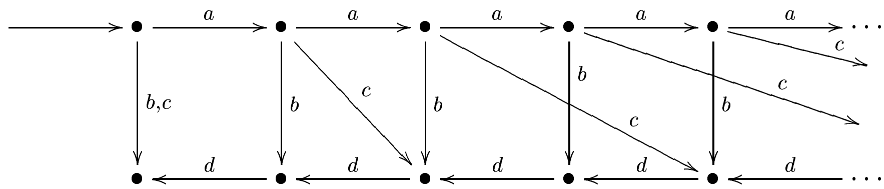


Figure 5.2: Bisimulation collapse of transition graph in Figure 5.1

It can be seen from the definition of (α,β)-fcPRS and Theorem 4.11 that fcBPA is a subclass of PDA. A PDA transition system which cannot be described up to bisimilarity by any fcBPA is presented in Example 5.2.

Example 5.2. *The PDA given by the following rewrite rules and the initial state qX describes the transition system represented in Figure 5.3.*

$$
\begin{array}{lll}
qX \xrightarrow{a} qAX & qX \xrightarrow{c} r & rX \xrightarrow{b} r \\
qA \xrightarrow{a} qAA & qA \xrightarrow{c} r & rA \xrightarrow{b} r \\
qA \xrightarrow{b} q & &
\end{array}
$$

$$
\begin{tikzcd}
{} \arrow[r] & qX \arrow[r, shift left, "a"] \arrow[d, "c"] & qAX \arrow[l, shift left, "b"] \arrow[r, shift left, "a"] \arrow[d, "c"] & qAAX \arrow[l, shift left, "b"] \arrow[r, shift left, "a"] \arrow[d, "c"] & \cdots \arrow[l, shift left, "b"] \\
& r & rX \arrow[l, "b"'] & rAX \arrow[l, "b"'] & \cdots \arrow[l, "b"']
\end{tikzcd}
$$

Figure 5.3: Transition system described in Example 5.2

To see that there is no fcBPA bisimilar to the transition system presented in Figure 5.3, suppose that we have such a fcBPA Δ. Let $M \subseteq$

$Const(\Delta) \times C^{\diamond}$ be the set of states such that $(X, m) \in M$ if and only if there is a reachable state of the form $(X.t, m)$. We can assume that for every process constant $X \in Const(\Delta)$ there is a $m \in C^{\diamond}$ such that $(X, m) \in M$. Let s be defined as

$$s = \max_{(X,m) \in M} s(X, m)$$

where $s(X, m)$ be the length of the longest word in $L((X, m)) \cap \{b, c\}^*$. The number s is finite as every sequence (possible in our system) consisting of $\{b, c\}^*$ is finite. Let (t, m) be the state of Δ bisimilar to $qA^{s'}X$, where $s' = s^{Const(\Delta)} + 1$. Due to bisimilarity with $qA^{s'}X$, the state (t, m) can do an arbitrary sequence of the form $b^i c b^{s'-i}$ $(0 \le i \le s')$. From the definition of s' follows that the term t consists of more than $|Const(\Delta)|$ constants and thus some process constant X occurs twice between the first $|Const(\Delta)|+1$ constants in t, i.e. $t = t_1.X.t_2.X.t_3$ (t_1, t_2, t_3 can be ε). A very important fact is that every sequence of the form $b^i c b^{s'-i}$ will erase at least the first $|Const(\Delta)|$ process constants from the state (t, m). Now, we can make transitions under b^* from $(t_1.X.t_2.X.t_3, m)$ to the state $(X.t_2.X.t_3, n)$ and then we can make other transitions $(X.t_2.X.t_3, n) \stackrel{cb^*}{\longrightarrow} (X.t_3, o)$. The state $(X.t_3, o)$ is bisimilar to the state rA^iX of PDA for appropriate i. The only one possible transition from the state rA^iX is the transition with the label b. But in the set of possible transitions from the state $(X.t_3, o)$ there is also the transition with the label c corresponding to the rule used for the first transition of the previous derivation sequence $(X.t_2.X.t_3, n) \stackrel{cb^*}{\longrightarrow} (X.t_3, o)$ as the entailment and consistency conditions are satisfied forever after the first application of the rule. It is the contradiction with the bisimilarity of $(X.t_3, o)$ and rA^iX.

We have demonstrated that fcBPA class has strictly greater expressive power than classic BPA and strictly lower expressive power than PDA, both with respect to the bisimulation equivalence. On the language expressibility level, all three classes are equal due to the known fact $L(\mathit{BPA}) = L(\mathit{PDA})$ and the demonstrated relation *BPA* $\subsetneq$ *fcBPA* $\subsetneq$ *PDA*.

5.2 fcBPP class

This section presents some basic facts around the class of the transition systems which can be defined by a $(1, P)$-fcPRS. The abbreviation *fcBPP* corresponds to BPP with finite constraint system.

We have already demonstrated that BPP is a subclass of fcBPP, the strictness follows from the example given below which offers a fcBPP transition system which is not in the BPP class.

Example 5.3. *Let Δ be a fcBPP of the form $(\mathcal{C}_{mn}, R, A)$, where $\mathcal{C}_{mn}$ is the constraint system from Example 4.3 and R contains the following rewrite rules.*

$$\begin{array}{ll} (A \xrightarrow{a} A\|X, tt, tt) & (X \xrightarrow{b} \varepsilon, m, tt) \\ (A \xrightarrow{b} \varepsilon, tt, m) & (X \xrightarrow{c} \varepsilon, n, tt) \\ (A \xrightarrow{c} \varepsilon, tt, n) & \end{array}$$

Behaviour of this system is presented in Figure 5.4.

$$\begin{tikzcd}
 & (\varepsilon, m) & (X, m) \arrow[l, "b"'] & (X\|X, m) \arrow[l, "b"'] & \cdots \arrow[l, "b"'] \\
 \arrow[r] & (A, tt) \arrow[u, "b"] \arrow[d, "c"] \arrow[r, "a"] & (A\|X, tt) \arrow[u, "b"] \arrow[d, "c"] \arrow[r, "a"] & (A\|X\|X, tt) \arrow[u, "b"] \arrow[d, "c"] \arrow[r, "a"] & \cdots \\
 & (\varepsilon, n) & (X, n) \arrow[l, "c"'] & (X\|X, n) \arrow[l, "c"'] & \cdots \arrow[l, "c"']
\end{tikzcd}$$

Figure 5.4: Transition system described in Example 5.3

The language generated by this system $L(\Delta) = \{a^{n-1}b^n, a^{n-1}c^n \mid n \geq 1\}$ cannot be generated by any BPP due to the Pumping Lemma presented by Christensen [Chr93] in the following form.

Lemma 5.4 (Pumping Lemma for BPP). *Let L be any language of $L(BPP)$. There exists a constant m such that if u is a word of L and the length of u (written $|u|$) is greater than m, then there exist $x, y, z \in \Sigma^*$ such that*

- $u = xz$,
- $|y| \geq 1$,
- $\forall i \geq 0 : xy^iz \in L$.

The definition of (α, β)-fcPRS classes and proved fact fcPN=PN imply that the fcBPP class is a subclass of PN. These classes differ, even with respect to the language equivalence. The language $L = \{a^nbc^nde^nf \mid n \geq 0\}$ is an instance of a language generated by PN, which cannot be described by any fcBPP. Example 5.5 shows a PN describing the language L, while Pumping Lemma presented in the following subsection gives the argument that there is no fcBPP generating the language L.

Example 5.5. *Let $\Delta = (R, W)$ be a (P, P)-PRS with rewrite rules as below.*

$$\begin{array}{ll} W \xrightarrow{a} W\|A\|B & Y\|B \xrightarrow{e} Y \\ W \xrightarrow{b} X & Y \xrightarrow{f} Z \\ X\|A \xrightarrow{c} X & Z\|A \xrightarrow{z} Z\|A \\ X \xrightarrow{d} Y & Z\|B \xrightarrow{z} Z\|B \end{array}$$

5.2.1 Pumping Lemma for fcBPP

The pumping lemma for fcBPP is formulated and proved in this subsection. The proof is similar to the one presented by Christensen for BPP case [Chr93] thanks to the fact that every possible sequence of actions contains a finite number of transitions which change the state of the store due to finiteness of a constraint system.

Let $\Delta = (\mathcal{C}, R, t_0)$ be a fcBPP. For every process constant $X \in Const(\Delta)$ and every constraint $m \in C^\diamond$, let $S_m(X)$ denote the set

$$S_m(X) = \{Y \in Const(\Delta) \mid \exists t \in P : (X, m) \longrightarrow^+ (Y \| t, m)\}^1,$$

i.e. the set of process constants Y which can be derived from (X, m) without changes on the store. We extend this definition to parallel terms in obvious manner, thus

$$S_m(A_1 \| A_2 \| \ldots \| A_j) = \bigcup_{i \in \{1,2,\ldots,j\}} S_m(A_i).$$

Lemma 5.6. *Let $\Delta = (\mathcal{C}, R, t_0)$ be a fcBPP. If there exists some derivation of a word $u = u_1u_2 \ldots u_k \in L(\Delta)$ of the form*

$$(t_0, tt) = (t_0, m_0) \xrightarrow{u_1} (t_1, m_1) \xrightarrow{u_2} \ldots \xrightarrow{u_k} (t_k, m_k) \not\longrightarrow$$

such that $\forall i \in \{0, 1, 2, \ldots, k\}, \forall X \in t_i$ it holds $X \notin S_{m_i}(X)$, then $|u| \leq h$, where h is a constant depending only on Δ.

Proof. At first we focus on maximum "flat" parts of the above derivation, which are of the form

$$(t_i, m_i) \xrightarrow{u_{i+1}} (t_{i+1}, m_{i+1}) \xrightarrow{u_{i+2}} \ldots \xrightarrow{u_{i+j}} (t_{i+j}, m_{i+j}),$$

where the state of the store (in following marked as m) keeps unchanged ($m = m_i = m_{i+1} = \cdots = m_{i+j}$), $i = 0$ or $m_{i-1} \neq m$, and $i + j = k$ or $m \neq m_{i+j+1}$. We denote $u' = u_{i+1}u_{i+2} \ldots u_{i+j}$. From this flat part we deduce another derivation sequence

$$(r_0 \| s_0, m) \xrightarrow{v_1} (r_1 \| s_1, m) \xrightarrow{v_2} \ldots \xrightarrow{v_p} (r_p \| s_p, m),$$

where $v_1, v_2, \ldots, v_p \in Act^+$, $r_0 \| s_0 = t_i$, in r_0 there are all constants from t_i which are rewritten in the derivation sequence $(t_i, m) \xrightarrow{u'} (t_{i+j}, m)$, and in s_0 there are constants which do not actively participate in this derivation sequence. Now $r_l \| s_l$ $(l = 1, 2, \ldots, p)$ rises from $r_{l-1} \| s_{l-1}$ by one rewriting

[1]The relation $\longrightarrow^+$ (resp. $\longrightarrow^*$) is apprehended as usual, i.e. $(t_1, m) \longrightarrow^+ (t_2, n)$ (resp. $(t_1, m) \longrightarrow^* (t_2, n)$) iff there exists $w \in Act^+$ (resp. $w \in Act^*$) such that $(t_1, m) \xrightarrow{w} (t_2, n)$.

of each constant from r_{l-1} in the same way as a constant has been rewritten in the original flat derivation sequence (thus $|v_l| = |r_{l-1}|$) and still it holds that r_l contains constants, which are rewritten in the original flat derivation sequence, while s_l contains the other constants (thus $s_{l-1} \subseteq s_l$). We finish rewriting when r_l is empty (thus $r_p = \varepsilon$ and $s_p = t_{i+j}$). It is clear that $v = v_1 v_2 \dots v_p$ is a permutation of u', especially $|v| = |u'|$. By replacing $(t_i, m) \xrightarrow{u'} (t_{i+j}, m)$ with $(r_0 \| s_0, m) \xrightarrow{v} (r_p \| s_p, m)$ in the original derivation we get a correct derivation of the word $u_1 \dots u_i v u_{i+j+1} \dots u_n$ of the length k. Further, for each X in r_l ($l = 0, 1, 2, \dots, p$) there exists t_z ($i \leq z \leq i + j$) such that $X \in t_z$.

Now we show that $S_m(r_{l-1}) \supsetneq S_m(r_l)$ for each $1 \leq l < p$.

"$\supseteq$" It comes directly from the fact that each constant from r_l has an ancestor in r_{l-1}.

"$\neq$" Let us assume that for some $1 \leq l < p$ we have $S_m(r_{l-1}) = S_m(r_l)$. For each $X \in r_l$ ($r_l \neq \varepsilon$) it holds that $X \in S_m(r_{l-1})$ and thus $X \in S_m(r_l)$. From the premise $X \notin S_m(X)$ follows that there exists some $Y \in r_l$, $Y \neq X$ such that $X \in S_m(Y)$. Analogous reasoning as for X can be done for Y, i.e. from $Y \in r_l$ it follows that $Y \in S_m(r_{l-1}) = S_m(r_l)$ and $Y \notin S_m(Y)$, $Y \notin S_m(X)$. In conclusion we get $Y \in S_m(r_l)$ and $Y \notin S_m(X \| Y)$. Again, there exists $Z \in r_l$, $Z \notin \{X, Y\}$ such that $Y \in S_m(Z)$ and thus also $\{X, Y\} \subseteq S_m(Z)$. We know $Z \in r_l$ and $Z \notin S_m(Z)$, hence we get $Z \in S_m(r_l)$ and $Z \notin S_m(X \| Y \| Z)$. We can continue in this fashion to the point where we have the contradiction $W \in S_m(r_l)$ and $W \notin S_m(r_l)$.

Hence we have

$$|Const(\Delta)| \geq |S_m(r_0)| > |S_m(r_1)| > \dots > |S_m(r_{p-1})| \geq 0.$$

This implies $|Const(\Delta)| \geq p - 1$. Further, for each $1 \leq l \leq p$ it holds that

$$|v_l| = |r_{l-1}| \leq |r_0| a^{l-1} \leq |r_0| a^{p-1} \leq |r_0| a^{|Const(\Delta)|},$$

where a is a maximum number of constants in right sides of rewrite rules in Δ. Now we restrict the length of u'

$$\begin{aligned} |u'| &= |v| = \sum_{l=1}^{p} |v_l| \leq \sum_{l=1}^{p} |r_0| a^{|Const(\Delta)|} = p |r_0| a^{|Const(\Delta)|}, \\ |u'| &\leq p |r_0| a^{|Const(\Delta)|} \leq (|Const(\Delta)| + 1) |t_i| a^{|Const(\Delta)|}. \end{aligned}$$

In conclusion we get the restriction on the length of flat parts of the original derivation

$$|u'| \leq |t_i| b,$$

where $b = (|Const(\Delta)| + 1)a^{|Const(\Delta)|}$.

In general it holds that each sequence of derivation steps consists of non-flat steps and flat derivation sequences. The number of "unflat" steps $(t_i, m_i) \xrightarrow{u_{i+1}} (t_{i+1}, m_{i+1})$, where $m_i \neq m_{i+1}$, is limited by $|C^\diamond| - 1$. The cardinality of the set C also constrains the number of flat parts to $|C^\diamond|$. Therefore

$$|u| \leq |C^\diamond| - 1 + \sum_{j=1}^{|C^\diamond|} |t'_j| b,$$

where (t'_j, m'_j) is the first state of the j-th flat derivation sequence, i.e. m'_j is the j-th different state of the store used in the original derivation and (t'_j, m'_j) is the first state in this derivation with the constraint m'_j in the store. Hence $(t'_1, m'_1) = (t_0, tt)$.

The last step is to restrict the length of t'_j for $j > 1$. We can deduce a restriction

$$|t'_j| \leq |t'_{j-1}| + (a-1)(|t'_{j-1}|b + 1)$$

thanks to the facts that each application of a rewrite rule cannot add more than $a - 1$ constants to the string of constants in the actual state and that the number of these applications is limited by the length of the previous flat string plus one (the unflat derivation step). The previous inequality can be modified in the following way.

$$\begin{array}{rcl}
|t'_j| & \leq & |t'_{j-1}| + a(|t'_{j-1}|b + 1) \\
|t'_j| & \leq & |t'_{j-1}|(1 + ab + a) \\
|t'_j| & \leq & |t'_1|(1 + ab + a)^{j-1} \\
|t'_j| & \leq & |t_0|(1 + ab + a)^{j-1}
\end{array}$$

By summarisation we get

$$|u| \leq |C^\diamond| - 1 + b|t_0| \sum_{j=1}^{|C^\diamond|} (1 + ab + a)^{j-1},$$

where $b = (|Const(\Delta)| + 1)a^{|Const(\Delta)|}$. The sum on the right side of the previous inequality can be modified as it is an geometric progression. The final form of desired h is then

$$h = |C^\diamond| - 1 + b|t_0| \frac{(1 + ab + a)^{|C^\diamond|} - 1}{ab + a},$$

where a is the maximum number of constants in right sides of rewrite rules in Δ and $b = (|Const(\Delta)| + 1)a^{|Const(\Delta)|}$. □

The pumping lemma formulated below is a simple consequence of the previous lemma.

Lemma 5.7 (Pumping Lemma for fcBPP). *Let L be a language of L(fcBPP). There exists a constant h such that if u is a word of L and $|u| > h$ then there exist $x, y, z, w \in Act^*$ such that*

- $u = xz$,
- $|y| > 1$,
- $\forall i \geq 0 : xy^i zw^i \in L$.

Proof. Given L we have a fcBPP Δ such that $L = L(\Delta)$. It follows from Lemma 5.6 that each derivation

$$(t_0, tt) = (t_0, m_0) \xrightarrow{u_1} (t_1, m_1) \xrightarrow{u_2} \dots \xrightarrow{u_k} (t_k, m_k) \not\longrightarrow$$

of the word $u = u_1 u_2 \dots u_k \in L(\Delta)$, $|u| > h$ contains some state $(t_j, m_j) = (X \| t'_j, m_j)$, where $X \in S_{m_j}(X)$. The definition of $S_{m_j}(X)$ says that there exist $t \in P$ and $y \in Act^+$ such that $(X, m_j) \xrightarrow{y} (X \| t, m_j)$. Further, let $w \in Act^*$ be a word in $L((t, m_k))$, i.e. there exists a terminal state (t', n) such that $(t, m_k) \xrightarrow{w} (t', n)$. Now the derivation

$$(t_0, tt) \xrightarrow{u_1 \dots u_j} (t_j, m_j) \xrightarrow{y^i} (t_j t^i, m_j) \xrightarrow{u_{j+1} \dots u_k} (t^i, m_k) \xrightarrow{w^i} (t'^i, n) \not\longrightarrow$$

is the correct one for all $i \geq 0$. To make the proof complete we should add that $x = u_1 \dots u_j$ and $z = u_{j+1} \dots u_k$. $\square$

5.3 fcPA, fcPAD, fcPAN classes

From the Lemma 4.7 follows that PA is a subclass of the fcPA class, PAD is a subclass of fcPAD, and PAN is a subclass of fcPAN. To prove that mentioned PRS classes are strict subclasses of corresponding fcPRS classes, we present two fcPRS systems. The first is a fcBPA system which is not bisimilar to any PAN system. The second will be fcBPP system which is not bisimilar to any PAD system.

Example 5.8. *Let us consider a fcBPA system given as an $(1, S)$-fcPRS with the constraint system $\mathcal{C}_{m,n}$ introduced in Example 4.3 and the initial process term $U.X$.*

$$\begin{array}{ll}
(U \xrightarrow{a} U.A, tt, tt) & (A \xrightarrow{a} \varepsilon, tt, tt) \\
(U \xrightarrow{b} U.B, tt, tt) & (B \xrightarrow{b} \varepsilon, tt, tt) \\
(U \xrightarrow{c} \varepsilon, tt, m) & (X \xrightarrow{e} \varepsilon, m, tt) \\
(U \xrightarrow{d} \varepsilon, tt, n) & (X \xrightarrow{f} \varepsilon, n, tt)
\end{array}$$

The fcBPA system given above is bisimilar to the pushdown system defined in Example 3.8, which is not bisimilar to any PAN system. Hence this fcBPA system is not bisimilar to any PAN, and as corollary we get PA $\subsetneq$ fcPA and PAN $\subsetneq$ fcPAN.

We will prove that there is no PAD system bisimilar to fcBPP system given by the example below.

Example 5.9. *Let us consider a fcBPP system given as an* $(1, P)$*-fcPRS with the constraint system depicted below and the initial state* (X, tt)*.*

$$
\begin{array}{c}
\mathit{ff} \\
| \\
o \\
| \\
tt
\end{array}
\qquad\qquad
\begin{array}{l}
(X \xrightarrow{a} X\|A, tt, tt) \\
(X \xrightarrow{b} X\|B, tt, tt) \\
(X \xrightarrow{e} \varepsilon, tt, o) \\
(A \xrightarrow{c} \varepsilon, o, tt) \\
(B \xrightarrow{d} \varepsilon, o, tt)
\end{array}
$$

Lemma 5.10. *If there is a PAD system bisimilar to the fcBPP system from Example 5.9, then there is also a PDA system bisimilar to this fcBPP.*

Proof. Let Δ be a PAD with the initial state Q (we can assume that the initial state is a single constant) such that $Q \sim (X, tt)$. As on the left-hand side of rewrite rules Δ only sequential composition can occur, some part of parallel composition $t_1\|t_2$ can influence the behaviour of such system only if there is a reachable state of the form $(t_1\|t_2).t_3$ where t_3 can be ε. If there is no such a state, we can remove all parallel compositions from the rules and we get a PDA system bisimilar to Δ and thus also bisimilar to the considered fcBPP process.

Another situation arises if there is a reachable state of Δ of the form $(t_1\|t_2).t_3$, where t_3 can be ε. Let us assume that during the derivation of the state $(t_1\|t_2).t_3$ from Q there is no other state of the form $(t_1'\|t_2').t_3'$ (t_3 can be ε). As Q is a single process constant and any parallel composition $s_1\|s_2$ in a term $p.(s_1\|s_2).p'$ cannot be changed by any rewriting until p is ε, there must be some rewrite rule $(t \xrightarrow{x} l.(t_1\|t_2).r) \in \Delta$ (l, r can be ε, $x \in \{a, b, c, d, e\}$) such that $t_1\|t_2$ is the mentioned parallel composition. There are two cases.

1. The state $(t_1\|t_2).t_3$ was derived from Q under a word $w \in \{a, b\}^*$. We show that t_1 or t_2 is then deadlocked. With respect to the definition of PAD, which does not provide any form of communication or synchronisation between processes in a parallel composition, just one component of $t_1\|t_2$ can enable the action e, let us assume that it is t_2. Then t_1 is deadlocked – it cannot do neither the actions a or b (as these actions are disabled after the action e) nor the actions c or d (as these actions are disabled before e). Nevertheless, the term $t_1.t'$ is not necessarily deadlocked for some term t'. Hence, the parallel composition $t_1\|t_2$ in the rule $(t \xrightarrow{x} l.(t_1\|t_2).r) \in \Delta$ can be changed to the sequential composition $t_2.t_1$. We should insert some separator

between t_2 and t_1 (resp. l and t_2) to keep the impossibility of communication between parts of parallel composition (resp. between l and part of the following parallel composition). Thus we replace the rule $(t \xrightarrow{x} l.(t_1\|t_2).r) \in \Delta$ by the rule $t \xrightarrow{x} l.X.t_2.X.t_1.r$ (resp. $t \xrightarrow{x} t_2.X.t_1.r$ if $l = \varepsilon$), where $X \notin Const(\Delta)$ is a new constant, and we add new rewrite rule $X.s \xrightarrow{x} s'$ to Δ for every rewrite rule $s \xrightarrow{x} s' \in \Delta$ (if we already have the rules of the form $X.s \xrightarrow{x} s'$ in modified Δ, we do not need to add them again in the future). These changes do not affect the behaviour of Δ.

2. The action e occurs during the derivation of the state $(t_1\|t_2).t_3$ from Q. The state $(t_1\|t_2).t_3$ is then bisimilar to a state $(A^n\|B^m, o)$[2] of considered fcBPP and thus every possible sequence of actions performed by the process $(t_1\|t_2).t_3$ is finite, as well as every possible sequence performed by the term $t_1\|t_2$. We construct a finite labelled (acyclic) transition graph where the vertices are processes reachable from the parallel composition $t_1\|t_2$ (which is the root of the graph) and edges naturally correspond to actions (resp. applications of rewrite rules). Now we assign a fresh process constant to each vertex of the graph which has some parallel composition inside (the vertices without any parallel composition keep unchanged). We replace the rule $(t \xrightarrow{x} l.(t_1\|t_2).r) \in \Delta$ by the rule $t \xrightarrow{x} l.Z.r$, where $Z \notin Const(\Delta)$ is a process constant assigned to $t_1\|t_2$. For every edge of the graph from the vertex A (where A is a fresh constant) to the vertex v we add a rule $A \xrightarrow{x} v$ (where x is the label of the edge) to Δ. The behaviour of Δ is still unchanged thanks to the fact that if $(t_1\|t_2).t_3 \xrightarrow{*} t'.t_3$ then the term t_3 can be changed by the following transition only if there is no parallel composition in t', and the fact that the vertices without any parallel composition are unchanged.

In both cases, the number of parallel compositions in rewrite rules has decreased (with one exception – when we add rules of the form $X.s \xrightarrow{x} s'$, then the number of parallel compositions can be doubled, but it does not matter as we make it only once). If there is still a reachable state of the form $(t_1\|t_2).t_3$ in modified Δ, we can use the same method again. As the number of parallel compositions in rewrite rules is finite, after finite number of steps we get a PAD system without any reachable state of the form $(t_1\|t_2).t_3$, which is the situation discussed at the beginning of this proof. $\square$

The class of context-free languages (i.e. the class of languages generated by PDA processes) is closed under intersection with regular languages. The

[2]The expression A^n is an abbreviation for n copies of process constant A in parallel composition. The abbreviation B^m has an analogous meaning.

language L generated by the fcBPP system from Example 5.9 is not context-free, as its intersection with the regular language $a^*b^*ec^*d^*$ is the language $L \cap a^*b^*ec^*d^* = \{a^n b^m ec^n d^m \mid m, n \geq 0\}$ which is not context-free. Thus there is no PDA process bisimilar to fcBPP from Example 5.9 and from the Lemma 5.10 follows that there is no PAD process bisimilar to considered fcBPP. The direct corollary is the inequality PAD $\subsetneq$ fcPAD.

It has been proven that PA, PAD and PAN classes are strict subclasses of corresponding (α, β)-fcPRS classes. It is obvious from the definition that fcPA is a subclass of fcPAD and fcPAN. It remains to show that fcPA is a strict subclass of fcPAN and fcPAD and that fcPAN differs from fcPAD. To prove it, we introduce a PDA process which is not bisimilar to any fcPAN process, and a PAN process which is not bisimilar to any fcPAD process.

Example 5.11. *Let us consider the pushdown process described in Example 3.8 with added rewrite rules below.*

$$\begin{array}{ll} V \xrightarrow{x} U.X & \qquad W \xrightarrow{x} U.X \\ V \xrightarrow{z} Z & \qquad W \xrightarrow{z} Z \end{array}$$

This system behaves like the one defined in Example 3.8, but when the original system terminates, the enhanced system can choose between termination under the action z and restart under the action x.

Lemma 5.12. *If there is a fcPAN system bisimilar to the PDA process from Example 5.11, then there is a PAN process bisimilar to the PDA system from Example 3.8.*

Proof. Let Δ be a fcPAN system bisimilar to the PDA process defined in Example 5.11. From the finiteness of the constraint system used in Δ follows that there exists a non-terminal reachable state (t, o) of Δ such that every non-terminal state reachable from (t, o) has also o on the store (the contrary implies the infiniteness of the constraint system). As (t, o) is non-terminal, there exist a word $w \in \{a, b, c, d, e, f\}^*$ such that $(t, o) \xrightarrow{w.x} (s, o)$, where (s, o) is bisimilar to the state $U.X$ of the PDA process from Example 5.11. To summarise, (s, o) is a fcPAN process bisimilar to $U.X$ and every non-terminal state reachable from (s, o) has o on the store (terminal states are reachable only under the action z).

If we remove from Δ the rules labelled by actions x, z and consider the state (s, o) to be the initial, we obtain the system with reachable states with o on the store, bisimilar to the pushdown process from Example 3.8.

Now, let Δ' be a PAN system with the initial state s and with the set of rewrite rules consisting of rules $l \xrightarrow{v} r$, where $(l \xrightarrow{v} r, m, n) \in \Delta$, $o \vdash m$, $o \wedge n = o$ and $v \in \{a, b, c, d, e, f\}$. It is clear from above arguments that this PAN system Δ' is bisimilar to the PDA system defined in Example 3.8. $\square$

Mayr in [May97b] has proved that there is no PAN process bisimilar to the PDA process from Example 3.8, thus there is no fcPAN process bisimilar to the pushdown process described by Example 5.11. Hence, fcPA is a strict subclass of fcPAD and the classes fcPAN, fcPAD are different. To check that fcPA is also a strict subclass of fcPAN, we show that there is a PAN process which cannot be described by any fcPAD process with respect to bisimulation equivalence.

Example 5.13. *Let Δ be a PAN process with the initial state $(X\|A\|B).W$ and the following rewrite rules.*

$$\begin{array}{lll}
X \xrightarrow{g} X\|A\|B & Y\|A \xrightarrow{a} Y & X \xrightarrow{y} \varepsilon \\
X \xrightarrow{c} Y & Y\|B \xrightarrow{b} Y & Y \xrightarrow{y} \varepsilon \\
X\|A \xrightarrow{d} Z & Y\|A \xrightarrow{d} Z & Z \xrightarrow{y} \varepsilon \\
X\|B \xrightarrow{d} Z & Y\|B \xrightarrow{d} Z & A \xrightarrow{y} \varepsilon \\
 & & B \xrightarrow{y} \varepsilon \\
 & & W \xrightarrow{x} (X\|A\|B).W \\
 & & W \xrightarrow{z} D
\end{array}$$

The first two columns of rewrite rules include the same rules as Petri net given by Example 3.9. Also the initial state of that PN is very similar to the one of PAN system above. This PAN system can behave as mentioned Petri net (it can deviate from the behaviour of PN only under action y) and states corresponding to terminal states of considered PN can perform a sequence of actions y^* to reach the state W. The state W can perform the action z leading to deadlock, or the action x restarting the PAN system.

Lemma 5.14. *If there is a fcPAD system bisimilar to the PAN process from Example 5.13, then there is a PAD process bisimilar to the PN system from Example 3.9.*

Proof. The proof is made in the same fashion as the previous one. Let Δ be a fcPAD system bisimilar to the PAN process defined in Example 5.13. From the finiteness of constraint system used in Δ follows that there exists a non-terminal reachable state (t, o) of Δ such that every non-terminal state reachable from (t, o) has also o on the store (the contrary implies the infiniteness of the constraint system). As (t, o) is non-terminal, there exist a word $w \in y^*$ such that $(t, o) \xrightarrow{w.x} (s, o)$, where (s, o) is bisimilar to the state $(X\|A\|B).W$ of the considered PAN process. To summarise, (s, o) is a fcPAD process bisimilar to $(X\|A\|B).W$ and every non-terminal state reachable from (s, o) has o on the store (terminal states are reachable only under action the z).

If we remove from Δ the rules labelled by actions y, x, z and consider the state (s, o) as the initial, we obtain the system with reachable states with o on the store, bisimilar to the Petri net from Example 3.9.

Now, let Δ' be a PAD system with the initial state s and with the set of rewrite rules consisting of rules $l \xrightarrow{v} r$, where $(l \xrightarrow{v} r, m, n) \in \Delta$, $o \vdash m$, $o \wedge n = o$ and $v \in \{g, a, b, c, d\}$. It is clear from above arguments that this PAD system Δ' is bisimilar to the Petri net system defined in Example 3.9.
□

Again, Mayr in [May97b] has showed that the Petri net described by Example 3.9 is not bisimilar to any PAD system. Hence, the PAN system from Example 5.13 is not bisimilar to any fcPAD system and we have proved that fcPA is a strict subclass of the class of fcPAN processes. We also have demonstrated that the difference between classes fcPAD and fcPAN is "symmetric".

5.4 fcPRS class

At the beginning of this section, we should explain why the edge between PRS and fcPRS classes in the fcPRS-hierarchy (depicted on Figure 4.4) is dotted while other edges are not. The reason is that we have no proper proof (yet – as we hope) that the fcPRS class has strictly bigger expressibility than the PRS class. It is obvious from the definitions that PRS $\subseteq$ fcPRS, but we can provide only intuition for PRS $\subsetneq$ fcPRS. The assumed witness of the inequality can be found in the fcPA below.

Example 5.15. *Let Δ be a fcPA system with the initial process term $X \| Y$ and the following constraint system and rewrite rules.*

$$\begin{array}{cc}
\begin{array}{c} \mathit{ff} \\ | \\ o \\ | \\ p \\ | \\ \mathit{tt} \end{array}
&
\begin{array}{l}
(X \xrightarrow{a} X.A, tt, tt) \\
(X \xrightarrow{b} X.B, tt, tt) \\
(Y \xrightarrow{c} Y.C, tt, tt) \\
(X \xrightarrow{x} \varepsilon, tt, p) \\
(Y \xrightarrow{y} \varepsilon, p, o) \\
(A \xrightarrow{a'} \varepsilon, o, tt) \\
(B \xrightarrow{b'} \varepsilon, o, tt) \\
(C \xrightarrow{c'} \varepsilon, o, tt)
\end{array}
\end{array}$$

The behaviour of Δ defined in the example above is as follows. At the beginning, the process X can perform some actions a, b and remember the order of the actions, while the process Y can perform just the action c and count the number of performed actions c. The process X can also perform the action x, make a remark p on the store about this action and terminate. Thereafter, the process Y can perform the action y, make a remark o on the store and terminate. When both processes X and Y are terminated (i.e. there is $p \wedge o = o$ on the store), actions a', b', c' can be performed. The order (and the count) of actions a', b' corresponds in reversed order to actions a, b produced before termination of the process X. The count of actions c' is the same as the count of actions c performed before termination of the process Y.

We can approve that this fcPA system is not bisimilar to any PAD process. For the proof we consider the fcPA process without rules labelled by the action b (if we assume that there is a PAD process bisimilar to the original fcPA, then there is also a PAD system without b action bisimilar to the fcPA without b action). Then the behaviour of our system is very similar to the behaviour of fcBPP from Example 5.9, which is not bisimilar to any PAD process. The proof is very similar too.

We can also approve that the considered fcPA process is not bisimilar to any Petri net. The argumentation is based on the fact, that if we remove the rules labelled by c from the fcPA system, then we get a system describing the language $L = \{w.x.y.w^R \mid w \in \{a,b\}^*\}$. The proof that there is no Petri net generating the language L, can be found in [Pet81].

Now we try to explain (on very intuitive level) why we think that there is no PRS process bisimilar to the considered fcPA. Let us assume that Δ is such a bisimilar PRS system. We know this PRS cannot be described by any PAD process. Thus, there must be reachable state with some parallel composition. As the use of the parallel composition must be "non-removable", the information about performed actions a, b, c should be stored in some components of this parallel composition. There should be one parallel component (let us call it p) which saves the information about the order of actions a, b (and thus p is a sequential composition, at least at the top-level), and another parallel component (let us call it q) which remembers the number of performed actions c (the information about the count of actions c cannot be mixed with the information about the order of actions a, b, because after the action y we need a "random access" to the count of actions c). As the sequence of actions a, b can be arbitrary long, the size of corresponding parallel component p is "unbounded" (i.e. for every $n \geq 0$, there is a reachable state where $size(p) > n$). Let m be the maximum size of left-hand sides of rewrite rules in Δ. Further, consider the state of the form $(p \| q \| s).r$, where $size(p) > m$ and process terms s, r can be ε. Then there is no rule, which can change p together with some other part of the term. In other word, there is no way how can q or s provide an information to p. We need such kind of communication for the transition labelled by y, which allows to perform actions a', b', c'. One possible way how to enable these actions at the same time, is to add some term l in front of the parallel composition and enable the action by removing l. But any application of a rewrite rule on the process term of the form $l.(p \| q \| s).r$ cannot modify the process term p if p is large enough. Thus we cannot add information about next possibly performed actions a, b to p (as well as l cannot be generated by p after the action x if p is large enough). In other words, the problem is that a very large parallel component (which is an sequential composition at the top-level) cannot get any information from other parallel components.

We should note that we already know that both, fcPAD and fcPAN classes, are strict subclasses of fcPRS with respect to bisimilarity. This follows directly from the fact, that fcPAD and fcPAN are incomparable subclasses of the fcPRS class.

Chapter 6

Conclusion

We have enriched process rewrite systems with the mechanism related to computing with partial information in the form used in widely studied concurrent constraint programming. In the case of process rewrite systems, this mechanism can be effectively used to provide some information to every part of the process term.

It has been proven that the enriching the classes of finite systems, pushdown processes, and Petri nets with the finite constraint system does not change their expressibility with respect to the bisimulation equivalence and even with respect to isomorphism of generated labelled transition systems. On the contrary, the process rewrite systems of classes BPA, BPP, PA, PAD, and PAN extended with finite constraint system establish corresponding new classes fcBPA, fcBPP, fcPA, fcPAD, and fcPAN as the expressive power of such systems increases. Regrettably, we cannot state that PRS is a strict subclass of the fcPRS class. Although, a commentary in this sense to the relation between fcPRS and PRS classes was given.

The hierarchy of fcPRS classes has been introduced and the strictness with respect to the bisimulation equivalence of such a hierarchy (with the exception in the relation between PRS and fcPRS classes) has been proven, mainly with use of examples. Despite of the fact that the hierarchy is not strict on the language equivalence level, we demonstrated that BPP is a strict subclass of fcBPP and fcBPP is a strict subclass of Petri nets even with respect to the language equivalence. We have also presented the Pumping Lemma for fcBPP.

6.1 Future research

The area of process rewrite systems with finite constraint systems still offers many interesting topics. The topic number one is obviously the relation between classes of fcPRS and PRS processes. Other two topics for our future work are provided by the fcBPP class. The first one is an open question of decidability of the bisimulation equivalence for fcBPP since the decidability of the bisimulation equivalence for BPP has been already proven by Christensen, Hirshfeld and Moller [CHM93] and Moller [Mol96] has shown that the bisimulation equivalence is undecidable for multiset automata[1]. The second interesting challenge around the fcBPP class would be to specify the boundary of decidability of the weak bisimulation equivalence with finite-state processes. Mayr [May96] has proved that the weak bisimulation equivalence with FS processes is decidable for BPP and Jančar, Kučera and Mayr [JKM98] have demonstrated undecidability of this problem for MSA.

Next possible subject is to observe the dependency between the shape of constraint system (number of constraints, branching limitations, etc.) and expressive power of process rewrite systems using such constraint systems. We take into account just two classes of process rewrite systems – with the trivial constraint systems $\mathcal{C}_\varepsilon$ (defined in Example 4.2) and with arbitrary finite constraint systems. Probably, there can be found a finer hierarchy of process rewrite systems with finite constraint systems.

Totally different mission is to employ an infinite constraint system.

[1]MSA are in [Mol96] called as PPDA. In [Mol98] there was also demonstrated that the class of MSA is a strict subclass of Petri nets. It was proven in [Str00b] that fcBPP is a strict subclass of MSA and that the expressibility of MSA systems is not changed by enriching with finite constraint systems.

Bibliography

[BCS96] O. Burkart, D. Caucal, and B. Steffen. Bisimulation collapse and the process taxonomy. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: 7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 247–262. Springer-Verlag, 1996.

[BE97] O. Burkart and J. Esparza. More infinite results. *Bulletin of the European Association for Theoretical Computer Science*, 62:138–159, 1997.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[Cau92] D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106:61–86, 1992.

[CHM93] S. Christensen, Y. Hirshfeld, and F. Moller. Bisimulation equivalence is decidable for basic parallel processes. In Eike Best, editor, *CONCUR '93: 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, 1993.

[Chr93] S. Christensen. *Decidability and Decomposition in Process Algebras*. PhD thesis, Department of Computer Science, University of Edinburgh, 1993.

[CM90] D. Caucal and R. Monfort. On the transition graphs of automata and grammars. In *WG '90: Graph-Theoretic Concepts in Computer Science*, volume 484 of *Lecture Notes in Computer Science*, pages 311–337. Springer-Verlag, 1990.

[dBP92] F. S. de Boer and C. Palamidessi. A process algebra of concurrent constraint programming. In Krzysztof Apt, editor, *JICSLP '92: Joint International Conference and Symposium on Logic Programming*, pages 463–477. MIT Press, 1992.

[Esp97] J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Jan95] P. Jančar. Undecidability of bisimilarity for Petri nets and some related problems. *Theoretical Computer Science*, 148(2):281–301, 1995.

[JKM98] P. Jančar, A. Kučera, and R. Mayr. Deciding bisimulation-like equivalences with finite-state processes. *Lecture Notes in Computer Science*, 1443:200–211, 1998.

[May96] R. Mayr. Weak bisimulation and model checking for basic parallel processes. *Lecture Notes in Computer Science*, 1180:88–99, 1996.

[May97a] R. Mayr. Combining Petri nets and PA-processes. In M. Abadi and T. Ito, editors, *TACS '97: International Symposium on Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 547–561. Springer-Verlag, 1997.

[May97b] R. Mayr. Process rewrite systems. *Electronic Notes in Theoretical Computer Science*, 7, 1997.

[May98] R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems*. PhD thesis, TU München, 1998.

[Mil80] R. Milner. A calculus on communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mol96] F. Moller. Infinite results. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: 7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 195–216. Springer-Verlag, 1996.

[Mol98] F. Moller. A taxonomy of infinite state processes. *Electronic Notes in Theoretical Computer Science*, 18, 1998.

[Par81] D. M. R. Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science: 5th GI-Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[Sar89] V. A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Computer Science Department, Carnegie Mellon University, 1989.

[SR90] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM Press, 1990.

[SRP91] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, 1991.

[Sti95] C. Stirling. Local model checking games. In Insup Lee and Scott A. Smolka, editors, *CONCUR '95: 6th International Conference on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, 1995.

[Str00a] J. Strejček. Constrained rewrite transition systems. Master's thesis, Faculty of Science, Masaryk University, 2000.

[Str00b] J. Strejček. Constrained rewrite transition systems. Technical Report FIMU-RS-2000-12, Faculty of Informatics, Masaryk University, 2000.

[vG90] R. J. van Glabbeek. The linear time-branching time spectrum. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90: 1st International Conference on Concurrency Theory*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 1990.