

Parallel Pushdown Automata and Commutative Context-Free Grammars in Bisimulation Semantics (Extended Abstract)

Jos C. M. Baeten

CWI
Amsterdam, The Netherlands
Jos.Baeten@cwi.nl

Bas Luttik

Eindhoven University of Technology
Eindhoven, The Netherlands
s.p.luttik@tue.nl

A classical theorem states that the set of languages given by a pushdown automaton coincides with the set of languages given by a context-free grammar. In previous work, we proved the pendant of this theorem in a setting with interaction: the set of processes given by a pushdown automaton coincides with the set of processes given by a finite guarded recursive specification over a process algebra with actions, choice, sequencing and guarded recursion, if and only if we add sequential value passing. In this paper, we look what happens if we consider parallel pushdown automata instead of pushdown automata, and a process algebra with parallelism instead of sequencing.

1 Introduction

This paper contributes to our ongoing project to integrate the theory of automata and formal languages on the one hand and concurrency theory on the other hand. The integration requires a more refined view on the semantics of automata, grammars and expressions. Instead of treating automata as language acceptors, and grammars and expressions as syntactic means to specify languages, we propose to view them both as defining process graphs. The great benefit of this approach is that process graphs can be considered modulo a plethora of behavioural equivalences [18]. One can still consider language equivalence and recover the classical theory of automata and formal languages. But one can also consider finer notions such as bisimilarity, which is better suited for interacting processes.

The project started with a structural characterisation of the class of finite automata of which the processes are denoted by regular expressions up to bisimilarity [5]. The investigation of the expressiveness of regular expressions in bisimulation semantics was continued in [9]. In [10], we replaced the Turing machine as an abstract model of a computer by the Reactive Turing Machine, which has interaction as an essential ingredient. Transitions have labels to give a notion of interactivity, and we consider the resulting process graphs modulo bisimilarity rather than language equivalence. Thus a Reactive Turing Machine defines an *executable* interactive process, refining the notion of computable function.

In the same way as classical automata theory defines a hierarchy of formal languages, we obtain a hierarchy of processes. In [4], we proved that the set of processes given by a pushdown automaton coincides with the set of processes given by a finite guarded recursive specification over a process algebra with actions, choice, sequencing and guarded recursion, if and only if we add sequential value passing. Pushdown automata provide an abstract model of a computer with a memory in the form of a stack. In this paper, we consider the abstract model of a computer with a memory in the form of a bag. We consider the correspondence between parallel pushdown automata and commutative context-free grammars. In the process setting, a commutative context-free grammar is a process algebra comprising actions, choice, parallelism and recursion. We start out from the process algebra BPP, extended with constants for acceptance and non-acceptance (deadlock).

Then we find that in one direction, every process of a finite guarded recursive specification over this process algebra is the process of a parallel pushdown automaton, but not the other way around: there are parallel pushdown automata with a process that is not the process of any finite guarded recursive specification. This is even the case for the one-state parallel pushdown automaton of the bag itself, there is no finite guarded BPP-specification for it. If we do want to get a recursive specification for the bag, we need to give some actions priority over others, and can find a satisfactory specification over BPP_θ , BPP extended with the priority operator. Indeed, we can obtain a finite guarded specification over BPP_θ for every one-state parallel pushdown automaton. On the other hand, there is a parallel pushdown automaton with two states that does not have a finite guarded specification over BPP_θ .

If we add communication with value passing to this algebra, resulting in BCP_θ , we do get a complete correspondence: a process is the process of a parallel pushdown automaton if and only if it is the process of a finite guarded recursive specification. We can also get this result in a setting without the priority operator, so over BCP , but then we need that the set of values can be countable, and we have also countable summation.

To conclude, we provide a characterisation of parallel pushdown processes as a regular process communicating with a bag. In the case without priority operator, we need a form of asymmetric communication.

2 Preliminaries

As a common semantic framework we use the notion of a *labelled transition system*.

Definition 1. A labelled transition system is a quadruple $(\mathcal{S}, \mathcal{A}, \rightarrow, \downarrow)$, where

1. \mathcal{S} is a set of states;
2. \mathcal{A} is a set of actions, $\tau \notin \mathcal{A}$ is the unobservable or silent action;
3. $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \cup \{\tau\} \times \mathcal{S}$ is an $\mathcal{A} \cup \{\tau\}$ -labelled transition relation; and
4. $\downarrow \subseteq \mathcal{S}$ is the set of final or accepting states.

A process graph is a labelled transition system with a special designated root state \uparrow , i.e., it is a quintuple $(\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ such that $(\mathcal{S}, \mathcal{A}, \rightarrow, \downarrow)$ is a labelled transition system, and $\uparrow \in \mathcal{S}$. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$ and $s \downarrow$ for $s \in \downarrow$.

For $w \in \mathcal{A}^*$ we define $s \xrightarrow{w} t$ inductively, for all states s, t, u : first, $s \xrightarrow{\varepsilon} s$, and then, for $a \in \mathcal{A}$, if $s \xrightarrow{a} t$ and $t \xrightarrow{w} u$, then $s \xrightarrow{aw} u$, and if $s \xrightarrow{\tau} t$ and $t \xrightarrow{w} u$, then $s \xrightarrow{w} u$.

We see that τ -steps do not contribute to the string w . We write $s \longrightarrow t$ for there exists $a \in \mathcal{A} \cup \{\tau\}$ such that $s \xrightarrow{a} t$. Similarly, we write $s \twoheadrightarrow t$ for “there exists $w \in \mathcal{A}^*$ such that $s \xrightarrow{w} t$ ” and say that t is *reachable* from s . If $s \twoheadrightarrow t$ takes at least one step, we write $s \twoheadrightarrow^+ t$. We write $s \not\rightarrow t$ if there is no $t \in \mathcal{S}$ with $s \xrightarrow{a} t$. Finally, we write $s \xrightarrow{(a)} t$ for “ $s \xrightarrow{a} t$ or $a = \tau$ and $s = t$ ”.

By considering language equivalence classes of process graphs, we recover language equivalence as a semantics, but we can also consider other equivalence relations. Notable among these is *bisimilarity*.

Definition 2. Let $(\mathcal{S}, \mathcal{A}, \rightarrow, \downarrow)$ be a labelled transition system. A symmetric binary relation R on \mathcal{S} is a strong bisimulation if it satisfies the following conditions for every $s, t \in \mathcal{S}$ such that $s R t$ and for all $a \in \mathcal{A} \cup \{\tau\}$:

1. if $s \xrightarrow{a} s'$ for some $s' \in \mathcal{S}$, then there is a $t' \in \mathcal{S}$ such that $t \xrightarrow{a} t'$ and $s' R t'$; and

2. if $s \downarrow$, then $t \downarrow$.

If there is a strong bisimulation relating s and t we write $s \Leftrightarrow t$.

Sometimes we can use the *strong* version of bisimilarity defined above, which does not give special treatment to τ -labelled transitions. In general, when we do give special treatment to τ -labeled transitions, we use some form of *branching bisimulation* [21].

Definition 3. Let $(\mathcal{S}, \mathcal{A}, \rightarrow, \downarrow)$ be a labelled transition system. A symmetric binary relation R on \mathcal{S} is a branching bisimulation if it satisfies the following conditions for every $s, t \in \mathcal{S}$ such that $s R t$ and for all $a \in \mathcal{A} \cup \{\tau\}$:

1. if $s \xrightarrow{a} s'$ for some $s' \in \mathcal{S}$, then there are states $t', t'' \in \mathcal{S}$ such that $t \xrightarrow{\varepsilon} t'' \xrightarrow{(a)} t'$, $s R t''$ and $s' R t'$; and
2. if $s \downarrow$, then there is a state $t' \in \mathcal{S}$ such that $t \xrightarrow{\varepsilon} t'$ and $t' \downarrow$.

If there is a branching bisimulation relating s and t , we write $s \Leftrightarrow_b t$.

In this article, we use the finest branching bisimilarity called *divergence-preserving branching bisimilarity*, which was introduced in [21] (see also [20] and [23] for an overview of recent results).

Definition 4. A branching bisimulation R is divergence-preserving if for all $s, t \in \mathcal{S}$, whenever there is a infinite sequence of states s_0, s_1, \dots such that $s = s_0$, $s_i \xrightarrow{\tau} s_{i+1}$ and $s_i R t$ for all $i \geq 0$, then there is a state t' with $t \xrightarrow{\varepsilon}^+ t'$ and $s_i R t'$ for some $i \geq 0$. We write $s \Leftrightarrow_b^\Delta t$ if there is a divergence-preserving branching bisimulation relating s and t .

Theorem 1. Strong bisimilarity, branching bisimilarity and divergence-preserving branching bisimilarity are equivalence relations on labeled transition systems.

Proof. See [12] and [20]. □

A *process* is a divergence-preserving branching bisimilarity equivalence class of process graphs.

3 Parallel Pushdown Automata

We consider an abstract model of a computer with a memory in the form of a *bag*: the bag is an unordered multiset, an element can be removed from the bag (*get*), or an element can be added to it (*put*). Moreover, we can see when an element does not occur in the bag (a *failed get*). This is somewhat different than the definition in [24], who defined parallel pushdown automata by means of rewrite systems.

We claim our definition is a more natural one, when we compare with the definition of a pushdown automaton. In a pushdown automaton, we can pop the top element of the stack, or we can observe there is no top element (i.e., the stack is empty). In a bag, on the other hand, all elements are directly accessible. We can pop (remove) any element, or observe this element does not occur. Just observing that the bag is empty, does not lead to a satisfactory theory (see [25]).

We use notation \mathcal{D}^\cup for the set of bags with elements from \mathcal{D} . We use two disjoint copies of \mathcal{D} , $\mathcal{D}^+ = \{(d, +) \mid d \in \mathcal{D}\}$ and $\mathcal{D}^- = \{(d, -) \mid d \in \mathcal{D}\}$. We write $+d$ instead of $(d, +)$ and $-d$ instead of $(d, -)$. We denote $\mathcal{D}^\pm = \mathcal{D}^+ \cup \mathcal{D}^-$.

Definition 5 (parallel pushdown automaton). A parallel pushdown automaton M is a sextuple $(\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ where:

1. \mathcal{S} is a finite set of states,

2. \mathcal{A} is a finite input alphabet, $\tau \notin \mathcal{A}$ is the unobservable step,
3. \mathcal{D} is a finite data alphabet,
4. $\rightarrow \subseteq \mathcal{S} \times (\mathcal{A} \cup \{\tau\}) \times \mathcal{D}^\pm \times \mathcal{D}^\mathbb{N} \times \mathcal{S}$ is a finite set of transitions or steps,
5. $\uparrow \in \mathcal{S}$ is the initial state, in the initial state the bag is empty,
6. $\downarrow \subseteq \mathcal{S}$ is the set of final or accepting states.

If $(s, a, +d, x, t) \in \rightarrow$ with $d \in \mathcal{D}$, we write $s \xrightarrow{a[+d/x]} t$, and this means that the machine, when it is in state s and d is an element of the bag, can consume input symbol a , replace d by the bag x and thereby move to state t . On the other hand, we write $s \xrightarrow{a[-d/x]} t$, and this means that the machine, when it is in state s and the bag does not contain a d , can consume input symbol a , put x in the bag and thereby move to state t . In steps $s \xrightarrow{\tau[+d/x]} t$ and $s \xrightarrow{\tau[-d/x]} t$, no input symbol is consumed, only the bag is modified.

Notice that we defined a parallel pushdown automaton in such a way that it can be detected whether or not an element occurs in the bag.

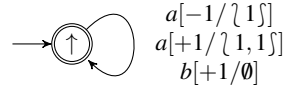


Figure 1: Parallel pushdown automaton of a counter.

For example, consider the parallel pushdown automaton depicted in Figure 1. It represents the process that can start to read an a , and after it has read at least one a , can read additional a 's but can also read b 's. Upon acceptance, it will have read up to as many b 's as it has read a 's. Interpreting symbol a as an increment and b as a decrement, we can see this process as a *counter*.

We do not consider the language of a parallel pushdown automaton, but rather consider the process, i.e., the divergence-preserving branching bisimilarity equivalence class of the process graph of a parallel pushdown automaton. A state of this process graph is a pair (s, x) , where $s \in \mathcal{S}$ is the current state and $x \in \mathcal{D}^\mathbb{N}$ is the current contents of the bag. In the initial state, the bag is empty. In a final state, acceptance can take place irrespective of the contents of the bag. The transitions in the process graph are labeled by the inputs of the pushdown automaton or τ .

Definition 6. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a parallel pushdown automaton. The process graph $\mathcal{P}(M) = (\mathcal{S}_{\mathcal{P}(M)}, \mathcal{A}, \rightarrow_{\mathcal{P}(M)}, \uparrow_{\mathcal{P}(M)}, \downarrow_{\mathcal{P}(M)})$ associated with M is defined as follows:

1. $\mathcal{S}_{\mathcal{P}(M)} = \{(s, x) \mid s \in \mathcal{S} \text{ \& } x \in \mathcal{D}^\mathbb{N}\};$
2. $\rightarrow_{\mathcal{P}(M)} \subseteq \mathcal{S}_{\mathcal{P}(M)} \times \mathcal{A} \cup \{\tau\} \times \mathcal{S}_{\mathcal{P}(M)}$ is the least relation such that for all $s, s' \in \mathcal{S}$, $a \in \mathcal{A} \cup \{\tau\}$, $d \in \mathcal{D}$ and $x, x' \in \mathcal{D}^\mathbb{N}$ we have

$$(s, [d] \cup x) \xrightarrow{a}_{\mathcal{P}(M)} (s', x' \cup x) \text{ if, and only if, } s \xrightarrow{a[+d/x]} s' ;$$

$$(s, x) \xrightarrow{a}_{\mathcal{P}(M)} (s', x' \cup x) \text{ if, and only if, there exists } d \notin x \text{ such that } s \xrightarrow{a[-d/x]} s' ;$$

3. $\uparrow_{\mathcal{P}(M)} = (\uparrow, \emptyset);$

$$4. \downarrow \mathcal{P}(M) = \{(s, x) \mid s \in \downarrow \text{ \& } x \in \mathcal{D}^{\mathcal{U}}\}.$$

To distinguish, in the definition above, the set of states, the transition relation, the initial state and the set of accepting states of the parallel pushdown automaton from similar components of the associated process graph, we have attached a subscript $\mathcal{P}(M)$ to the latter. In the remainder of this paper, we will suppress the subscript whenever it is already clear from the context whether a component of the parallel pushdown automaton or its associated process graph is meant.

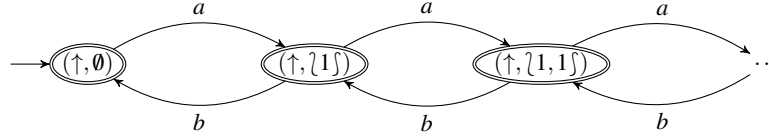


Figure 2: The process graph of the counter.

Figure 2 depicts the process graph associated with the pushdown automaton depicted in Figure 1.

In language equivalence, the definition of acceptance in parallel pushdown automata leads to the same set of languages when we define acceptance by final state (as we do here) and when we define acceptance by empty bag (not considering final states). In bisimilarity, these notions are different: acceptance by empty bag yields a smaller set of processes than acceptance by final state. Note that the process graph in Figure 2 has infinitely many non-bisimilar final states. It is, therefore, not bisimilar to the process graph of a parallel pushdown automaton that accepts by empty bag. For details, see [6, 25].

In order to illustrate that we can realise acceptance by empty bag also if we define acceptance by final state, consider the parallel pushdown automaton of the counter that only accepts when empty in Figure 3. We need three states to realise a process graph that is divergence-preserving branching bisimilar to the process graph in Figure 2, but with only the initial state accepting.

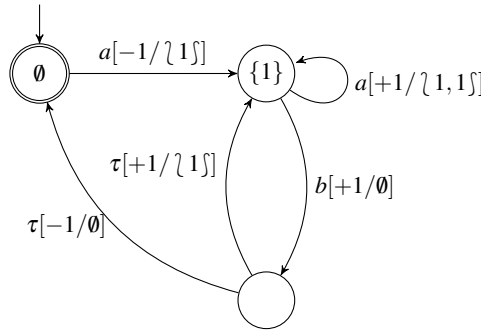


Figure 3: Counter only accepting when empty.

An important example of a parallel pushdown automaton is the bag process itself. We consider the bag that is always accepting in Figure 4. For a given data set \mathcal{D} , it has actions $ins(d)$ (insert), $rem(d)$ (remove) and $show(\neg d)$ (show there is no d). For each $d \in \mathcal{D}$, there are the transitions shown. We need the $show(\neg d)$ transitions later, to indicate that no (further) remove transitions are possible. We use this in Section 7.

A parallel pushdown automaton has only finitely many transitions, so there is a maximum number of transitions from a given state, called its *branching degree*. Then, also the associated process graph has a

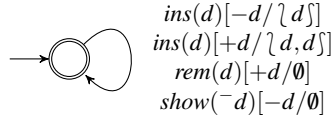


Figure 4: Parallel pushdown automaton of an always accepting bag.

branching degree, that cannot be larger than the branching degree of the underlying parallel pushdown automaton. Thus, in a process graph associated with a parallel pushdown automaton, the branching is always *bounded*. However, it is possible that its divergence-preserving branching bisimilarity equivalence class contains a process graph that is infinitely branching. Consider the following example.

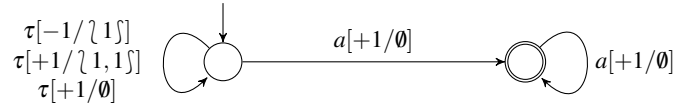


Figure 5: Parallel pushdown automaton with a divergence.

Example 1. Consider the parallel pushdown automaton in Figure 5. It has a process graph consisting of two infinite rows of nodes. The nodes in the top row all have a divergence, and modulo a divergence-preserving branching bisimilarity can collapse into one node, as shown in the process graph in Figure 6. This top node still needs a divergent τ loop.

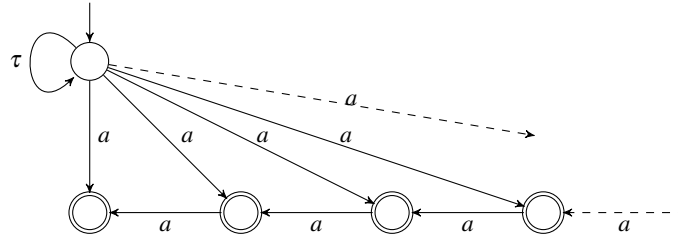


Figure 6: Process graph divergence-preserving branching bisimilar to the parallel pushdown automaton with divergence.

4 Parallel Processes

In the process setting, a commutative context-free grammar is a process algebra comprising actions, choice, parallelism and recursion. We start out from the process algebra PA of [14], but with sequential composition restricted to action prefixing, and then extended with constants $\mathbf{0}$ and $\mathbf{1}$ to denote deadlock and acceptance. We call this process algebra $\text{BPP}^{\mathbf{01}}$, for Basic Parallel Processes with $\mathbf{0}$ and $\mathbf{1}$.

Let \mathcal{A} be a set of *actions* and $\tau \notin \mathcal{A}$ the *silent action*, symbols denoting atomic events, and let \mathcal{P} be a finite set of *process identifiers*. The sets \mathcal{A} and \mathcal{P} serve as parameters of the process theory that we shall introduce below. We use symbols a, b, \dots , possibly indexed, to range over $\mathcal{A} \cup \{\tau\}$, symbols X, Y, \dots , possibly indexed, to range over \mathcal{P} . The set of *parallel process expressions* is generated by the following grammar ($a \in \mathcal{A} \cup \{\tau\}$, $X \in \mathcal{P}$):

$$p ::= \mathbf{0} \mid \mathbf{1} \mid a.p \mid p + p \mid p \parallel p \mid X.$$

The constants $\mathbf{0}$ and $\mathbf{1}$ respectively denote the *deadlocked* (i.e., inactive but not accepting) process and the *accepting* process. For each $a \in \mathcal{A} \cup \{\tau\}$ there is a unary action prefix operator $a._$. We fix a finite data set \mathcal{D} , and actions can be parametrised with a data element. The binary operators $+$ and \parallel denote alternative composition and parallel composition, respectively. We adopt the convention that $a._$ binds strongest and $+$ binds weakest.

For a (possibly empty) sequence p_1, \dots, p_n we inductively define $\sum_{i=1}^n p_i = \mathbf{0}$ if $n = 0$ and $\sum_{i=1}^n p_i = (\sum_{i=1}^{n-1} p_i) + p_n$ if $n > 0$. Likewise, for a sequence p_1, \dots, p_n we inductively define $\parallel_{i=1}^n p_i = \mathbf{1}$ if $n = 0$ and $\parallel_{i=1}^n p_i = (\parallel_{i=1}^{n-1} p_i) \parallel p_n$ if $n > 0$.

A recursive specification over parallel process expressions is a mapping Γ from \mathcal{P} to the set of parallel process expressions. The idea is that the process expression p associated with a process identifier $X \in \mathcal{P}$ by Γ *defines* the behaviour of X . We prefer to think of Γ as a collection of *defining equations* $X \stackrel{\text{def}}{=} p$, exactly one for every $X \in \mathcal{P}$. We shall, throughout the paper, presuppose a recursive specification Γ defining the process identifiers in \mathcal{P} , and we shall usually simply write $X \stackrel{\text{def}}{=} p$ for $\Gamma(X) = p$. Note that, by our assumption that \mathcal{P} is finite, Γ is finite too.

$$\begin{array}{c}
\frac{}{\mathbf{1} \downarrow} \quad \frac{}{a.p \xrightarrow{a} p} \\
\frac{p \downarrow}{(p+q) \downarrow} \quad \frac{q \downarrow}{(p+q) \downarrow} \quad \frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} \quad \frac{q \xrightarrow{a} q'}{p+q \xrightarrow{a} q'} \\
\frac{p \downarrow \quad q \downarrow}{p \parallel q \downarrow} \quad \frac{p \xrightarrow{a} p'}{p \parallel q \xrightarrow{a} p' \parallel q} \quad \frac{q \xrightarrow{a} q'}{p \parallel q \xrightarrow{a} p \parallel q'} \\
\frac{p \xrightarrow{a} p' \quad X \stackrel{\text{def}}{=} p}{X \xrightarrow{a} p'} \quad \frac{p \downarrow \quad X \stackrel{\text{def}}{=} p}{X \downarrow}
\end{array}$$

Figure 7: Operational semantics for parallel process expressions.

We associate behaviour with process expressions by defining, on the set of process expressions, a unary acceptance predicate \downarrow (written postfix) and, for every $a \in \mathcal{A} \cup \{\tau\}$, a binary transition relation \xrightarrow{a} (written infix), by means of the transition system specification presented in Figure 7.

By means of these rules, the set of parallel process expressions turns into a labelled transition system, so we have strong bisimilarity, branching bisimilarity and divergence-preserving branching bisimilarity on parallel process expressions.

The operational rules presented in Fig 7 are in the so-called *path format* from which it immediately follows that strong bisimilarity is a congruence [11]. (Divergence-preserving) branching bisimilarity, however, is not a congruence, but by adding a rootedness condition we get rooted (divergence-preserving)

branching bisimilarity which is a congruence [21]. As we will not use equational reasoning in this paper, we will not use the rootedness condition.

Some recursive specifications over BPP^{01} will give processes that cannot be the process of a commutative pushdown automaton.

Example 2. Consider the recursive equation

$$X \stackrel{\text{def}}{=} a.1 + X \parallel b.1 .$$

We show the process graph generated by the operational rules in Figure 8. As $X \xrightarrow{a} 1$, we get $X \parallel b.1 \xrightarrow{a} 1 \parallel b.1 = b.1$ and so $X \xrightarrow{a} b.1$. Continuing like this we get $X \xrightarrow{a} b^n.1$ for each n . Note we also have $X \xrightarrow{b} X$.

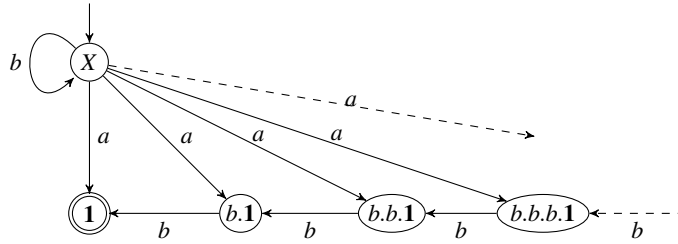


Figure 8: Process graph of the recursive specification of Example 2.

Theorem 2. The process graph of Figure 8 is not divergence-preserving branching bisimilar to the process graph of any parallel pushdown automaton.

To exclude recursive specifications over BPP^{01} that give rise to process graphs with states that necessarily have infinitely many outgoing transitions, it suffices to formulate a standard *guardedness* condition for recursive specifications.

Definition 7. We say a recursive specification is *weakly guarded* if every occurrence of a process identifier in the definition of some (possibly different) process identifier occurs within the scope of an action prefix from $\mathcal{A} \cup \{\tau\}$, and *strongly guarded* if every occurrence of a process identifier in the definition of some process identifier occurs within the scope of an action prefix from \mathcal{A} .

We will show that every finite weakly guarded recursive specification over BPP^{01} yields a parallel pushdown automaton. We first consider a couple of examples.

Example 3. Consider the recursive specification

$$AC \stackrel{\text{def}}{=} 1 + a.(AC \parallel (1 + b.1)) .$$

By following the operational rules, we obtain a process graph that is bisimilar to the one shown in Figure 2, and thus we obtain the parallel pushdown automaton in Figure 1. This is the always accepting counter.

If, instead, we use the equation

$$EC \stackrel{\text{def}}{=} 1 + a.(EC \parallel b.1) .$$

we get the counter that only accepts when it is empty, see the parallel pushdown automaton in Figure 3. Now we can generalize the equation of AC to the following

$$AB \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} \text{ins}(d). (AB \parallel (\mathbf{1} + \text{rem}(d).\mathbf{1})) .$$

We see that this is a specification of the bag. However, this bag does not have the $\text{show}(\bar{d})$ actions to signal that a d does not occur in the bag. In fact, we will show that there is no finite weakly guarded specification over $\text{BPP}^{\mathbf{01}}$ that gives rise to the parallel pushdown automaton in Figure 4. For now, we first look at the other direction, to show that a finite weakly guarded specification over $\text{BPP}^{\mathbf{01}}$ yields the process of a parallel pushdown automaton.

Since we have weakly guarded recursion, we can bring every $\text{BPP}^{\mathbf{01}}$ -term into head normal form. The following result uses strong bisimulation, not branching bisimulation.

Theorem 3. *Let Γ be a weakly guarded $\text{BPP}^{\mathbf{01}}$ -specification. Every process expression p can be brought into head normal form, i.e. there are $a_i \in \mathcal{A} \cup \{\tau\}$ and process expressions p_i such that*

$$p \Leftrightarrow (\mathbf{1} +) \sum_{i=1}^n a_i.p_i$$

where the $\mathbf{1}$ summand may or may not occur.

As a result, we can bring every guarded recursive specification into Greibach Normal Form.

Definition 8. *A recursive specification Γ is in Greibach Normal Form if every equation has the form $X \stackrel{\text{def}}{=} (\mathbf{1} +) \sum_{i=1}^n a_i.\xi_i$ for actions $a_i \in \mathcal{A} \cup \{\tau\}$, where each ξ_i is a parallel composition of identifiers of Γ , and $n \geq 0$.*

Theorem 4. *Let Γ be a weakly guarded $\text{BPP}^{\mathbf{01}}$ -specification over identifiers \mathcal{P} . Then there is a finite set of identifiers \mathcal{Q} with $\mathcal{P} \subseteq \mathcal{Q}$ and a recursive specification in Greibach Normal Form Δ over identifiers \mathcal{Q} such that for all $X, Y \in \mathcal{P}$ we have $X \Leftrightarrow Y$ with respect to Γ if, and only if, $X \Leftrightarrow Y$ with respect to Δ .*

Now we are ready to prove the main result of this section.

Theorem 5. *Every weakly guarded recursive specification over $\text{BPP}^{\mathbf{01}}$ has a process graph that is divergence-preserving branching bisimilar to the process graph of a parallel pushdown automaton.*

Proof. Without loss of generality, we can assume the specification is in Greibach Normal Form. Then, all states in the generated process graph are given by a parallel composition of identifiers of the specification. Divide the identifiers of the specification into the accepting identifiers \mathbb{A} (that have a $\mathbf{1}$ summand) and the non-accepting identifiers \mathbb{N} that do not have a $\mathbf{1}$ summand. A state in the generated process graph is accepting iff all identifiers in the parallel composition are from \mathbb{A} . In the parallel pushdown automaton to be constructed, we need to keep track when the last element of \mathbb{N} is removed, in order to switch to an accepting state.

We take the data set \mathcal{D} to be the set of identifiers of the specification. S is the initial identifier. In the states of the parallel pushdown automaton, we will encode whether or not there is an element of \mathbb{N} , so there is a state for each subset (not multiset) of \mathbb{N} . As inspiration, we use the parallel pushdown automaton of the counter that only accepts when empty in Figure 3.

The states of the parallel pushdown automaton are as follows:

- N , for $N \subseteq \mathbb{N}$ (a subset, not a submultiset). The bisimulation will relate state $(N, x \cup y)$ for any multiset $x \in \mathbb{A}^{\uplus}$ to the parallel composition of the elements of $x \cup y$, if y contains all elements of N and no other non-accepting identifiers.

- There is an auxiliary state N_X , for each $N \subseteq \mathbb{N}$ and $X \in N$.

The initial state is \emptyset . \emptyset is the only accepting state.

Now the steps:

1. $\emptyset \xrightarrow{a[-S/\xi]} \emptyset$, whenever S has a summand $a.\xi$ and ξ has only accepting identifiers.
2. $\emptyset \xrightarrow{a[-S/\xi]} N$ whenever S has a summand $a.\xi$ and ξ has at least one non-accepting identifier. N collects the non-accepting identifiers of ξ .
3. $N \xrightarrow{a[+X/\xi]} N'$, whenever $X \notin N$ is accepting, X has a summand $a.\xi$ and N' unites N with the non-accepting identifiers of ξ .
4. $N \xrightarrow{a[+X/\xi]} N'$, whenever $X \in N$ has a summand $a.\xi$, ξ has at least one non-accepting identifier and N' unites N with the non-accepting identifiers of ξ .
5. if $X \in N \subseteq \mathbb{N}$, X has a summand $a.\xi$ with all identifiers in ξ accepting, add three transitions

$$N \xrightarrow{a[+X/\xi]} N_X \xrightarrow{\tau[+X/\lambda X]} N.$$

and

$$N_X \xrightarrow{\tau[-X/\emptyset]} N - \{X\}.$$

Notice that all the added τ -steps in the transition system are inert, as from the added N_X states exactly one transition can be taken, depending on whether or not X occurs in the parallel composition. \square

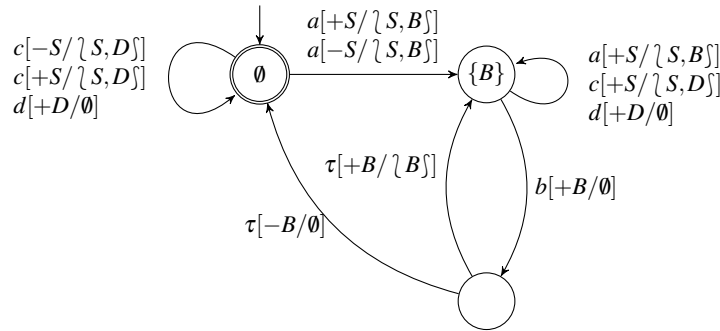


Figure 9: Parallel pushdown automaton of Example 4.

Example 4. Let the guarded recursive specification Γ be given as follows. Notice it is in Greibach Normal Form, and $\mathbb{N} = \{B\}$, $\mathbb{A} = \{S, D\}$.

$$S \stackrel{\text{def}}{=} \mathbf{1} + a.(S \parallel B) + c.(S \parallel D) \quad B \stackrel{\text{def}}{=} b.\mathbf{1} \quad D \stackrel{\text{def}}{=} \mathbf{1} + d.\mathbf{1}$$

Following the proof of Theorem 5 results in the parallel pushdown automaton shown in Figure 9. Notice the similarity with the parallel pushdown automaton shown in Figure 3.

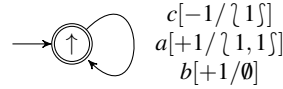


Figure 10: Parallel pushdown automaton of a counter with a change.

As we stated, the other direction does not work: we cannot find a finite weakly guarded BPP^{01} -specification for the one-state parallel pushdown automaton of the always accepting bag in Figure 4. It is technically somewhat simpler to prove such a negative result for the one-state parallel pushdown automaton shown in Figure 10.

Theorem 6. *For the one-state parallel pushdown automaton in Figure 10 there is no finite weakly guarded BPP^{01} specification such that their process graphs are divergence-preserving branching bisimilar.*

Now let us reconsider the parallel pushdown automaton of the bag. The problem is, that a $\text{show}(\neg d)$ -action can only occur if no $\text{rem}(d)$ -action can occur. Thus, in a sum context, the $\text{show}(\neg d)$ action should have *lower priority* than the $\text{rem}(d)$ action. In general, we assume we have a partial ordering $<$ on $\mathcal{A} \cup \{\tau\}$, where $a < b$ means that a has lower priority than b , satisfying that $\tau < a$ never holds, and whenever $a < b$ then also $a < \tau$. The *priority operator* θ will implement the priorities, and is given by the operational rules in Figure 11, see [3, 1]. Notice that the second rule for the priority operator uses a negative premise. Transition system specifications with negative premises may, in general, not define a unique transition relation that agrees with provability from the transition system specification, but our restriction to weakly guarded specifications eliminates this problem [22, 15, 19]. Also, note that (rooted) branching bisimilarity is not compatible with the priority operator, but divergence-preserving branching bisimilarity is [17].

$$\begin{array}{c}
 \frac{p \downarrow}{\theta(p) \downarrow} \quad \frac{p \xrightarrow{a} p' \quad \forall b > a \quad p \not\xrightarrow{b}}{\theta(p) \xrightarrow{a} \theta(p')} \\
 \\
 \frac{p \xrightarrow{a} p'}{\rho_f(p) \xrightarrow{f(a)} \rho_f(p')} \quad \frac{p \downarrow}{\rho_f(p) \downarrow}
 \end{array}$$

Figure 11: Operational semantics for priorities and renaming.

With the help of this operator, we can give the following specification of the always accepting bag, assuming $\text{show}(\neg d) < \text{rem}(d)$.

$$\text{ABag} \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} \text{ins}(d). \theta(\text{ABag} \parallel (\mathbf{1} + \text{rem}(d). \mathbf{1})) + \sum_{d \in \mathcal{D}} \text{show}(\neg d). \text{ABag} .$$

For the parallel pushdown automaton in Figure 10, it is enough to take $c < b$. In general, we need to put a priority ordering on the labels of a parallel pushdown automaton. This may not be possible if some labels are the same, or if a τ occurs as a label. Therefore, we need to ensure all the labels in the parallel pushdown automaton are distinct and from \mathcal{A} , in order to be able to impose a priority ordering.

Thus, given a parallel pushdown automaton, we consider another parallel pushdown automaton with distinct labels, solve the problem for that automaton, and then rename the labels again to their original values. This renaming is done by a *renaming operator* p_f , where f is any function on $\mathcal{A} \cup \{\tau\}$ satisfying $f(\tau) = \tau$. The renaming operator has the operational rules shown in Figure 11, see [2, 1].

Now we extend $\text{BPP}^{\mathbf{01}}_\theta$ to include the priority operator and renaming operators. We call this extended algebra $\text{BPP}^{\mathbf{01}}_\theta$. Theorem 5 can be extended to $\text{BPP}^{\mathbf{01}}_\theta$.

Theorem 7. *Every weakly guarded recursive specification over $\text{BPP}^{\mathbf{01}}_\theta$ has a process graph that is divergence-preserving branching bisimilar to the process graph of a parallel pushdown automaton.*

In the other direction, it works for every one-state parallel pushdown automaton.

Theorem 8. *For every one-state parallel pushdown automaton there is a finite weakly guarded $\text{BPP}^{\mathbf{01}}_\theta$ specification such that their process graphs are divergence-preserving branching bisimilar.*

Thus, for all one-state parallel pushdown automata we can find a specification in $\text{BPP}^{\mathbf{01}}_\theta$. This result does not extend to parallel pushdown automata with more than one state.

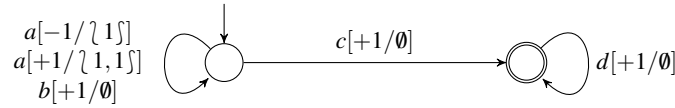


Figure 12: Parallel pushdown automaton that cannot be specified in $\text{BPP}^{\mathbf{01}}_\theta$.

Theorem 9. *There is a parallel pushdown automaton with two states, such that there is no weakly guarded $\text{BPP}^{\mathbf{01}}_\theta$ specification with the same process.*

In order to recover the correspondence between parallel pushdown automata and parallel process algebra, we need to add communication with value passing.

5 Communicating processes

We extend the basic parallel processes $\text{BPP}^{\mathbf{01}}$ by adding a communication mechanism. We assume we have a finite set of communication ports \mathcal{C} , and that each parametrised action $c(d)$ is the result of the communication of the send action $c!d$ and the receive action $c?d$. The data set \mathcal{D} is finite. Define $\text{COM}_C = \{c!d, c?d \mid c \in C, d \in \mathcal{D}\}$ for a set of ports $C \subseteq \mathcal{C}$. The *encapsulation operator* $\partial_C()$ will block the send and receive actions from the set of ports C . The *abstraction operator* τ_C will hide all parametrised actions from the set of ports C .

The process algebra $\text{BCP}^{\mathbf{01}}$ extends $\text{BPP}^{\mathbf{01}}$ with communication, encapsulation and abstraction; likewise, $\text{BCP}^{\mathbf{01}}_\theta$ extends $\text{BPP}^{\mathbf{01}}_\theta$. Using communication, we can specify communicating bags: ABag^{io} defines the always accepting bag with input port i and output port o , while EBag^{io} defines the bag with input port i and output port o that is only accepting when it is empty. We see both the $\text{rem}(d)$ actions and the $\text{show}(\neg d)$ actions as outputs.

$$\text{ABag}^{io} \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} i?d. \theta(\text{ABag}^{io} \parallel (\mathbf{1} + o!(^+d).\mathbf{1})) + \sum_{d \in \mathcal{D}} o!(^-d).\text{ABag}^{io}$$

$$\begin{array}{c}
\frac{p \xrightarrow{c!d} p' \quad q \xrightarrow{c?d} q'}{p \parallel q \xrightarrow{c(d)} p' \parallel q' \quad q \parallel p \xrightarrow{c(d)} q' \parallel p'} \\
\frac{p \downarrow}{\partial_C(p) \downarrow} \quad \frac{p \xrightarrow{a} p' \quad a \notin COM_C}{\partial_C(p) \xrightarrow{a} \partial_C(p')} \\
\frac{p \downarrow}{\tau_C(p) \downarrow} \quad \frac{p \xrightarrow{c(d)} p' \quad c \in C}{\tau_C(p) \xrightarrow{\tau} \tau_C(p')} \quad \frac{p \xrightarrow{a} p' \quad a \neq c(d) \text{ for } c \in C}{\tau_C(p) \xrightarrow{a} \tau_C(p')}
\end{array}$$

Figure 13: Operational semantics for communication, encapsulation and abstraction.

$$EBag^{io} \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.\theta(EBag^{io} \parallel o!(^+d).\mathbf{1}) + \sum_{d \in \mathcal{D}} o!(-d).EBag^{io}.$$

In Section 7, we will use the communicating always accepting bag to make the communication between a finite control and a memory in the form of a bag explicit. Here, we restate a classical result: putting two bags with unrestricted capacity in series will again be a bag with unrestricted capacity.

$$ABag^{io} \xleftrightarrow{\Delta}_b \tau_{\{\ell\}}(\partial_{\{\ell\}}(ABag^{i\ell} \parallel ABag^{\ell o})) \quad EBag^{io} \xleftrightarrow{\Delta}_b \tau_{\{\ell\}}(\partial_{\{\ell\}}(EBag^{i\ell} \parallel EBag^{\ell o}))$$

Again, we can bring every BCP_{θ}^{01} -term into head normal form.

Theorem 10. *Let Γ be a weakly guarded BCP_{θ}^{01} -specification. Every process expression p can be brought into head normal form, i.e. there are $a_i \in \mathcal{A} \cup \{\tau\}$ and process expressions p_i such that*

$$p \xleftrightarrow{\Delta} (\mathbf{1}+) \sum_{i=1}^n a_i.p_i$$

where the $\mathbf{1}$ summand may or may not occur.

Proof. By induction on the structure of p (see [1]). We use Milner's Expansion Law, now with communication. \square

As a result, we can bring every guarded recursive specification into Greibach Normal Form.

Definition 9. *A recursive specification Γ over BCP_{θ}^{01} is in Greibach Normal Form if every equation has the form*

$$X \stackrel{\text{def}}{=} (\mathbf{1}+) \sum_{i=1}^n a_i.\tau_C(\partial_C(\theta(\xi_i))).$$

for actions $a_i \in \mathcal{A} \cup \{\tau\}$, where each ξ_i is a parallel composition of identifiers of Γ , and $n \geq 0$.

Theorem 11. *Let Γ be a weakly guarded BCP_{θ}^{01} -specification over identifiers \mathcal{P} . Then there is a finite set of identifiers \mathcal{Q} with $\mathcal{P} \subseteq \mathcal{Q}$ and a recursive specification in Greibach Normal Form Δ over identifiers \mathcal{Q} such that for all $X, Y \in \mathcal{P}$ we have $X \xleftrightarrow{\Delta} Y$ with respect to Γ if, and only if, $X \xleftrightarrow{\Delta} Y$ with respect to Δ .*

6 The full correspondence

With the help of value-passing communication, we can now establish our main result: for every parallel pushdown automaton we can find a specification in $\text{BCP}_{\theta}^{\mathbf{01}}$. The communication actions will pass on the information of the current state of the parallel pushdown automaton. Let us look at the parallel pushdown automaton in Figure 12, that did not have a finite specification in $\text{BCP}_{\theta}^{\mathbf{01}}$.

Example 5. Consider the parallel pushdown automaton in Figure 12, with initial state s and accepting state t . We need to distinguish between the two a -actions on state s , let us call them a^- and a^+ . Action a^- has lower priority than a^+ . We just need to communicate in which of the states we are, so we use actions $p!s, p!t, p?s$ and $p?t$ for some communication port p . Actions $p(s), p(t)$ have the highest priority. As all components in a parallel composition need the state information, we need to communicate the state information repeatedly, until all components are brought into the right position. After this, we need to exit the communication process. Define $P_s \stackrel{\text{def}}{=} \mathbf{1} + p!s.P_s + \text{exit}.\mathbf{1}$ and $P_t \stackrel{\text{def}}{=} \mathbf{1} + p!t.P_t + \text{exit}.\mathbf{1}$ and $p(s) > \text{exit}, p(t) > \text{exit}$ and $\text{exit} > e$ for $e \in \{a^+, a^-, b, c, d\}$.

$$\begin{aligned} S &\stackrel{\text{def}}{=} a^-.\tau_p(\partial_p(\theta(P_s \parallel X_0 \parallel X_1))) \\ X_1 &\stackrel{\text{def}}{=} p?s.(a^+.(P_s \parallel X_1 \parallel X_1) + b.P_s + c.P_t) + p?t.(\mathbf{1} + d.P_t) \\ X_0 &\stackrel{\text{def}}{=} p?s.a^-(P_s \parallel X_1) + p?t.\mathbf{1} \end{aligned}$$

Theorem 12. For every parallel pushdown automaton there is a finite weakly guarded specification over $\text{BCP}_{\theta}^{\mathbf{01}}$ such that their process graphs are divergence-preserving branching bisimilar.

Theorem 13. For every finite weakly guarded $\text{BCP}_{\theta}^{\mathbf{01}}$ -specification there is a parallel pushdown automaton such that their process graphs are divergence-preserving branching bisimilar.

Proof. As again, we can bring a finite weakly guarded $\text{BCP}_{\theta}^{\mathbf{01}}$ -specification into Greibach Normal Form, this proof goes along the lines of the proof of Theorem 5. The only difference is, is that because of a communication action, two non-accepting identifiers can be removed from a parallel composition at the same time. \square

To conclude this section, we consider how far we can go with communication, but without priorities. In the bag, not using priorities, it is required to count the number of remove transitions, in order to know when a show absence transition is enabled. We can do this counting in the communication actions, but then the parametrising data set \mathcal{D} becomes infinite, and the specification uses countable sums. This is a drawback, in our opinion.

Theorem 14. For every parallel pushdown automaton there is a finite weakly guarded specification over $\text{BCP}^{\mathbf{01}}$ extended with infinite choice, such that their process graphs are divergence-preserving branching bisimilar.

Example 6. Consider the parallel pushdown automaton in Figure 10, with state s . As the data set is a singleton, we just need to count the number of 1's, and we use natural numbers as parameters.

$$\begin{aligned} S &\stackrel{\text{def}}{=} \mathbf{1} + c.\tau_s(\partial_s(s!1.\mathbf{1} \parallel X_{\{1\}})) \\ X_{\{1\}} &\stackrel{\text{def}}{=} s?1.(\mathbf{1} + (a.s!2.\mathbf{1} \parallel X_{\{1\}} \parallel X_{\{1\}}) + (b.s!0.\mathbf{1} \parallel X_{\emptyset})) + \\ &\quad + \sum_{n \geq 2} s?n.(\mathbf{1} + (a.s!(n+1).\mathbf{1} \parallel X_{\{1\}} \parallel X_{\{1\}}) + b.s!(n-1).\mathbf{1}) \\ X_{\emptyset} &\stackrel{\text{def}}{=} s?0.(\mathbf{1} + (c.s!1.\mathbf{1} \parallel X_{\{1\}})). \end{aligned}$$

7 A characterisation

A computer shows interaction between a finite control and the memory. The finite control can be represented by a regular process (a finite automaton). In [7], we considered a memory in the form of a stack, and we established that a pushdown process can be characterised as a regular process communicating with a stack. Here, we have a memory in the form of a bag, and we can establish a similar result.

Theorem 15. *A process p is a parallel push-down process, if and only there is a regular process q such that $p \xleftrightarrow{\Delta}_b \tau_{\{i,o\}}(\partial_{\{i,o\}}(q \parallel ABag^{io}))$.*

In [8], it was established that every parallel process expression is rooted branching bisimilar to a regular process communicating with a process that is defined as follows:

$$AB^{io} \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.AB^{io} \parallel (\mathbf{1} + o!d.\mathbf{1}).$$

By Theorem 5, every parallel process expression denotes a parallel pushdown process, and so Theorem 15 can be applied to get a characterisation in terms of a regular process that communicates with $ABag^{io}$, the always accepting bag. Note, however, that $ABag^{io}$ uses the priority operator to facilitate the show absence actions. With the process AB^{io} from [8] we can establish a similar result, but we have to replace receiving an element from a bag by *getting* an element from a bag, where failure to get a particular element can be detected (see [13, 1]). Thus, for $d \in \mathcal{D}$, we add elements $c??^+d$ (a get) and $c??^-d$ (a failed get) to COM_C with $c \in C$, and add $c \times d$ for a failed communication that will also be hidden by τ_C with $c \in C$. We add the operational rules in Figure 14. Notice the second rule uses a negative premise. Still, as we use weakly guarded recursion, and the rules are in so-called *panth* format, we obtain a labelled transition system, see [26].

$$\frac{p \xrightarrow{c??^+d} p' \quad q \xrightarrow{c!d} q'}{p \parallel q \xrightarrow{c(d)} p' \parallel q' \quad q \parallel p \xrightarrow{c(d)} q' \parallel p'} \quad \frac{p \xrightarrow{c??^-d} p' \quad q \xrightarrow{c!d} q'}{p \parallel q \xrightarrow{c \times d} p' \parallel q \quad q \parallel p \xrightarrow{c \times d} q \parallel p'}$$

Figure 14: Operational semantics for get communication.

Theorem 16. *A process p is a parallel push-down process, if and only there is a regular process q such that $p \xleftrightarrow{\Delta}_b \tau_{\{i,o\}}(\partial_{\{i,o\}}(q \parallel AB^{io}))$.*

We see the bag is the prototypical parallel pushdown process, as all parallel pushdown processes can be realised as a regular process communicating with a bag. A bag is not a pushdown process. Likewise, the stack is the prototypical pushdown process, but not a parallel pushdown process. The counter is not a regular process, but it is both a pushdown process and a parallel pushdown process. Figure 15 provides a complete picture. The queue is not a pushdown process and also not a parallel pushdown process. It is the prototypical executable process, as every executable process can be characterized as a regular process communicating with an always accepting queue. By using a queue, the Turing tape can be defined.

8 Conclusion

In language theory, the set of languages given by a parallel pushdown automaton coincides with the set of languages given by a commutative context-free grammar. A language is an equivalence class

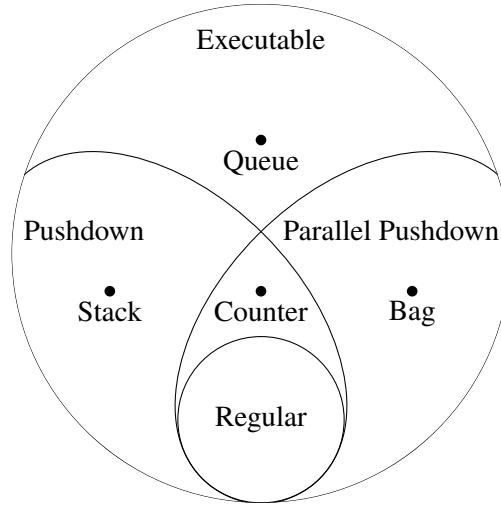


Figure 15: Classification of Executable, Pushdown, Parallel Pushdown, and Regular processes and the prototypical processes Queue, Bag, Stack and Counter.

of process graphs modulo language equivalence. A process is an equivalence class of process graphs modulo divergence-preserving branching bisimulation.

This paper solves the question how we can characterize the set of processes given by a parallel pushdown automaton. In the process setting, a commutative context-free grammar is a process algebra with actions, choice, parallel composition and finite recursion. We need to limit to weakly guarded recursion in the process setting. Starting out from the seminal process algebra PA of [14] with sequential composition restricted to action prefixing, we need to add constants for the inactive and accepting process and for the inactive non-accepting (deadlock) process. Thus, we arrive at the process algebra BPP_{θ}^{01} of the basic parallel processes. We extend this algebra with the priority operator θ , in order to give some actions priority over others.

Then, every finite weakly guarded BPP_{θ}^{01} specification yields the process of a parallel pushdown automaton, but not the other way around, there are processes of parallel pushdown automata that cannot be given by a finite weakly guarded BPP_{θ}^{01} specification. For parallel pushdown automata with just one state, such a specification can be found.

We obtain a complete correspondence by adding value passing communication.

The set of processes given by a parallel pushdown automaton coincides with the set of processes given by a finite weakly guarded recursive specification over a process algebra with actions, choice, priorities, and parallel composition with value passing communication.

We also provide another characterisation of parallel pushdown processes: a process is a parallel pushdown process if and only if there is a regular process such that the process is divergence-preserving branching bisimilar to the regular process communicating with an always accepting bag.

This paper contributes to our ongoing project to integrate automata theory and process theory. As a result, we can present the foundations of computer science using a computer model with interaction. Such a computer model relates more closely to the computers we see all around us.

As future work, we need to compare the algebra used here with Petri nets, see e.g. [16].

References

- [1] J. C. M. Baeten, T. Basten & M. A. Reniers (2009): *Process Algebra: Equational Theories of Communicating Processes*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, doi:10.1017/CBO9781139195003.
- [2] Jos C. M. Baeten & Jan A. Bergstra (1988): *Global Renaming Operators in Concrete Process Algebra*. *Inf. Comput.* 78(3), pp. 205–245, doi:10.1016/0890-5401(88)90027-2.
- [3] Jos C. M. Baeten, Jan A. Bergstra & Jan Willem Klop (1986): *Syntax and Defining Equations for an Interrupt Mechanism in Process Algebra*. *Fundamenta Informaticae* 9, pp. 127–168, doi:10.3233/FI-1986-9202.
- [4] Jos C. M. Baeten, Cesare Carissimo & Bas Luttik (2023): *Pushdown Automata and Context-Free Grammars in Bisimulation Semantics*. *Logical Methods in Computer Science* 19, pp. 15:1–15.32, doi:10.46298/LMCS-19(1:15)2023.
- [5] Jos C. M. Baeten, Flavio Corradini & Clemens Grabmayer (2007): *A characterization of regular expressions under bisimulation*. *J. ACM* 54(2), p. 6, doi:10.1145/1219092.1219094.
- [6] Jos C. M. Baeten, Pieter Cuijpers, Bas Luttik & Paul van Tilburg (2009): *A Process-Theoretic Look at Automata*. In Farhad Arbab & Marjan Sirjani, editors: *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers, Lecture Notes in Computer Science* 5961, Springer, pp. 1–33, doi:10.1007/978-3-642-11623-0_1.
- [7] Jos C. M. Baeten, Pieter J. L. Cuijpers & P. J. A. van Tilburg (2008): *A Context-Free Process as a Pushdown Automaton*. In Franck van Breugel & Marsha Chechik, editors: *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings, Lecture Notes in Computer Science* 5201, Springer, pp. 98–113, doi:10.1007/978-3-540-85361-9_11.
- [8] Jos C. M. Baeten, Pieter J. L. Cuijpers & Paul J. A. van Tilburg (2008): *A Basic Parallel Process as a Parallel Pushdown Automaton*. In Thomas T. Hildebrandt & Daniele Gorla, editors: *Proceedings of the 15th Workshop on Expressiveness in Concurrency, EXPRESS 2008, Toronto, ON, Canada, August 23, 2008, Electronic Notes in Theoretical Computer Science* 242, Elsevier, pp. 35–48, doi:10.1016/j.entcs.2009.06.012.
- [9] Jos C. M. Baeten, Bas Luttik, Tim Muller & Paul J. A. van Tilburg (2016): *Expressiveness modulo Bisimilarity of Regular Expressions with Parallel Composition*. *Mathematical Structures in Computer Science* 26, pp. 933–968, doi:10.1017/S0960129514000309.
- [10] Jos C. M. Baeten, Bas Luttik & Paul van Tilburg (2013): *Reactive Turing machines*. *Inf. Comput.* 231, pp. 143–166, doi:10.1016/j.ic.2013.08.010.
- [11] Jos C. M. Baeten & Chris Verhoef (1993): *A Congruence Theorem for Structured Operational Semantics with Predicates*. In Eike Best, editor: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings, Lecture Notes in Computer Science* 715, Springer, pp. 477–492, doi:10.1007/3-540-57208-2_33.
- [12] Twan Basten (1996): *Branching Bisimilarity is an Equivalence Indeed!* *Inf. Process. Lett.* 58(3), pp. 141–147, doi:10.1016/0020-0190(96)00034-8.
- [13] Jan A. Bergstra (1985): *Put and get, primitives for synchronous unreliable message passing*. *Logic group preprint series* 3, pp. 1–14.
- [14] Jan A. Bergstra & Jan Willem Klop (1984): *Process Algebra for Synchronous Communication*. *Information and Control* 60(1-3), pp. 109–137, doi:10.1016/S0019-9958(84)80025-X.
- [15] Roland N. Bol & Jan Friso Groote (1996): *The Meaning of Negative Premises in Transition System Specifications*. *J. ACM* 43(5), pp. 863–914, doi:10.1145/234752.234756.
- [16] Jürgen Dassow, Gairatzhan Mavlankulov, Mohamed Othman, Sherzod Turaev, Mohd Selamat & R Stiebe (2012): *Grammars Controlled by Petri Nets*. In Pawel Pawlewski, editor: *Petri Nets*, chapter 15, IntechOpen, Rijeka, pp. 337–358, doi:10.5772/50637.
- [17] Wan Fokkink, Rob van Glabbeek & Bas Luttik (2019): *Divide and congruence III: From decomposition of modal formulas to preservation of stability and divergence*. *Inf. Comput.* 268, doi:10.1016/j.ic.2019.104435.

- [18] Rob J. van Glabbeek (1993): *The Linear Time - Branching Time Spectrum II*. In Eike Best, editor: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings, Lecture Notes in Computer Science 715*, Springer, pp. 66–81, doi:10.1007/3-540-57208-2_6.
- [19] Rob van Glabbeek (2004): *The meaning of negative premises in transition system specifications II*. *J. Log. Algebr. Program.* 60-61, pp. 229–258, doi:10.1016/j.jlap.2004.03.007.
- [20] Rob van Glabbeek, Bas Luttik & Nikola Trcka (2009): *Branching Bisimilarity with Explicit Divergence*. *Fundamenta Informaticae* 93(4), pp. 371–392, doi:10.3233/FI-2009-109.
- [21] Rob van Glabbeek & Peter Weijland (1996): *Branching time and abstraction in bisimulation semantics*. *Journal of the ACM* 43(3), pp. 555–600, doi:10.1145/233551.233556.
- [22] Jan Friso Groote (1993): *Transition System Specifications with Negative Premises*. *Theor. Comput. Sci.* 118(2), pp. 263–299, doi:10.1016/0304-3975(93)90111-6.
- [23] Bas Luttik (2020): *Divergence-Preserving Branching Bisimilarity*. In Ornela Dardha & Jurriaan Rot, editors: *Proceedings Combined 27th International Workshop on Expressiveness in Concurrency and 17th Workshop on Structural Operational Semantics, EXPRESS/SOS 2020, and 17th Workshop on Structural Operational Semantics, Online, 31 August 2020, EPTCS 322*, pp. 3–11, doi:10.4204/EPTCS.322.2.
- [24] Faron Moller (1996): *Infinite Results*. In Ugo Montanari & Vladimiro Sassone, editors: *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings, Lecture Notes in Computer Science 1119*, Springer, pp. 195–216, doi:10.1007/3-540-61604-7_56.
- [25] Paul J. A. van Tilburg (2011): *From computability to executability : a process-theoretic view on automata theory*. Ph.D. thesis, Mathematics and Computer Science, doi:10.6100/IR716374.
- [26] Chris Verhoef (1995): *A Congruence Theorem for Structured Operational Semantics with Predicates and Negative Premises*. *Nord. J. Comput.* 2(2), pp. 274–302.