A Context-Free Process as a Pushdown Automaton

J.C.M. Baeten, P.J.L. Cuijpers, and P.J.A. van Tilburg

Division of Computer Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands,
{j.c.m.baeten,p.j.l.cuijpers,p.j.a.v.tilburg}@tue.nl

Abstract. A well-known theorem in automata theory states that every context-free language is accepted by a pushdown automaton. We investigate this theorem in the setting of processes, using the rooted branching bisimulation and contrasimulation equivalences instead of language equivalence. In process theory, different from automata theory, interaction is explicit, so we realize a pushdown automaton as a regular process communicating with a stack.

1 Introduction

Automata and formal language theory have a place in every undergraduate computer science curriculum, as this provides students with a simple model of computation, and an understanding of computability. This simple model of computation does not include the notion of interaction, which is more and more important at a time when computers are always connected.

Adding interaction to automata theory leads to concurrency theory. The two models of computation are strongly related, and have much in common. Still, research into both models has progressed more or less independently. We are embarked on a program that studies similarities and differences between the two models, and that shows how concepts, notations, methods and techniques developed in one of the fields can be beneficial in the other field.

This paper studies, in a concurrency theoretic setting, the relation between the notion of a context-free process [4,18,9], and that of a pushdown automaton (i.e. a regular process that interacts with a stack) [17]. In order to obtain a full correspondence with automata theory, we extend the definition of context-free processes of [9] with deadlock ($\mathbf{0}$, as in [18]) and termination ($\mathbf{1}$, studied here for the first time). The goal of this paper, is to show how every context-free process can be translated into a pushdown automaton. The main difference with the work of [17], is that we do this while explicitly modeling the interaction between the regular process and the stack in this automaton. As it turns out, the addition of termination leads to additional expressivity of context-free processes, which in turn leads to a case distinction in the translation. Finally, the results in [17], show that the translation in the other direction is not always possible for context-free processes without termination, but as $\mathbf{1}$ gives us additional expressivity, it might

hold in the new setting. However, as the translation in one direction is already not trivial, we leave the other direction as future work.

This paper is structured as follows. We first introduce our definitions of regular and context-free processes, and the associated equational theory, in Sects. 2 and 3, respectively. Then, in Sect. 4, we give the general structure of our translation, and study the different cases mentioned before as instances of this structure. We conclude the paper in Sect. 5, and give recommendations for future work.

2 Regular Processes

Before we introduce context-free processes, we first consider the notion of a regular process and its relation to regular languages in automata theory. We start with a definition of the notion of transition system from process theory. A finite transition system can be thought of as a non-deterministic finite automaton. In order to have a complete analogy, the transition systems we study have a subset of states marked as final states.

Definition 1 (Transition system). *A* transition system *M is a quintuple $(\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ where:*

1. *$\mathcal{S}$ is a set of states,*
2. *$\mathcal{A}$ is an alphabet,*
3. *$\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the set of* transitions *or* steps*,*
4. *$\uparrow \in \mathcal{S}$ is the initial state,*
5. *$\downarrow \subseteq \mathcal{S}$ is a set of final states.*

For $(s, a, t) \in \rightarrow$ we write $s \xrightarrow{a} t$. For $s \in \downarrow$ we write $s\downarrow$. A finite transition system *or* non-deterministic finite automaton *is a transition system of which the sets $\mathcal{S}$ and $\mathcal{A}$ are finite.*

In accordance with automata theory, where a *regular language* is a language equivalence class of a non-deterministic finite automaton, we define a *regular process* to be a bisimulation equivalence class [13] of a finite transition system. Contrary to automata theory, it is well-known that not every regular process has a *deterministic* finite transition system (i.e. a transition system for which the relation $\rightarrow$ is functional). The set of deterministic regular processes is a proper subset of the set of regular processes.

Next, consider the automata theoretic characterization of a regular language by means of a right-linear grammar. In process theory, a grammar is called a *recursive specification*: it is a set of recursive equations over a set of variables. A right-linear grammar then coincides with a recursive specification over a finite set of variables in the Minimal Algebra MA. (We use standard process algebra notation as propagated by [2,5].)

Definition 2. *The signature of* Minimal Algebra *MA is as follows:*

1. *There is a constant* $\mathbf{0}$*; this denotes inaction, a deadlock state; other names are* δ *or stop.*
2. *There is a constant* $\mathbf{1}$*; this denotes termination, a final state; other names are* ε*, skip or the empty process.*
3. *For each element of the alphabet* $\mathcal{A}$ *there is a unary operator* $a._-$ *called* action prefix*; a term* $a.x$ *will execute the elementary action* a *and then proceed as* x*.*
4. *There is a binary operator* $+$ *called* alternative composition*; a term* $x+y$ *will either execute* x *or execute* y*, a choice will be made between the alternatives.*

The constants $\mathbf{0}$ and $\mathbf{1}$ are needed to denote transition systems with a single state and no transitions. The constant $\mathbf{0}$ denotes a single state that is not a final state, while $\mathbf{1}$ denotes a single state that is also a final state.

Definition 3. *Let* $\mathcal{V}$ *be a set of* variables*. A* recursive specification *over* $\mathcal{V}$ *with initial variable* $S \in \mathcal{V}$ *is a set of equations of the form* $X = t_X$*, exactly one for each* $X \in \mathcal{V}$*, where each right-hand side* t_X *is a term over some signature, possibly containing elements of* $\mathcal{V}$*. A recursive specification is called finite, if* $\mathcal{V}$ *is finite.*

We find that a finite recursive specification over MA can be seen as a right-linear grammar. Now each finite transition system corresponds directly to a finite recursive specification over MA, using a variable for every state. To go from a term over MA to a transition system, we use *structural operational semantics* [1], with rules given in Table 2.1.

$$\frac{}{\mathbf{1}\downarrow} \qquad \frac{}{a.x \xrightarrow{a} x}$$

$$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \qquad \frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'} \qquad \frac{x\downarrow}{x + y\downarrow} \qquad \frac{y\downarrow}{x + y\downarrow}$$

$$\frac{t_X \xrightarrow{a} x \quad X = t_X}{X \xrightarrow{a} x} \qquad \frac{t_X\downarrow \quad X = t_X}{X\downarrow}$$

Table 2.1: Operational rules for MA and recursion ($a \in \mathcal{A}, X \in \mathcal{V}$).

3 Context-Free Processes

Considering the automata theoretic notion of a context-free grammar, we find a correspondence in process theory by taking a recursive specification over a finite set of variables, and over the *Sequential Algebra* SA, which is MA extended with

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x\!\downarrow \quad y \xrightarrow{a} y'}{x \cdot y \xrightarrow{a} y'} \qquad \frac{x\!\downarrow \quad y\!\downarrow}{x \cdot y\!\downarrow}$$

Table 3.1: Operational rules for sequential composition ($a \in \mathcal{A}$).

sequential composition $\_ \cdot \_$. We extend the operational rules of Table 2.1 with rules for sequential composition, in Table 3.1.

Now consider the following specification

$$S = \mathbf{1} + S \cdot a.\mathbf{1}.$$

Our first observation is that, by means of the operational rules, we derive an infinite transition system, which moreover is infinitely branching. All the states of this transition system are different in bisimulation semantics, and so this is in fact an infinitely branching process. Our second observation is that this recursive specification has infinitely many different (non-bisimilar) solutions in the transition system model, since adding any non-terminating branch to the initial node will also give a solution. This is because the equation is *unguarded*, the right-hand side contains a variable that is not in the scope of an action-prefix operator, and also cannot be brought into such a form. So, if there are multiple solutions to a recursive specification, we have multiple processes that correspond to this specification. This is an undesired property.

These two observations are the reason to restrict to guarded recursive specifications only. It is well-known that a guarded recursive specification has a unique solution in the transition system model (see [7,6]). This restriction leads to the following definition.

Definition 4. *A* context-free process *is the bisimulation equivalence class of the transition system generated by a finite guarded recursive specification over Sequential Algebra SA.*

In this paper, we use equational reasoning to manipulate recursive specifications. The equational theory of SA is given in Table 3.2. Note that the axioms $x \cdot (y + z) = x \cdot y + x \cdot z$ and $x \cdot \mathbf{0} = \mathbf{0}$ do not hold in bisimulation semantics (in contrast to language equivalence). The given theory constitutes a sound and ground-complete axiomatization of the model of transition systems modulo bisimulation (see [6,5]). Furthermore, we often use the aforementioned principle, that guarded recursive specifications have unique solutions [6].

Using the axioms, any guarded recursive specification can be brought into *Greibach normal form* [4]:

$$X = \sum_{i \in I_X} a_i.\xi_i \ (+ \mathbf{1}).$$

In this form, every right-hand side of every equation consists of a number of summands, indexed by a finite set I_X (the empty sum is $\mathbf{0}$), each of which is

$x+y$	$=y+x$	$x+\mathbf{0}$	$=x$
$(x+y)+z$	$=x+(y+z)$	$\mathbf{0}\cdot x$	$=\mathbf{0}$
$x+x$	$=x$	$\mathbf{1}\cdot x$	$=x$
$(x+y)\cdot z$	$=x\cdot z+y\cdot z$	$x\cdot\mathbf{1}$	$=x$
$(x\cdot y)\cdot z$	$=x\cdot(y\cdot z)$	$(a.x)\cdot y$	$=a.(x\cdot y)$

Table 3.2: Equational theory of SA ($a \in \mathcal{A}$).

$\mathbf{1}$, or of the form $a_i.\xi_i$, where ξ_i is the sequential composition of a number of variables (the empty sequence is $\mathbf{1}$). We define $\mathcal{I}$ as the multiset resulting of the union of all index sets. For a recursive specification in Greibach normal form, every state of the transition system is given by a sequence of variables. Note that we can take the index sets associated with the variables to be disjoint, so that we can define a function $V : \mathcal{I} \to \mathcal{V}$ that gives, for any index that occurs somewhere in the specification, the variable of the equation in which it occurs.

As an example, we consider the important context-free process *stack*. Suppose D is a finite data set, then we define the following actions in $\mathcal{A}$, for each $d \in D$:

- $?d$: push d onto the stack;
- $!d$: pop d from the stack.

Now the recursive specification is as follows:

$$S = \mathbf{1} + \sum_{d\in D} ?d.S \cdot !d.S.$$

In order to see that the above process indeed defines a stack, define processes S_σ, denoting the stack with contents $\sigma \in D^*$, as follows: the first equation for the empty stack, the second for any nonempty stack, with top d and tail σ:

$$S_\varepsilon = S, \qquad\qquad S_{d\sigma} = S \cdot !d.S_\sigma.$$

Then it is straightforward to derive the following equations:

$$S_\varepsilon = \mathbf{1} + \sum_{d\in D} ?d.S_d, \qquad\qquad S_{d\sigma} = !d.S_\sigma + \sum_{e\in D} ?e.S_{ed\sigma}.$$

We obtain the following specification for the stack in Greibach normal form:

$$S = \mathbf{1} + \sum_{d\in D} ?d.T_d \cdot S, \qquad\qquad T_d = !d.\mathbf{1} + \sum_{e\in D} ?e.T_e \cdot T_d.$$

Finally, we define the *forgetful stack*, which can forget a datum it has received when popped, as follows:

$$S = \mathbf{1} + \sum_{d\in D} ?d.S \cdot (\mathbf{1} + !d.S).$$

Due to the presence of $\mathbf{1}$, a context-free process may have unbounded branching [8] that we need to mimic with our pushdown automaton. One possible

solution is to use forgetfulness of the stack to get this unbounded branching in our pushdown automaton, as we will show in the next section. Note that when using a more restrictive notion of context-free processes we have bounded branching, and thus we don't need the forgetfulness property.

The above presented specifications are still meaningful when D is an infinite data set (see e.g. [15,14]), but does not represent a term in SA anymore. In this paper, we use infinite summation in some intermediate results, but the end results are finite. Note that the infinite sums also preserve the notion of congruence we are working with.

Now, consider the notion of a pushdown automaton. A pushdown automaton is just a finite automaton, but at every step it can push a number of elements onto a stack, or it can pop the top of the stack, and take this information into account in determining the next move. Thus, making the interaction explicit, a pushdown automaton is a regular process communicating with a stack.

In order to model the interaction between the regular process and the stack, we briefly introduce communication by synchronization. We introduce the *Communication Algebra* CA, which extends MA and SA with the *parallel composition* operator $\|$. Parallel processes can execute actions independently (called interleaving), or can synchronize by executing matching actions. In this paper, it is sufficient to use a particular communication function, that will only synchronize actions $!d$ and $?d$ (for the same $d \in D$). The result of such a synchronization is denoted $\mathbin{?}d$. CA also contains the *encapsulation operator* $\partial_*(\_)$, which blocks actions $!d$ and $?d$, and the *abstraction operator* $\tau_*(\_)$ which turns all $\mathbin{?}d$ actions into the internal action τ. We show the operational rules in Table 3.3.

$$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y} \qquad \frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'} \qquad \frac{x\!\downarrow \quad y\!\downarrow}{x \parallel y\!\downarrow}$$

$$\frac{x \xrightarrow{?d} x' \quad y \xrightarrow{!d} y'}{x \parallel y \xrightarrow{\mathbin{?}d} x' \parallel y'} \qquad \frac{x \xrightarrow{!d} x' \quad y \xrightarrow{?d} y'}{x \parallel y \xrightarrow{\mathbin{?}d} x' \parallel y'}$$

$$\frac{x \xrightarrow{a} x' \quad a \neq !d, ?d}{\partial_*(x) \xrightarrow{a} \partial_*(x')} \qquad \frac{x\!\downarrow}{\partial_*(x)\!\downarrow}$$

$$\frac{x \xrightarrow{\mathbin{?}d} x'}{\tau_*(x) \xrightarrow{\tau} \tau_*(x')} \qquad \frac{x \xrightarrow{a} x' \quad a \neq \mathbin{?}d}{\tau_*(x) \xrightarrow{a} \tau_*(x')} \qquad \frac{x\!\downarrow}{\tau_*(x)\!\downarrow}$$

Table 3.3: Operational rules for CA ($a \in \mathcal{A}$).

Our finite axiomatization of transition systems of CA modulo rooted branching bisimulation uses the auxiliary operators $\_ \between \_$ and $\_ \mid \_$ [7,16]. See Table 3.4 for the axioms and [5] for an explanation of these axioms.

$x \parallel y$	$= x \between y + y \between x + x \mid y$	$a.(\tau.(x+y)+x)$	$= a.(x+y)$
$\mathbf{0} \between x$	$= \mathbf{0}$	$x \mid y$	$= y \mid x$
$\mathbf{1} \between x$	$= \mathbf{0}$	$x \parallel \mathbf{1}$	$= x$
$a.x \between y$	$= a.(x \parallel y)$	$\mathbf{1} \mid x + \mathbf{1}$	$= \mathbf{1}$
$(x+y) \between z$	$= x \between z + y \between z$	$(x \parallel y) \parallel z$	$= x \parallel (y \parallel z)$
$\mathbf{0} \mid x$	$= \mathbf{0}$	$(x \mid y) \mid z$	$= x \mid (y \mid z)$
$(x+y) \mid z$	$= x \mid z + y \mid z$	$(x \between y) \between z$	$= x \between (y \parallel z)$
$\mathbf{1} \mid \mathbf{1}$	$= \mathbf{1}$	$(x \mid y) \between z$	$= x \mid (y \between z)$
$a.x \mid \mathbf{1}$	$= \mathbf{0}$	$x \between \tau.y$	$= x \between y$
$!d.x \mid ?d.y$	$= ⁉d.(x \parallel y)$	$x \mid \tau.y$	$= \mathbf{0}$
$a.x \mid b.y$	$= \mathbf{0}$ *if* $\{a,b\} \neq \{!d, ?d\}$		
$\partial_*(\mathbf{0})$	$= \mathbf{0}$	$\tau_*(\mathbf{0})$	$= \mathbf{0}$
$\partial_*(\mathbf{1})$	$= \mathbf{1}$	$\tau_*(\mathbf{1})$	$= \mathbf{1}$
$\partial_*(!d.x)$	$= \partial_*(?d.x) = \mathbf{0}$	$\tau_*(⁉d.x)$	$= \tau.\tau_*(x)$
$\partial_*(a.x)$	$= a.\partial_*(x)$ *if* $a \notin \{!d, ?d\}$	$\tau_*(a.x)$	$= a.\tau_*(x)$ *if* $a \neq ⁉d$
$\partial_*(x+y)$	$= \partial_*(x) + \partial_*(y)$	$\tau_*(x+y)$	$= \tau_*(x) + \tau_*(y)$

Table 3.4: Equational theory of CA ($a \in \mathcal{A} \cup \{\tau\}$).

The given equational theory is sound and ground-complete for the model of transition systems modulo rooted branching bisimulation [13]. This is the preferred model we use, but all our reasoning in the following takes place in the equational theory, so is model-independent provided the models preserve validity of the axioms and unique solutions for guarded recursive specifications.

4 Pushdown Automata

The main goal of this paper, is to prove that every context-free process is equal to a regular process communicating with a stack. Thus, if P is any context-free process, then we want to find a regular process Q such that

$$P = \tau_*(\partial_*(Q \parallel S_\sigma)),$$

where S_σ is a state of a stack process. Without loss of generality, we assume in this section that P is given in Greibach normal form.

The first, intermediate, solution we present uses a potentially infinite data type D. If D is infinite, then the stack is not a context-free process. Also, we define Q in the syntax of Minimal Algebra, but it may have infinitely many different variables, so it may not be a regular process. Later, we specialize to cases where the data type is finite, and these problems do not occur. We do this by reducing the main solution using several assumptions, that categorize the possibilities for P into three classes: *opaque*, *bounded branching*, and *unrestricted* specifications.

4.1 Intermediate Solution

The infinite data type D we use for our intermediate solution, consists of pairs. The first element of the pair is a variable of P. The second element is a multiset over $\mathcal{I}$, i.e. a multiset over V, plus an indication of a termination option. So, $D = \mathcal{V} \times (\mathcal{I} \cup \{\mathbf{1}\} \to \mathbb{N})$.

For multisets A, B, we write $A(a) = n$ if the element a occurs n times in A, and we write $A \uplus B$ to denote union of multisets such that $(A \uplus B)(a) = A(a) + B(a)$. We use the subscript c in a process term $(p)_c$ to denote that p only occurs in the term if condition c holds. Finally, we call a variable *transparent* if its equation has an $\mathbf{1}$-summand. We denote the set of transparent variables of P with $\mathcal{V}^{+\mathbf{1}}$.

Now, we prove the main theorem by first stating the specification of our solution and introducing some formalisms, before giving the main proof. The proof will provide insight in how and why our solution works.

Theorem 1. *For every context-free process P there exists a process Q given by a recursive specification over MA such that $P = \tau_*(\partial_*(Q \parallel S_\sigma))$ for some state S_σ of the (partially) forgetful stack.*

Proof. Let E be a finite recursive specification of P in Greibach normal form. Now, let F be a recursive specification that contains the following equations for every variable $X \in \mathcal{V}$ of the specification E, $i \in I_X$ and multiset A over $\mathcal{I}$:

$$\hat{X}(i, A) = \mathrm{Push}(\xi_i, A),$$

with $\mathrm{Push}(\xi, A)$ recursively defined as

$$\mathrm{Push}(\mathbf{1}, A) = \mathrm{Ctrl}(A),$$

$$\mathrm{Push}(\xi' Y, A) = \begin{cases} !\langle Y, A\rangle.\mathrm{Push}(\xi', I_Y) & \text{if } Y \notin \mathcal{V}^{+\mathbf{1}}, \\ !\langle Y, A\rangle.\mathrm{Push}(\xi', I_Y \uplus A) & \text{if } Y \in \mathcal{V}^{+\mathbf{1}}. \end{cases}$$

where Y is a variable at the end of the original sequence and ξ' is the sequence that is left over when Y has been removed. So, $\mathrm{Push}(\xi, A)$ is defined backwards with respect to sequence ξ, necessary to preserve the correct structure on the stack while pushing.

In addition, let F also contain the following equations of a *partially forgetful stack* and a (regular) finite control.

$$S = \mathbf{1} + \sum_{\substack{\langle V, A\rangle \in D \\ V \notin \mathcal{V}^{+\mathbf{1}}}} ?\langle V, A\rangle.S \cdot !\langle V, A\rangle.S + \sum_{\substack{\langle V, A\rangle \in D \\ V \in \mathcal{V}^{+\mathbf{1}}}} ?\langle V, A\rangle.S \cdot (\mathbf{1} + !\langle V, A\rangle.S),$$

$$\mathrm{Ctrl}(A) = \sum_{i \in \mathcal{I}} \sum_{0 < l \le A(i)} a_i.\mathrm{Pop}(i, l)\ (+\,\mathbf{1})_{A(\mathbf{1}) \ge 1},$$

$$\mathrm{Pop}(i, l) = \begin{cases} \displaystyle\sum_{\substack{\langle V, A\rangle \in D \\ i \in I_V}} ?\langle V, A\rangle.\hat{V}(i, A) & \text{if } V(i) \notin \mathcal{V}^{+\mathbf{1}}, \\ \displaystyle\sum_{\substack{\langle V, A\rangle \in D \\ i \in I_V \wedge A(i) = l-1}} ?\langle V, A\rangle.\hat{V}(i, A) & \text{if } V(i) \in \mathcal{V}^{+\mathbf{1}}, \end{cases}$$

The process $\mathrm{Ctrl}(A)$ allows for a choice to be made among the possible enabled actions a_i, referred to by the indices in the multiset A. It can also terminate if the termination option $\mathbf{1}$ is present in A. Once an action has been chosen, Ctrl calls Pop with the index i of the action that was executed and the occurrence l of the variable belonging to that index, $V(i)$, on the stack that needs to be popped. Once that variable, say $V(i) = X$, has been popped, $\hat{X}(i, A)$ is executed to mimic the rest of the behavior when a_i has been executed, namely pushing ξ_i on the stack. Note that this means that A, the multiset of possible actions, always has to correspond with the contents of the partially forgetful stack.

Before we show how the above specification mimics the specification of P, we first study the structure of P itself more closely. In Greibach normal form, every state in P is labeled with a sequential composition of variables $X\xi$ (or in the trivial case, $\mathbf{1}$). Substituting the Greibach normal form of the leading variable X gives us the following:

$$X\xi = \Big(\sum_{i \in I_X} a_i.\xi_i \ (+\mathbf{1}) \Big) \cdot \xi = \sum_{i \in I_X} a_i.\xi_i \cdot \xi \ (+\xi).$$

Introducing a fresh variable $\overline{P}(\xi)$ for each possible sequence ξ, we obtain the following equivalent infinite recursive specification.

$$\overline{P}(\mathbf{1}) = \mathbf{1}, \qquad \overline{P}(X\xi) = \sum_{i \in I_X} a_i.\overline{P}(\xi_i\xi) \ (+\overline{P}(\xi)).$$

Note that this specification is still guarded, as the unfolding of the unguarded recursion will always terminate.

In order to link the sequences that make up the states of P to the contents of the stack in our specification F, we use two functions h and e. The function h determines, for a given sequence $X\xi$, the multiset that contains for each index $i \in \mathcal{I}$ the number of occurrences of the process variable $V(i)$ in a sequence that is reachable through termination of preceding variables. It also determines whether a termination is possible through the entire sequence.

$$h(\mathbf{1}) = \{\mathbf{1}\},$$
$$h(X\xi) = \begin{cases} I_X & \text{if } X \notin \mathcal{V}^{+\mathbf{1}}, \\ I_X \uplus h(\xi) & \text{if } X \in \mathcal{V}^{+\mathbf{1}}. \end{cases}$$

The function e, defined by $e(\mathbf{1}) = \mathbf{1}$ and $e(X\xi) = \langle X, h(\xi) \rangle e(\xi)$, then represents the actual contents of the stack.

Lemma 1. *Let $i \in \mathcal{I}$. Then $h(X\xi)(i) = h(\xi)(i)$ iff $i \notin I_X$.*

Having characterized the relationship between states of P and the partially forgetful stack of F, we define $Q = \mathrm{Ctrl}(h(X))$, where X is the initial variable of E, and we continue to prove $P = \tau_*(\partial_*(Q \parallel S_{e(X)})) = \big[Q \parallel S_{e(X)}\big]_*$.[1] More precisely, we will prove for any sequence of variables ξ, that

$$\overline{P}(\xi) = \big[\mathrm{Ctrl}(h(\xi)) \parallel S_{e(\xi)}\big]_* .$$

[1] From here on, $[p]_*$ is used as a shorthand notation for $\tau_*(\partial_*(p))$.

1. If $\xi = \mathbf{1}$, then $\overline{P}(\mathbf{1}) = \left[\mathrm{Ctrl}(h(\mathbf{1})) \parallel S_{e(\mathbf{1})}\right]_* = [\mathbf{1} \parallel S_{\mathbf{1}}]_* = \mathbf{1}$
2. If $\xi = X\xi'$, then there are two cases.
   (a) Assume that $X \notin \mathcal{V}^{+\mathbf{1}}$. First, apply the definition of $h(X\xi')$ and then the definition of $\mathrm{Ctrl}(I_X)$.

$$\begin{aligned}\overline{P}(X\xi') &\stackrel{?}{=} \left[\mathrm{Ctrl}(h(X\xi')) \parallel S_{e(X\xi')}\right]_* \\ &= \left[\mathrm{Ctrl}(I_X) \parallel S_{e(X\xi')}\right]_* \\ &= \sum_{i\in\mathcal{I}} \sum_{0<l\leq I_X(i)} a_i.\left[\mathrm{Pop}(i,l) \parallel S_{e(X\xi')}\right]_* \ (+\ \left[\mathbf{1} \parallel S_{e(X\xi')}\right]_*)_{I_X(\mathbf{1})\geq 1}\end{aligned}$$

   Note that I_X is a set, so it follows that $I_X(i) = 1$ for $i \in I_X$ and $I_X(i) = 0$ for all $i \in \mathcal{I} - I_X$. Therefore, the first two summations can be written as $\sum_{i\in I_X}$ when we instantiate $l = 1$. Because it also follows that $I_X(\mathbf{1}) = 0$, we remove the conditional summand $\left[\mathbf{1} \parallel S_{e(X\xi')}\right]_*$.

$$= \sum_{i\in I_X} a_i.\left[\mathrm{Pop}(i,1) \parallel S_{e(X\xi')}\right]_*$$

   Unfold the definition of $S_{e(X\xi')}$ once, then perform the pop by applying the definitions of $\mathrm{Pop}(i,1)$ and $\hat{X}(i,h(\xi'))$.

$$\begin{aligned}&= \sum_{i\in I_X} a_i.\tau.\left[\hat{X}(i,h(\xi')) \parallel S_{e(\xi')}\right]_* \\ &= \sum_{i\in I_X} a_i.\tau.\left[\mathrm{Push}(\xi_i,h(\xi')) \parallel S_{e(\xi')}\right]_*\end{aligned}$$

   Finally, perform $|\xi_i|$ pushes by repeatedly applying the definitions of $\mathrm{Push}(\xi,A)$ and $S_{e(\xi)}$.

$$\begin{aligned}&= \sum_{i\in I_X} a_i.\tau^{|\xi_i|+1}.\left[\mathrm{Ctrl}(h(\xi_i\xi')) \parallel S_{e(\xi_i\xi')}\right]_* \\ &= \sum_{i\in I_X} a_i.\left[\mathrm{Ctrl}(h(\xi_i\xi')) \parallel S_{e(\xi_i\xi')}\right]_* \\ &= \sum_{i\in I_X} a_i.\overline{P}(\xi_i\xi').\end{aligned}$$

   (b) Assume that $X \in \mathcal{V}^{+\mathbf{1}}$. First, substitute the definition of $\mathrm{Ctrl}(h(X\xi'))$.

$$\begin{aligned}\overline{P}(X\xi') &\stackrel{?}{=} \left[\mathrm{Ctrl}(h(X\xi')) \parallel S_{e(X\xi')}\right]_* \\ &= \sum_{i\in\mathcal{I}} \sum_{0<l\leq h(X\xi')(i)} a_i.\left[\mathrm{Pop}(i,l) \parallel S_{e(X\xi')}\right]_*\end{aligned}$$

Split off the case that will pop the top element of the stack, namely when $i \in I_X$ and $l = h(X\xi')(i)$. By the same argument as in the previous case, we can write the first two summations as $\sum_{i \in I_X}$.

$$
\begin{aligned}
&= \sum_{i \in I_X} a_i . \left[\mathrm{Pop}(i, h(X\xi')(i)) \parallel S_{e(X\xi')} \right]_* \\
&\quad + \sum_{i \in \mathcal{I}} \sum_{\substack{0 < l \leq h(X\xi')(i) \\ i \notin I_X \vee l \neq h(X\xi')(i)}} a_i . \left[\mathrm{Pop}(i, l) \parallel S_{e(X\xi')} \right]_* \\
&\quad (+ \left[\mathbf{1} \parallel S_{e(X\xi')} \right]_*)_{h(X\xi')(\mathbf{1}) \geq 1}
\end{aligned}
$$

Consider the first summation. If $i \in I_X$ and $l = h(X\xi')(i)$, then $h(\xi')(i) = l-1$ by Lemma 1 and therefore by the definitions of $\mathrm{Pop}(i, l)$ and $S_{e(X\xi')}$:

$$
\begin{aligned}
&= \sum_{i \in I_X} a_i . \left[\sum_{\substack{\langle V, A' \rangle \in D \\ i \in I_v \wedge A'(i) = l-1}} ?\langle V, A' \rangle . \hat{X}(i, A') \parallel S_{\langle X, h(\xi')\rangle e(\xi')} \right]_* \\
&\quad + \sum_{i \in \mathcal{I}} \sum_{\substack{0 < l \leq h(X\xi')(i) \\ i \notin I_X \vee l \neq h(X\xi')(i)}} a_i . \left[\mathrm{Pop}(i, l) \parallel S_{e(X\xi')} \right]_* \\
&\quad (+ \left[\mathbf{1} \parallel S_{e(X\xi')} \right]_*)_{h(X\xi')(\mathbf{1}) \geq 1}
\end{aligned}
$$

The stack may contain a series of transparent variables with multisets in which the occurrence of index i is strictly smaller than at the top. So, only the top element can be popped.

$$
\begin{aligned}
&= \sum_{i \in I_X} a_i . \tau . \left[\hat{X}(h(\xi')) \parallel S_{e(\xi')} \right]_* \\
&\quad + \sum_{i \in \mathcal{I}} \sum_{\substack{0 < l \leq h(X\xi')(i) \\ i \notin I_X \vee l \neq h(X\xi')(i)}} a_i . \left[\mathrm{Pop}(i, l) \parallel S_{e(X\xi')} \right]_* \\
&\quad (+ \left[\mathbf{1} \parallel S_{e(X\xi')} \right]_*)_{h(X\xi')(\mathbf{1}) \geq 1}
\end{aligned}
$$

Now, consider the second summation and optional summand. Given that $0 < l \leq h(X\xi')(i)$, it follows from the combination of Lemma 1 (in case $i \notin I_X$) or $l \neq h(X\xi)(i)$ (in case $i \in I_X$), that $0 < l \leq h(\xi')(i)$. Because we have forgetfulness of the stack $S_{e(X\xi')}$, it holds that $\left[\mathrm{Pop}(i, l) \parallel S_{e(X\xi')} \right]_* = \left[\mathrm{Pop}(i, l) \parallel S_{e(\xi')} \right]_*$ and that if $h(X\xi')(\mathbf{1}) \geq 1$, then $h(\xi')(\mathbf{1}) \geq 1$.

$$
\begin{aligned}
&= \sum_{i \in I_X} a_i . \tau . \left[\hat{X}(h(\xi')) \parallel S_{e(\xi')} \right]_* \\
&\quad + \sum_{i \in \mathcal{I}} \sum_{0 < l \leq h(\xi')(i)} a_i . \left[\mathrm{Pop}(i, l) \parallel S_{e(\xi')} \right]_* \\
&\quad (+ \left[\mathbf{1} \parallel S_{e(\xi')} \right]_*)_{h(\xi')(\mathbf{1}) \geq 1}
\end{aligned}
$$

Apply the definition of $\hat{X}(i,h(\xi'))$ on the first summation. Substitute the second summation and the optional summand with the definition of $\mathrm{Ctrl}(\xi')$.

$$= \sum_{i\in I_X} a_i.\tau.\left[\mathrm{Push}(\xi_i,h(\xi'))\parallel S_{e(\xi')}\right]_* + \left[\mathrm{Ctrl}(\xi')\parallel S_{e(\xi')}\right]_*$$

Perform $|\xi_i|$ pushes by repeatedly applying the definitions of $\mathrm{Push}(\xi,A)$ and $S_{e(\xi)}$.

$$\begin{aligned}
&= \sum_{i\in I_X} a_i.\tau^{|\xi_i|+1}.\left[\mathrm{Ctrl}(h(\xi_i\xi'))\parallel S_{e(\xi_i\xi')}\right]_* + \left[\mathrm{Ctrl}(\xi')\parallel S_{e(\xi')}\right]_*\\
&= \sum_{i\in I_X} a_i.\left[\mathrm{Ctrl}(h(\xi_i\xi'))\parallel S_{e(\xi_i\xi')}\right]_* + \left[\mathrm{Ctrl}(\xi')\parallel S_{e(\xi')}\right]_*\\
&= \sum_{i\in I_X} a_i.\overline{P}(\xi_i\xi') + \overline{P}(\xi').
\end{aligned}$$

This concludes our proof that there exists a, possibly infinite, recursive specification over MA that, in parallel with a partially forgetful stack, is equivalent to a context-free process P. □

In the following subsections, we will study under which conditions this specification reduces to a finite recursive specification over MA.

4.2 Opacity

In [18], context-free processes with $\mathbf{0}$ but without $\mathbf{1}$ were presented. Related to the absence of $\mathbf{1}$, we find that the intermediate solution reduces to a finite recursive specification, if none of the variables are transparent ($\mathcal{V}^{+\mathbf{1}} = \emptyset$), i.e. the specification is *opaque*.

From the specification of $\mathrm{Push}(\xi,A)$ we observe that now only sets are pushed on the stack (i.e. multisets in which each element occurs at most once). Hence, we can use a data set $D' = \mathcal{V}\times\mathcal{P}(\mathcal{I}\cup\{\mathbf{1}\})$ that no longer is infinite. We obtain a new, finite recursive specification, by replacing the equations for S, $\mathrm{Ctrl}(A)$, $\mathrm{Pop}(i,l)$ and $\mathrm{Push}(\xi,A)$ by the following ones:

$$\begin{aligned}
S &= \mathbf{1} + \sum_{\langle V,A\rangle\in D'} ?\langle V,A\rangle.S\cdot !\langle V,A\rangle.S,\\
\mathrm{Ctrl}(A) &= \sum_{i\in A} a_i.\mathrm{Pop}(i)\ (+\,\mathbf{1})_{\mathbf{1}\in A},\\
\mathrm{Pop}(i) &= \sum_{\substack{\langle V,A\rangle\in D'\\ i\in I_V}} ?\langle V,A\rangle.\hat{V}(i,A),\\
\mathrm{Push}(\mathbf{1},A) &= \mathrm{Ctrl}(A),\\
\mathrm{Push}(\xi Y,A) &= !\langle Y,A\rangle.\mathrm{Push}(\xi,I_Y).
\end{aligned}$$

Corollary 1. *For any context-free process P with recursive specification E that is opaque, there exists a regular process Q such that $P = \left[Q\parallel S_{e(X)}\right]_*$.*

4.3 Bounded Branching

Consider the following example, in which the variable Y is transparent.

$$X = a.X \cdot Y + b.\mathbf{1}, \qquad\qquad Y = \mathbf{1} + c.\mathbf{1}.$$

By executing a n times followed by b, the system gets to state Y^n. Here we have unbounded branching, since $Y^n \xrightarrow{c} Y^k$ for every $k < n$. This means state Y^n has n different outgoing c-steps, since none of the states Y^k are bisimilar. Thus, we cannot put a bound on the number of summands in the entire specification. The observation that the presence of $\mathbf{1}$-summands can cause unbounded branching is due to [8].

In case we have unbounded branching, it can be shown that there is no finite solution modulo rooted branching bisimulation. The reason for this, is that a regular process is certainly boundedly branching, so that the introduction of unbounded branching must take place through communication with the stack (in any solution, not only ours). This will result in internal τ transitions to states that are not rooted branching bisimilar, which makes that the τ transitions cannot be eliminated.

Assume now, that we have a specification for P that results in boundedly branching behavior, then the intermediate solution (see Sect. 4.1) does reduce to a finite recursive specification. In that case, the number of variables in a sequence ξ that can perform a certain action is bounded by some natural number N. The stack itself is an example of such a process. Hence, $h(\xi)(i) \leq N$ for any $i \in \mathcal{I}$, so the multisets in the data type D will never contain more than N occurrences for each index. We can reduce our specification by replacing $\mathrm{Ctrl}(A)$ by the following equation:

$$\mathrm{Ctrl}(A) = \sum_{i \in \mathcal{I}} \sum_{0 < l \leq A(i) \leq N} a_i.\mathrm{Pop}(i,l) \ (+\,\mathbf{1})_{A(\mathbf{1}) \geq 1}.$$

Corollary 2. *For any context-free process P with recursive specification E that has bounded branching, there exists a regular process Q such that $P = \big[Q \parallel S_{e(X)}\big]_*$.*

4.4 Unrestricted

In the previous subsection, we showed that there is no suitable pushdown automaton for the context-free process P, if P has unbounded branching. However, this observation relies on the fact that certain τ transitions cannot be eliminated. In this subsection, we show that the intermediate solution reduces to a finite recursive specification, for any P, if we accept the axiom of *contrasimulation* [12,19]:

$$a.(\tau.x + \tau.y) = a.x + a.y \qquad (a \in \mathcal{A}).$$

By this we weaken the equivalence on our transition systems. We do not know whether there is a stronger equivalence in the linear-time – branching-time spectrum II [12] for which a solution exists.

Starting from the intermediate solution (see Sect. 4.1), we can derive the following using the axiom of contrasimulation. In the first step, we use the

observation that, given some $i \in \mathcal{I}$ and $0 < l \leq h(\xi)(i)$, there exists a $\xi_{i,l}$ such that $\langle V(i), h(\xi_{i,l})\rangle e(\xi_{i,l})$ is a suffix of the stack contents $e(\xi)$, reachable through the forgetfulness of the stack. In the last step, we use the claim that $\sum_{0<l\leq h(\xi)(i)} \left[\mathrm{Pop}(i,l) \parallel S_{e(\xi)}\right]_* = \left[\mathrm{Pop}(i) \parallel S_{e(\xi)}\right]_*$.

$$\begin{aligned}
\left[\mathrm{Ctrl}(h(\xi)) \parallel S_{e(\xi)}\right]_* &= \sum_{i\in\mathcal{I}} \sum_{0<l\leq h(\xi)(i)} a_i.\left[\mathrm{Pop}(i,l) \parallel S_{e(\xi)}\right]_* \ (+\ \dots) \\
&= \sum_{i\in\mathcal{I}} \sum_{0<l\leq h(\xi)(i)} a_i.\tau.\left[\hat{V}(i)(i,h(\xi_{i,l})) \parallel S_{e(\xi_{i,l})}\right]_* \ (+\ \dots) \\
&= \sum_{\substack{i\in\mathcal{I}\\ h(\xi)(i)\geq 1}} a_i.\Big(\sum_{0<l\leq h(\xi)(i)} \tau.\tau.\left[\hat{V}(i)(i,h(\xi_{i,l})) \parallel S_{e(\xi_{i,l})}\right]_*\Big) \ (+\ \dots) \\
&= \sum_{\substack{i\in\mathcal{I}\\ h(\xi)(i)\geq 1}} a_i.\Big(\sum_{0<l\leq h(\xi)(i)} \tau.\left[\hat{V}(i)(i,h(\xi_{i,l})) \parallel S_{e(\xi_{i,l})}\right]_*\Big) \ (+\ \dots) \\
&= \sum_{\substack{i\in\mathcal{I}\\ h(\xi)(i)\geq 1}} a_i.\Big(\sum_{0<l\leq h(\xi)(i)} \left[\mathrm{Pop}(i,l) \parallel S_{e(\xi)}\right]_*\Big) \ (+\ \dots) \\
&= \sum_{\substack{i\in\mathcal{I}\\ h(\xi)(i)\geq 1}} a_i.\left[\mathrm{Pop}(i) \parallel S_{e(\xi)}\right]_* \ (+\ \dots).
\end{aligned}$$

We can reduce our specification by replacing $\mathrm{Ctrl}(A)$ and introducing $\mathrm{Pop}(i)$:

$$\begin{aligned}
\mathrm{Ctrl}(A) &= \sum_{\substack{i\in\mathcal{I}\\ A(i)\geq 1}} a_i.\mathrm{Pop}(i) \ (+\,\mathbf{1})_{A(\mathbf{1})\geq 1}, \\
\mathrm{Pop}(i) &= \sum_{\substack{\langle V,A\rangle\in D\\ i\in I_V}} ?\langle V,A\rangle.\hat{V}(i,A).
\end{aligned}$$

Finally, because we never inspect the multiplicity of an index in a multiset nor remove an element, we can replace multisets by sets and use $i \in A$ instead of $A(i) \geq 1$ and $\cup$ instead of $\uplus$.

Corollary 3. *For any context-free process P with recursive specification E, there exists a regular process Q such that $P = \left[Q \parallel S_{e(X)}\right]_*$, assuming the axiom of contrasimulation.*

5 Concluding Remarks

Every context-free process can be realized as a pushdown automaton. A pushdown automaton in concurrency theory is a regular process communicating with a stack.

We define a context-free process as the bisimulation equivalence class of a transition system given by a finite guarded recursive specification over Sequential Algebra. This algebra is needed for a full correspondence with automata theory, and includes constants $\mathbf{0}, \mathbf{1}$ not included in previous definitions of a context-free process.

The most difficult case is when the given context-free process has unbounded branching. This can only happen when a state of the system is given by a sequence of variables that have $\mathbf{1}$-summands. In this case, there is no solution in rooted branching bisimulation semantics. We have found a solution in contrasimulation semantics, but do not know whether there are stronger equivalences in the spectrum of [12] for which a solution exists.

Concerning the reverse direction, not every regular process communicating with a stack is a context-free process. First of all, one must allow τ steps in the definition of context-free processes, because not all τ-steps of a pushdown automaton can be removed modulo rooted branching bisimulation or contrasimulation. Moreover, even if we allow τ steps, the theory of [17] shows that pushdown automata are more expressive than context-free processes without $\mathbf{1}$. It is not trivial whether this result is still true when the expressivity of context-free processes is enlarged by adding termination. Research in this direction is left as future work.

The other famous result concerning context-free processes is the fact that bisimulation equivalence is decidable on this class, see [11]. Again, this result has been established for processes not including $\mathbf{0}, \mathbf{1}$. We expect that addition of $\mathbf{0}$ will not cause any difficulties, but addition of $\mathbf{1}$ will. We leave as an open problem whether bisimulation is decidable on the class of context-free processes as we have defined it.

Most questions concerning regular processes are settled, as we discussed in Sect. 2. A very important class of processes to be considered next are the computable processes. In [3], it was demonstrated that a Turing machine in concurrency theory can be presented as a regular process communicating with two stacks. By this means, it was established that every computable process can be realized as the abstraction of a solution of a finite guarded recursive specification over communication algebra. This result also holds in the presence of the constant $\mathbf{1}$.

There are more classes of processes to be considered. The class of so-called *basic parallel processes* is given by finite guarded recursive specifications over Minimal Algebra extended with parallel composition (without communication). A prime example of such a process is the *bag*. Does the result of [10], that bisimulation is decidable on this class, still hold in the presence of $\mathbf{1}$? Can we write every basic parallel process as a regular process communicating with a bag?

Acknowledgments

We would like to thank the members of the Formal Methods group, in particular Bas Luttik, for their comments, suggestions and vlaai.

The research of Van Tilburg was supported by the project "Models of Computation: Automata and Processes" (nr. 612.000.630) of the Netherlands Organization for Scientific Research (NWO).

References

1. Aceto, L., Fokkink, W.J., Verhoef, C.: Structural operational semantics. In: J. Bergstra, A. Ponse, S. Smolka (eds.) Handbook of Process Algebra, pp. 197–292. North-Holland (2001)
2. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes. Cambridge University Press (2008)
3. Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: On the consistency of Koomen's fair abstraction rule. Theoretical Computer Science **51**(1–2), 129–176 (1987)
4. Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: Decidability of bisimulation equivalence for processes generating context-free languages. Journal of the ACM **40**(3), 653–682 (1993)
5. Baeten, J.C.M., Bravetti, M.: A ground-complete axiomatization of finite state processes in process algebra. In: M. Abadi, L. de Alfaro (eds.) Proceedings of CONCUR 2005, no. 3653 in LNCS, pp. 246–262. Springer (2005)
6. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Cambridge University Press (1990)
7. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control **60**(1/3), 109–137 (1984)
8. Bosscher, D.J.B.: Grammars modulo bisimulation. Ph.D. thesis, University of Amsterdam (1997)
9. Caucal, D.: Branching bisimulation for context-free processes. In: R. Shyamasundar (ed.) Proceedings of FSTTCS'92, no. 652 in LNCS, pp. 316–327. Springer-Verlag (1992)
10. Christensen, S., Hirshfeld, Y., Moller, F.: Bisimulation equivalence is decidable for basic parallel processes. In: E. Best (ed.) Proceedings of CONCUR 1993, no. 715 in LNCS, pp. 143–157. Springer-Verlag (1993)
11. Christensen, S., Hüttel, H., Stirling, C.: Bisimulation equivalence is decidable for all context-free processes. In: W. Cleaveland (ed.) Proceedings of CONCUR 1992, no. 630 in LNCS, pp. 138–147. Springer-Verlag (1992)
12. Glabbeek, R.J. van: The linear time – branching time spectrum ii. In: E. Best (ed.) Proceedings of CONCUR 1993, no. 715 in LNCS, pp. 66–81. Springer-Verlag (1993)
13. Glabbeek, R.J. van, Weijland, W.P.: Branching time and abstraction in bisimulation semantics. Journal of the ACM **43**(3), 555–600 (1996)
14. Groote, J.F., Reniers, M.A.: Algebraic process verification. In: J. Bergstra, A. Ponse, S. Smolka (eds.) Handbook of Process Algebra, pp. 1151–1208. North-Holland (2001)
15. Luttik, B.: Choice quantification in process algebra. Ph.D. thesis, University of Amsterdam (2002)
16. Moller, F.: The importance of the left merge operator in process algebras. In: M. Paterson (ed.) Proceedings of ICALP'90, no. 443 in LNCS, pp. 752–764. Springer-Verlag (1990)
17. Moller, F.: Infinite results. In: U. Montanari, V. Sassone (eds.) Proceedings of CONCUR '96, no. 1119 in LNCS, pp. 195–216. Springer-Verlag (1996)
18. Srba, J.: Deadlocking states in context-free process algebra. In: L. Brim, J. Gruska, J. Zlatuska (eds.) Proceedings of MFSC'98, no. 1450 in LNCS, pp. 388–398. Springer-Verlag (1998)
19. Voorhoeve, M., Mauw, S.: Impossible futures and determinism. Information Processing Letters **80**(1), 51–58 (2001)