A Process-Theoretic Look at Automata

J.C.M. Baeten, P.J.L. Cuijpers, B. Luttik, 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,s.p.luttik,p.j.a.v.tilburg}@tue.nl

Abstract. Automata theory presents roughly three types of automata: finite automata, pushdown automata and Turing machines. The automata are treated as language acceptors, and the expressiveness of the automata models are considered modulo language equivalence. This notion of equivalence is arguably too coarse to satisfactorily deal with a notion of interaction that is fundamental to contemporary computing. In this paper we therefore reconsider the automaton models from automata theory modulo branching bisimilarity, a well-known behavioral equivalence from process theory that has proved to be able to satisfactorily deal with interaction. We investigate to what extent some standard results from automata theory are still valid if branching bisimilarity is adopted as the preferred equivalence.

1 Introduction

Automata theory is the study of abstract computing devices, or "machines" [11]. It presents and studies roughly three types of automata: finite automata, pushdown automata and Turing machines. Finite automata are the simplest kind of automata; they are widely used to model and analyze finite-state state systems. Pushdown automata add to finite automata a restricted kind of unbounded memory in the form of a stack. Turing machines add to finite automata a more powerful notion of memory in the form of an unbounded tape.

In traditional automata theory, automata are treated as language acceptors. The idea is that a string accepted by the automaton represents a particular computation of the automaton, and the language accepted by it thus corresponds with the set of all computations of the automaton. The language-theoretic interpretation of automata is at the basis of all the standard results taught in an undergraduate course on the subject. For instance, the procedure of transforming a nondeterministic finite automaton into a deterministic one is deemed correct because the resulting automaton is language equivalent to the original automaton (two automata are language equivalent if they accept the same language). Another illustrative example is the correspondence between pushdown automata and context-free grammars: for every language generated by a context-free grammar there is a pushdown automaton that accepts it, and vice versa.

The language-theoretic interpretation abstracts from the moments of choice within an automaton. (For instance, it does not distinguish between, on the

one hand, the automaton that first accepts an a and subsequently chooses between accepting a b or a c, and, on the other hand, the automaton that starts with a choice between accepting ab and accepting ac.) As a consequence, the language-theoretic interpretation is only suitable under the assumption that an automaton is a stand-alone computational device; it is unsuitable if one some form interaction of the automaton with its environment (user, other automata running in parallel, etc.) may influence the course of computation.

Interaction and nondeterminism nowadays play a crucial role in computing systems. For instance, selecting an option in a web form can lead to different responses and different options in the following form, and so a fixed input string does not look so useful. Also, one browser query will lead to different answers every day, so it is difficult to see a computer as a function from input to output, there is inherent nondeterminism.

Process theory is the study of reactive systems, i.e., systems that depend on interaction with their environment during their execution. In process theory, a system is usually either directly modeled as a labeled transition system (which is actually a generalization of the notion of finite automaton), or as an expression in a process description language with a well-defined operational semantics that assigns a labeled transition system to each expression. In process theory, interaction between systems is treated as a first-class citizen. One its main contributions is a plethora of behavioral equivalences that to more or lesser extent preserve the branching structure of an automaton (see [7] for an overview). One of the finest behavioral equivalences studied in process theory, which arguably preserves all relevant moments of choice in a system, is *branching bisimilarity* [10].

In this paper we shall reconsider some of the standard results from automata theory when automata are considered modulo *branching bisimilarity* instead of language equivalence. We prefer to use branching bisimilarity because it arguably preserves all relevant moments of choice in a system [10]. Note that all the positive results obtained in this paper automatically also hold modulo any of the coarser behavioral equivalences, and hence also modulo Milner's observation equivalence [12]. Furthermore, it is fairly easy to see that most of our negative results also hold modulo observation equivalence; branching structure is needed only to a limited extent.

In Section 3 we consider *regular processes*, defined as branching bisimulation equivalence classes of labeled transition systems associated with finite automata. Most of the results we present in this section are well-known. The section is included for completeness and to illustrate the correspondence between finite automata and a special type of recursive specifications that can be thought of as the process-theoretic counterpart of regular grammars. We will obtain mostly negative results. Naturally, the determinization procedure standardly presented in automata theory to transform a nondeterministic finite automaton into a deterministic one is not valid modulo branching bisimilarity, and not every labeled transition system associated with a finite automaton is described by a regular expression. We do find a process-theoretic variant of the correspondence between

finite automata and right-linear grammars, while there is no process-theoretic variant of the correspondence between finite automata and left-linear grammars.

In Section 4 we consider *pushdown processes*, defined as branching bisimulation equivalence classes of labeled transition systems associated with pushdown automata. First we investigate three alternative termination conditions: termination by final state, by empty stack, or by both. Recall that these termination conditions are equivalent from a language-theoretic perspective. We shall prove that, modulo branching bisimilarity, the termination by empty stack and termination by final state interpretations lead to different notions of pushdown process, while the termination by empty stack and the termination by empty stack and final state coincide. We argue that termination by empty stack is better suited for a process-theoretic treatment than termination by final state. Then, we shall investigate the correspondence between pushdown processes (according to the termination by empty stack and final state interpretation) and processes definable by recursive TSP_τ specifications, which can be thought of as the process-theoretic counterpart of context-free grammars. We shall argue that not every pushdown process is definable by a recursive TSP_τ specification and identify a subclass of pushdown processes that are definable by (a special type of) recursive TSP_τ specifications.

In Section 5 we consider *computable processes*, defined as branching bisimulation equivalence classes of labeled transition systems associated with Turing machines. Being a universal model of computation, the Turing machine model has been particularly influential, probably due to its close resemblance with the computer: a Turing machine can be thought of as a computer running a single program that only interacts with the outside world through its memory, and only at the very beginning and the very end of the execution of the program. Thus, the notion of Turing machine focuses on the computational aspect of the execution of a computer; in particular, it abstracts from the interaction that a computer has with its environment (user, other computers in a network, etc.). Since we find interaction important, we shall work with a variation on the notion of Turing machine, known as *off-line* Turing machine. This notion starts with an empty tape, and the machine can take in one input symbol at a time. In [2], a computable process was defined indirectly, by using an encoding of a transition system by means of two computable functions (defined by a standard Turing machine). We show the two ways yield the same set of computable processes.

2 Process theory

In this section we briefly recap the basic definitions of the process algebra TCP_τ (Theory of Communicating Processes with τ). We refer to [1] for further details.

Syntax We presuppose a countably infinite *action alphabet* $\mathcal{A}$, and a countably infinite set of *names* $\mathcal{N}$. The actions in $\mathcal{A}$ denote the basic events that a process may perform. In this paper we shall furthermore presuppose a countably infinite *data alphabet* $\mathcal{D}$, a finite set $\mathcal{C}$ of *channels*, and assume that $\mathcal{A}$ includes special

actions $c?d$, $c!d$, $c\mathbin{?!}d$ ($d \in \mathcal{D}$, $c \in \mathcal{C}$), which, intuitively, denote the event that datum d is received, sent, or communicated along *channel* c.

Let $\mathcal{N}'$ be a finite subset of $\mathcal{N}$. The set of *process expressions* $\mathcal{P}$ over $\mathcal{A}$ and $\mathcal{N}'$ is generated by the following grammar:

$$p ::= \mathbf{0} \mid \mathbf{1} \mid a.p \mid \tau.p \mid p \cdot p \mid p + p \mid p \parallel p \mid \partial_c(p) \mid \tau_c(p) \mid N \qquad (a \in \mathcal{A}, N \in \mathcal{N}', c \in \mathcal{C}) \ .$$

Let us briefly comment on the operators in this syntax. The constant $\mathbf{0}$ denotes *deadlock*, the unsuccessfully terminated process. The constant $\mathbf{1}$ denotes the successfully terminated process. For each action $a \in \mathcal{A}$ there is a unary operator $a.$ denoting action prefix; the process denoted by $a.p$ can do an a-transition to the process denoted by p. The τ-transitions of a process will, in the semantics below, be treated as unobservable, and as such they are the process-theoretic counterparts of the so-called λ- or ϵ-transitions in the theory of automata and formal languages. For convenience, whenever $\mathcal{A}'$ is some subset of $\mathcal{A}$, we write $\mathcal{A}'_\tau$ for $\mathcal{A}' \cup \{\tau\}$. The binary operator $\cdot$ denotes *sequential composition*. The binary operator $+$ denotes *alternative composition* or *choice*. The binary operator $\parallel$ denotes *parallel composition*; actions of both arguments are interleaved, and in addition a communication $c\mathbin{?!}d$ of a datum d on channel c can take place if one argument can do an input action $c?d$ that matches an output action $c!d$ of the other component. The unary operator $\partial_c(p)$ encapsulates the process p in such a way that all input actions $c?d$ and output actions $c!d$ are blocked (for all data) so that communication is enforced. Finally, the unary operator $\tau_c(p)$ denotes abstraction from communication over channel c in p by renaming all communications $c\mathbin{?!}d$ to τ-transitions.

Let $\mathcal{N}'$ be a finite subset of $\mathcal{N}$, used to define processes by means of (recursive) equations. A *recursive specification* E over $\mathcal{N}'$ is a set of equations of the form

$$N \stackrel{\text{def}}{=} p$$

with as left-hand side a name N and as right-hand side a process expression p. It is required that a recursive specification E contains, for every $N \in \mathcal{N}'$, precisely one equation with N as left-hand side; this equation will be referred to as the *defining equation* for N in $\mathcal{N}'$.

One way to formalize the operational intuitions we have for the syntactic constructions of TCP_τ, is to associate with every process expression a labeled transition system.

Definition 1 (Labelled Transition System). *A* labeled transition system L *is defined as a four-tuple* $(\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ *where:*

1. $\mathcal{S}$ *is a set of states,*
2. $\rightarrow \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathcal{S}$ *is an* $\mathcal{A}_\tau$*-labeled* transition relation *on* $\mathcal{S}$*,*
3. $\uparrow \in \mathcal{S}$ *is the* initial *state,*
4. $\downarrow \subseteq \mathcal{S}$ *is the set of* final *states.*

If $(s, a, t) \in \rightarrow$*, we write* $s \xrightarrow{a} t$*. If* s *is a final state, i.e.,* $s \in \downarrow$*, we write* $s\downarrow$*.*

We use Structural Operational Semantics [15] to associate a transition relation with process expressions: we let $\rightarrow$ be the $\mathcal{A}_\tau$-labeled transition relation induced on the set of process expressions $\mathcal{P}$ by operational rules in Table 1. Note that the operational rules presuppose a recursive specification E.

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

$$\frac{p \xrightarrow{a} p'}{(p+q) \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{(p+q) \xrightarrow{a} q'} \qquad \frac{p\downarrow}{(p+q)\downarrow} \qquad \frac{q\downarrow}{(p+q)\downarrow}$$

$$\frac{p \xrightarrow{a} p'}{p \cdot q \xrightarrow{a} p' \cdot q} \qquad \frac{p\downarrow \quad q \xrightarrow{a} q'}{p \cdot q \xrightarrow{a} q'} \qquad \frac{p\downarrow \quad q\downarrow}{p \cdot q\downarrow}$$

$$\frac{p \xrightarrow{a} p'}{p \parallel q \xrightarrow{a} p' \parallel q} \qquad \frac{q \xrightarrow{a} q'}{p \parallel q \xrightarrow{a} p \parallel q'} \qquad \frac{p\downarrow \quad q\downarrow}{p \parallel q\downarrow}$$

$$\frac{p \xrightarrow{c!d} p' \quad q \xrightarrow{c?d} q'}{p \parallel q \xrightarrow{c\,!\!?\,d} p' \parallel q'} \qquad \frac{p \xrightarrow{c?d} p' \quad q \xrightarrow{c!d} q'}{p \parallel q \xrightarrow{c\,!\!?\,d} p' \parallel q'}$$

$$\frac{p \xrightarrow{a} p' \quad a \neq c?d, c!d}{\partial_c(p) \xrightarrow{a} \partial_c(p')} \qquad \frac{p\downarrow}{\partial_c(p)\downarrow}$$

$$\frac{p \xrightarrow{c\,!\!?\,d} p'}{\tau_c(p) \xrightarrow{\tau} \tau_c(p')} \qquad \frac{p \xrightarrow{a} p' \quad a \neq c\,!\!?\,d}{\tau_c(p) \xrightarrow{a} \tau_c(p')} \qquad \frac{p\downarrow}{\tau_c(p)\downarrow}$$

$$\frac{p_N \xrightarrow{a} p \quad (N \stackrel{\text{def}}{=} p_N) \in E}{N \xrightarrow{a} p} \qquad \frac{p_N\downarrow \quad (N \stackrel{\text{def}}{=} p_N) \in E}{N\downarrow}$$

Table 1: Operational rules for a recursive specification E (a ranges over $\mathcal{A}_\tau$, d ranges over $\mathcal{D}$, and c ranges over $\mathcal{C}$).

Let $\rightarrow$ be an $\mathcal{A}_\tau$-labeled transition relation on a set $\mathcal{S}$ of states. For $s, s' \in \mathcal{S}$ and $w \in \mathcal{A}^*$ we write $s \xrightarrow{w}\!\!\!\!\rightarrow s'$ if there exist states $s_0, \ldots, s_n \in \mathcal{S}$ and actions $a_1, \ldots, a_n \in \mathcal{A}_\tau$ such that $s = s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n = s'$ and w is obtained from $a_1 \cdots a_n$ by omitting all occurrences of τ. If $s \xrightarrow{\varepsilon}\!\!\!\!\rightarrow t$ (ε denotes the empty word), which just means that t is reachable from s by zero or more τ-transitions, then we shall simply write $s \longrightarrow\!\!\!\!\rightarrow t$.

Definition 2 (Reachability). *A state $t \in \mathcal{S}$ is* reachable *from a state $s \in \mathcal{S}$ if there exists $w \in \mathcal{A}^*$ such that $s \xrightarrow{w}\!\!\!\!\rightarrow t$.*

Definition 3. *Let E be a recursive specification and let p be a process expression. We define the labeled transition system $\mathcal{T}_E(p) = (\mathcal{S}_p, \rightarrow_p, \uparrow_p, \downarrow_p)$ associated with p and E as follows:*

1. *the set of states $\mathcal{S}_p$ consists of all process expressions reachable from p;*
2. *the transition relation $\rightarrow_p$ is the restriction to $\mathcal{S}_p$ of the transition relation $\rightarrow$ defined on all process expressions by the operational rules in Table 1, i.e., $\rightarrow_p = \rightarrow \cap (\mathcal{S}_p \times \mathcal{A}_\tau \times \mathcal{S}_p)$.*
3. *the process expression p is the initial state, i.e. $\uparrow_p = p$; and*
4. *the set of final states consists of all process expressions $q \in \mathcal{S}_p$ such that $q\downarrow$, i.e., $\downarrow_p = \downarrow \cap \mathcal{S}_p$.*

Given the set of (possibly infinite) labeled transition systems, we can divide out different equivalence relations on this set. Dividing out language equivalence throws away too much information, as the moments where choices are made are totally lost, and behavior that does not lead to a final state is ignored. An equivalence relation that keeps all relevant information, and has many good properties, is branching bisimulation as proposed by van Glabbeek and Weijland [10]. For motivations to use branching bisimulation as the preferred notion of equivalence, see [8].

Let $\rightarrow$ be an $\mathcal{A}_\tau$-labeled transition relation, and let $a \in \mathcal{A}_\tau$; we write $s \xrightarrow{(a)} t$ if $s \xrightarrow{a} t$ or $a = \tau$ and $s = t$.

Definition 4 (Branching bisimilarity). *Let $L_1 = (\mathcal{S}_1, \rightarrow_1, \uparrow_1, \downarrow_1)$ and $L_2 = (\mathcal{S}_2, \rightarrow_2, \uparrow_2, \downarrow_2)$ be labeled transition systems. A* branching bisimulation *from L_1 to L_2 is a binary relation $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ such that $\uparrow_1 \mathcal{R} \uparrow_2$ and, for all states s_1 and s_2, $s_1 \mathcal{R} s_2$ implies*

1. *if $s_1 \xrightarrow{a}_1 s_1'$, then there exist $s_2', s_2'' \in \mathcal{S}_2$ such that $s_2 \longrightarrow\!\!\!\!\!\rightarrow_2 s_2'' \xrightarrow{(a)}_2 s_2'$, $s_1 \mathcal{R} s_2''$ and $s_1' \mathcal{R} s_2'$;*
2. *if $s_2 \xrightarrow{a}_2 s_2'$, then there exist $s_1', s_1'' \in \mathcal{S}_1$ such that $s_1 \longrightarrow\!\!\!\!\!\rightarrow_1 s_1'' \xrightarrow{(a)}_1 s_1'$, $s_1'' \mathcal{R} s_2$ and $s_1' \mathcal{R} s_2'$;*
3. *if $s_1\downarrow_1$, then there exists s_2' such that $s_2 \longrightarrow\!\!\!\!\!\rightarrow_2 s_2'$ and $s_2'\downarrow_2$; and*
4. *if $s_2\downarrow_2$, then there exists s_1' such that $s_1 \longrightarrow\!\!\!\!\!\rightarrow_1 s_1'$ and $s_1'\downarrow_1$.*

The labeled transition systems L_1 and L_2 are branching bisimilar *(notation: $L_1 \leftrightarrows_b L_2$) if there exists a branching bisimulation from L_1 to L_2.*

Branching bisimilarity is an equivalence relation on labeled transition systems [5].

We need as auxiliary notions in our paper the notion of *inert* τ-transition and the notion of *branching degree* of a state. For a definition of these notions we first define the notion of branching bisimulation *on* a labeled transition system, and the notion of *quotient* of a labeled transition system by its maximal branching bisimulation.

Let $L = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ be a labeled transition system. A branching bisimulation *on* L is a binary relation $\mathcal{R}$ on $\mathcal{S}$ that satisfies conditions 1–4 of Definition 4 for all s_1 and s_2 such that $s_1 \mathcal{R} s_2$. Let $\mathcal{R}$ be the maximal branching bisimulation on L. Then $\mathcal{R}$ is an equivalence on $\mathcal{S}$; we denote by $[s]_\mathcal{R}$ the equivalence class of $s \in \mathcal{S}$ with respect to $\mathcal{R}$ and by $\mathcal{S}/\mathcal{R}$ the set of all equivalence classes of $\mathcal{S}$ with respect to $\mathcal{R}$. On $\mathcal{S}/\mathcal{R}$ we can define an $\mathcal{A}_\tau$-labeled transition relation $\rightarrow_\mathcal{R}$ by $[s]_\mathcal{R} \xrightarrow{a}_\mathcal{R} [t]_\mathcal{R}$ if, and only if, there exist

$s' \in [s]_{\mathcal{R}}$ and $t' \in [t]_{\mathcal{R}}$ such that $s' \xrightarrow{a} t'$. Furthermore, we define $\uparrow_{\mathcal{R}} = [\uparrow]_{\mathcal{R}}$ and $\downarrow_{\mathcal{R}} = \{s \mid \exists s' \in \downarrow.\ s \in [s']_{\mathcal{R}}\}$. Now, the *quotient* of L by $\mathcal{R}$ is the labeled transition system $L/\mathcal{R} = (\mathcal{S}/\mathcal{R}, \rightarrow/\mathcal{R}, \uparrow/\mathcal{R}, \downarrow/\mathcal{R})$. It is straightforward to prove that each labeled transition system is branching bisimilar to the quotient of this labeled transition system by its maximal branching bisimulation.

Definition 5 (Inert τ-transitions). *Let L be a labeled transition system and let s and t be two states in L. A τ-transition $s \xrightarrow{\tau} t$ is* inert *if s and t are related by the maximal branching bisimulation on L.*

If s and t are distinct states, then an inert τ-transition $s \xrightarrow{\tau} t$ can be *eliminated* from a labeled transition system, e.g., by removing all outgoing transitions of s, changing every outgoing transition $t \xrightarrow{a} u$ from t to an outgoing transition $s \xrightarrow{a} u$, and removing the state t. This operation yields a labeled transition system that is branching bisimilar to the original labeled transition system.

For example, consider Figure 1. Here, the inert τ-transition from state s to t in the transition system on the left is removed by removing the transition $s \xrightarrow{a} u$ and moving all outgoing transitions of t to s, resulting in the transition system on the right. This is possible because s and t are branching bisimilar.

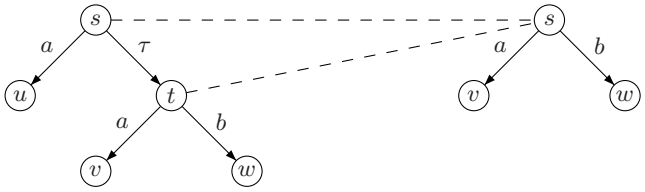

Fig. 1. Removing an inert τ-transition.

To get a notion of branching degree that is preserved modulo branching bisimilarity, we define the branching degree of a state as the branching degree of the corresponding equivalence class of states modulo the maximal branching bisimilarity.

Definition 6 (Branching degree). *Let L be a labeled transition system, and let $\mathcal{R}$ be its maximal branching bisimulation. The branching degree of a state s in L is the cardinality of the set $\{(a, [t]_{\mathcal{R}}) \mid [s]_{\mathcal{R}} \xrightarrow{a}_{\mathcal{R}} [t]_{\mathcal{R}}\}$ of outgoing edges of the equivalence class of s in the quotient $L/\mathcal{R}$.*

We say that L has finite branching *if all states of L have a finite branching degree. We say that L has* bounded branching *if there exists a natural number $n \geq 0$ such that every state has a branching degree of at most n.*

Branching bisimulations respect branching degrees in the sense that if $\mathcal{R}$ is a branching bisimulation from L_1 to L_2, s_1 is a state in L_1 and s_2 is a state in L_2 such that $s_1\ \mathcal{R}\ s_2$, then s_1 and s_2 have the same branching degree. Let p

and q be process expressions in the context of a recursive specification E; the following properties pertaining to branching degrees are fairly straightforward to establish: If $\mathcal{T}_E(p)$ and $\mathcal{T}_E(q)$ have bounded branching (or finite branching), then $\mathcal{T}_E(p \parallel q)$ has bounded branching (or finite branching) too, and if $\mathcal{T}_E(p)$ has bounded branching (or finite branching), then $\mathcal{T}_E(\partial_c(p))$ and $\mathcal{T}_E(\tau_c(p))$ have bounded branching (or finite branching) too.

3 Regular processes

A computer with a fixed-size, finite memory is just a finite control. This can be modeled by a finite automaton. Automata theory starts with the notion of a finite automaton. As non-determinism is relevant and basic in concurrency theory, we look at a non-deterministic finite automaton.

Definition 7 (Finite automaton). *A* finite automaton M *is defined as a five-tuple* $(\mathcal{S}, \mathcal{A}', \rightarrow, \uparrow, \downarrow)$ *where:*

1. $\mathcal{S}$ *is a finite set of states,*
2. $\mathcal{A}'$ *is a finite subset of* $\mathcal{A}$*,*
3. $\rightarrow \subseteq \mathcal{S} \times \mathcal{A}'_\tau \times \mathcal{S}$ *is a finite* $\mathcal{A}'_\tau$*-labeled* transition relation *on* $\mathcal{S}$*,*
4. $\uparrow \in \mathcal{S}$ *is the* initial *state,*
5. $\downarrow \subseteq \mathcal{S}$ *is the set of* final *states.*

Clearly, from a finite automaton we obtain a labeled transition system by simply omitting $\mathcal{A}'$ from the five-tuple and declaring $\rightarrow$ to be an $\mathcal{A}_\tau$-labeled transition relation. In the remainder of this paper there is no need to make the formal distinction between a finite automaton and the labeled transition system thus associated to it.

Two examples of finite automata are given in Figure 2.

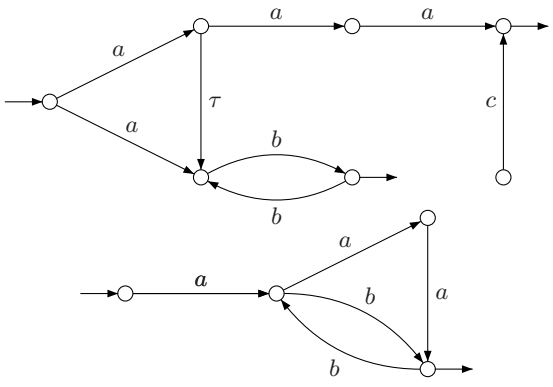

Fig. 2. Two examples of finite automata.

Definition 8 (Deterministic finite automaton). *A finite automaton* $M = (\mathcal{S}, \mathcal{A}', \rightarrow, \uparrow, \downarrow)$ *is* deterministic *if, for all states* $s, t_1, t_2 \in \mathcal{S}$ *and for all actions* $a \in \mathcal{A}'$, $s \xrightarrow{a}\!\!\!\!\!\rightarrow t_1$ *and* $s \xrightarrow{a}\!\!\!\!\!\rightarrow t_2$ *implies* $t_1 = t_2$.

In the theory of automata and formal languages, it is usually also required in the definition of deterministic that the transition relation is *total* in the sense that for all $s \in \mathcal{S}$ and for all $a \in \mathcal{A}'$ there exists $t \in \mathcal{S}$ such that $s \xrightarrow{a} t$. The extra requirement is clearly only sensible in the language interpretation of automata; we shall not be concerned with it here.

The upper automaton in Figure 2 is non-deterministic and has an unreachable c-transition. The lower automaton is deterministic and does not have unreachable transitions; it is not total.

In the theory of automata and formal languages, finite automata are considered as language acceptors.

Definition 9 (Language equivalence). *The* language $\mathcal{L}(L)$ *accepted by a labeled transition system* $L = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ *is defined as*

$$\mathcal{L}(L) = \{w \in \mathcal{A}^* \mid \exists s \in \downarrow \textit{ such that } \uparrow \xrightarrow{w}\!\!\!\!\!\rightarrow s\} \ .$$

Labeled transition systems L_1 *and* L_2 *are* language equivalent *(notation:* $L_1 \equiv L_2$*) if* $\mathcal{L}(L_1) = \mathcal{L}(L_2)$.

Recall that a finite automaton is a special kind of labeled transition system, so the above definition pertains directly to finite automata. The language of both automata in Figure 2 is $\{aaa\} \cup \{ab^{2n-1} \mid n \geq 1\}$; the automata are language equivalent.

A language $L \subseteq \mathcal{A}^*$ accepted by a finite automaton is called a *regular language*. A *regular process* is a branching bisimilarity class of labeled transition systems that contains a finite automaton.

The following standard results pertaining to finite automata are found in every textbook on the theory of automata and formal languages:

1. For every finite automaton there exists a language equivalent automaton without τ-transitions.
2. For every finite automaton there exists a language equivalent *deterministic* finite automaton.
3. Every language accepted by a finite automaton is the language described by a regular expression, and, conversely, every language described by a regular expression is accepted by a finite automaton.
4. Every language accepted by a finite automaton is generated by a regular (i.e., right-linear or left-linear) grammar, and, conversely, every language generated by a regular grammar is accepted by a finite automaton.

We shall discuss below to what extent these results are still valid in branching bisimulation semantics.

Silent steps and non-determinism Not every regular process has a representation as a finite automaton without τ-transitions, and not every regular process has a representation as a deterministic finite automaton. In fact, it can be proved that there does not exist a finite automaton without τ-transitions that is branching bisimilar with the upper finite automaton in Figure 2. Nor does there exist a deterministic finite automaton branching bisimilar with the upper finite automaton in Figure 2.

Regular grammars and regular expressions Not every regular process is given by a regular expression, see [3]. We show a simple example in Figure 3 of a finite transition system that is not bisimilar to any transition system that can be associated with a regular expression.

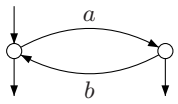

Fig. 3. Not bisimilar to a regular expression.

In the theory of automata and formal languages, the notion of *grammar* is used as a syntactic mechanism to describe languages. The corresponding mechanism in concurrency theory is the notion of recursive specification.

We shall now consider the process theory BSP_τ (Basic Sequential Processes), which is a subtheory of the theory TCP_τ introduced in Section 2. The syntax of the process theory BSP_τ is obtained from that of TCP_τ by omitting sequential composition, parallel composition, encapsulation and abstraction. A BSP_τ recursive specification over a finite subset $\mathcal{N}'$ of $\mathcal{N}$ is a recursive specification over $\mathcal{N}'$ in which only $\mathbf{0}$, $\mathbf{1}$, N ($N \in \mathcal{N}'$), $a\_$ ($a \in \mathcal{A}_\tau$) and $\_ + \_$ are used to build process expressions.

Consider the operational rules in Table 1 that are relevant for BSP_τ, for a presupposed recursive specification E. Note that whenever p is a BSP_τ process expression and $p \xrightarrow{a} q$ then q is again a BSP_τ process expression. Moreover, q is a subterm of p, or q is a subterm of a right-hand side of the recursive specification E. Thus, it follows that the set of process expressions reachable from a BSP_τ process expression consists merely of BSP_τ process expressions, and that it is finite. So the labeled transition system $\mathcal{T}_E(p)$ associated with a BSP_τ process expression given a BSP_τ recursive specification E is a finite automaton. Below we shall also establish the converse, that every finite automaton can be specified, up to isomorphism, by a recursive specification over BSP_τ. First we illustrate the construction with an example.

Example 1. Consider the automaton depicted in Figure 4. Note that we have labeled each state of the automaton with a unique name; these will be the names of a recursive specification E. We will define each of these names with

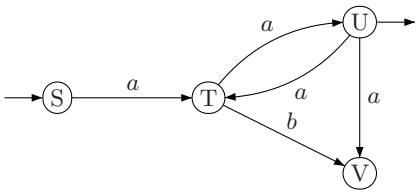

Fig. 4. Example automaton.

an equation, in such a way that the labeled transition system $\mathcal{T}_E(S)$ generated by the operational semantics in Table 1 is isomorphic (so certainly branching bisimilar) with the automaton in Figure 4.

The recursive specification for the finite automaton in Figure 4 is:

$$S \stackrel{\text{def}}{=} a.T \ ,$$
$$T \stackrel{\text{def}}{=} a.U + b.V \ ,$$
$$U \stackrel{\text{def}}{=} a.V + \mathbf{1} \ ,$$
$$V \stackrel{\text{def}}{=} \mathbf{0} \ .$$

The action prefix $a.T$ on the right-hand side of the equation defining S is used to express that S has an a-transition to T. Alternative composition is used on the right-hand side of the defining equation for T to combine the two transitions going out from T. The $\mathbf{1}$-summand on the right-hand side of the defining equation for U indicates that U is a final state. The symbol $\mathbf{0}$ on the right-hand side of the defining equation for V expresses that V is a deadlock state.

Theorem 1. *For every finite automaton M there exists a BSP_τ recursive specification E and a BSP_τ process expression p such that $M \leftrightarrows_b \mathcal{T}_E(p)$.*

Proof. The general procedure is clear from Example 1. Let $M = (\mathcal{S}, \mathcal{A}', \rightarrow, \uparrow, \downarrow)$. We associate with every state $s \in \mathcal{S}$ a name N_s, and define a recursive specification E on $\{N_s \mid s \in \mathcal{S}\}$. The recursive specification E consists of equations of the form

$$N_s \stackrel{\text{def}}{=} \sum\{a.N_t \mid s \stackrel{a}{\longrightarrow} t\} \ [+ \mathbf{1}] \ ,$$

with the contention that the summation $\sum\{a.N_t \mid s \stackrel{a}{\longrightarrow} t\}$ denotes $\mathbf{0}$ if the set $\{a.N_t \mid s \stackrel{a}{\longrightarrow} t\}$ is empty, and the optional $\mathbf{1}$-summand is present if, and only if, $s\downarrow$. It is easily verified that the binary relation $\mathcal{R} = \{(s, N_s) \mid s \in \mathcal{S}\}$ is a branching bisimulation. □

Incidentally, note that the relation $\mathcal{R}$ in the proof of the above theorem is an isomorphism, so the proof actually establishes that for every finite automaton M there exists a BSP_τ recursive specification E and a BSP_τ process expression

p such that the labeled transition system associated with p and E is isomorphic to M.

The above theorem can be viewed as the process-theoretic counterpart of the result from the theory of automata and formal languages that states that every language accepted by a finite automaton is generated by a so-called *right-linear* grammar. There is no reasonable process-theoretic counterpart of the similar result in the theory of automata and formal languages that every language accepted by a finite automaton is generated by a *left-linear* grammar, as we shall now explain.

$$\frac{p \xrightarrow{\beta} p'}{p.a \xrightarrow{\beta} p'.a} \qquad \frac{p\!\downarrow}{p.a \xrightarrow{a} \mathbf{1}}$$

Table 2: Operational rules for action postfix operators ($a, \beta \in \mathcal{A}_\tau$).

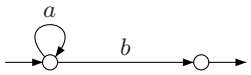

Fig. 5. A simple finite automaton.

To obtain the process-theoretic counterpart of a left-linear grammar, we should replace the action prefixes $a._-$ in BSP_τ by *action postfixes* $\_.a$, with the operational rules in Table 2. Not every finite automaton can be specified in the resulting language. To see this, note that action postfix distributes over alternative composition and is absorbed by $\mathbf{0}$. Therefore, for every process expression p over BSP_τ with action postfix instead of action prefix there exist finite sets I and J and elements w_i ($i \in I$) and w_j ($j \in J$) of $\mathcal{A}^*$ such that

$$p \leftrightarrows_{\mathrm{b}} \sum_{i \in I} N_i.w_i + \sum_{j \in J} \mathbf{1}.w_j \ [+\mathbf{1}] \ .$$

(Recall that empty summations are assumed to denote $\mathbf{0}$.) Hence, for every such process expression p, if $p \xrightarrow{a} p'$, then $p' \leftrightarrows_{\mathrm{b}} w$ for some $w \in \mathcal{A}^*$. A process expression denoting the finite automaton in Figure 5 cannot have this property, for after performing an a-transition there is still a choice between terminating with a b-transition, or performing another a-transition. We conclude that the automaton in Figure 5 cannot be described modulo branching bisimilarity in BSP_τ with action postfix instead of action prefix.

Conversely, with action postfixes instead of action prefixes in the syntax, it is possible to specify labeled transition systems that are not branching bisimilar with a finite automaton.

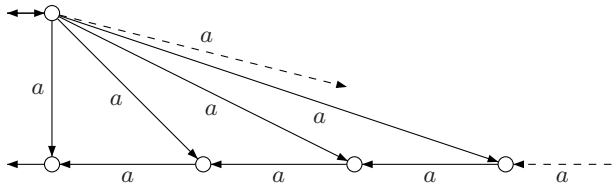

Fig. 6. Infinitely branching process of unguarded equation.

Example 2. For instance, consider the recursive specification over $\{X\}$ consisting of the equation

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

The labeled transition system associated with X by the operational semantics is depicted in Figure 6. Note that in this figure, the initial state is also final. It can be proved that the infinitely many states of the depicted labeled transition systems are all distinct modulo branching bisimilarity. It follows that the labeled transition system associated with X is not branching bisimilar to a finite automaton.

We conclude that the classes of processes defined by right-linear and left-linear grammars do not coincide.

4 Pushdown and context-free processes

As an intermediate between the notions of finite automaton and Turing machine, the theory of automata and formal languages treats the notion of pushdown automaton, which is a finite automaton with a stack as memory. Several definitions of the notion appear in the literature, which are all equivalent in the sense that they accept the same languages.

Definition 10 (Pushdown automaton). *A* pushdown automaton M *is defined as a six-tuple* $(\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ *where:*

1. $\mathcal{S}$ *a finite set of states,*
2. $\mathcal{A}'$ *is a finite subset of* $\mathcal{A}$*,*
3. $\mathcal{D}'$ *is a finite subset of* $\mathcal{D}$*,*
4. $\rightarrow \subseteq \mathcal{S} \times (\mathcal{D}' \cup \{\varepsilon\}) \times \mathcal{A}'_\tau \times \mathcal{D}'^* \times \mathcal{S}$ *is a* $(\mathcal{D}' \cup \{\varepsilon\}) \times \mathcal{A}'_\tau \times \mathcal{D}'^*$*-labeled transition relation on* $\mathcal{S}$*,*
5. $\uparrow \in \mathcal{S}$ *is the initial state, and*
6. $\downarrow \subseteq \mathcal{S}$ *is the set of final states.*

If $(s, d, a, \delta, t) \in \rightarrow$*, we write* $s \xrightarrow{d,a,\delta} t$*.*

The pair of a state together with particular stack contents will be referred to as the *configuration* of a pushdown automaton. Intuitively, a transition $s \xrightarrow{d,a,\delta} t$ (with $a \in \mathcal{A}'$) means that the automaton, when it is in a configuration consisting of a state s and a stack with the datum d on top, can consume input symbol a, replace d by the string δ and move to state t. Likewise, writing $s \xrightarrow{\varepsilon,a,\delta} t$ means that the automaton, when it is in state s and the stack is empty, can consume input symbol a, put the string δ on the stack, and move to state t. Transitions of the form $s \xrightarrow{d,\tau,\delta} t$ or $s \xrightarrow{\varepsilon,\tau,\delta} t$ do not entail the consumption of an input symbol, but just modify the stack contents.

When considering a pushdown automaton as a language acceptor, it is generally assumed that it starts in its initial state with an empty stack. A computation consists of repeatedly consuming input symbols (or just modifying stack contents without consuming input symbols). When it comes to determining whether or not to accept an input string there are two approaches: "acceptance by final state" (FS) and "acceptance by empty stack" (ES). The first approach accepts a string if the pushdown automaton can move to a configuration with a final state by consuming the string, ignoring the contents of the stack in this configuration. The second approach accepts the string if the pushdown automaton can move to a configuration with an empty stack, ignoring whether the state of this configuration is final or not. These approaches are equivalent from a language-theoretic point of view, but not from a process-theoretic point of view, as we shall see below. We shall also consider a third approach in which a configuration is terminating if it consists of a terminating state *and* an empty stack (FSES). We shall see that, from a process-theoretic point of view, the ES and FSES approaches lead to the same notion of pushdown process, whereas the FS approach leads to a different notion.

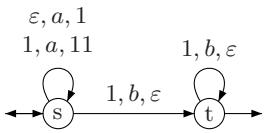

Fig. 7. Example pushdown automaton.

Depending on the adopted acceptance condition, the pushdown automaton in Figure 7 accepts the language $\{a^m b^n \mid m \geq n \geq 0\}$ (FS) or the language $\{a^n b^n \mid n \geq 0\}$ (ES,FSES).

Definition 11. *Let $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ be a pushdown automaton. The labeled transition system $\mathcal{T}(M)$ associated with M is defined as follows:*

1. *the set of states of $\mathcal{T}(M)$ is $\mathcal{S} \times \mathcal{D}'^*$;*
2. *the transition relation of $\mathcal{T}(M)$ satisfies*

   (a) $(s, d\zeta) \xrightarrow{a} (t, \delta\zeta)$ *iff* $s \xrightarrow{d,a,\delta} t$ *for all* $s, t \in \mathcal{S}$, $a \in \mathcal{A}'_\tau$, $d \in \mathcal{D}'$, $\delta, \zeta \in \mathcal{D}'^*$, *and*
   (b) $(s, \varepsilon) \xrightarrow{a} (t, \delta)$ *iff* $s \xrightarrow{\varepsilon,a,\delta} t$;
3. *the initial state of* $\mathcal{T}(M)$ *is* $(\uparrow, \varepsilon)$; *and*
4. *for the set of final states* $\downarrow$ *we consider three alternative definitions:*
   (a) $(s, \zeta)\downarrow$ *in* $\mathcal{T}(M)$ *iff* $s\downarrow$ *(the FS interpretation),*
   (b) $(s, \zeta)\downarrow$ *in* $\mathcal{T}(M)$ *iff* $\zeta = \varepsilon$ *(the ES interpretation), and*
   (c) $(s, \zeta)\downarrow$ *in* $\mathcal{T}(M)$ *iff* $s\downarrow$ *and* $\zeta = \varepsilon$ *(the FSES interpretation).*

This definition now gives us the notions of pushdown language and pushdown process: a *pushdown language* is the language of the transition system associated with a pushdown automaton, and a *pushdown process* is a branching bisimilarity class of labeled transition systems containing a labeled transition system associated with a pushdown automaton.

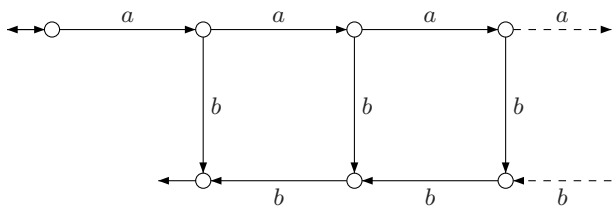

Fig. 8. A pushdown process.

The labeled transition system in Figure 8 is the labeled transition system associated with the pushdown automaton of Figure 7 according to the FSES or ES interpretations. To obtain the labeled transition system associated to the pushdown automaton in Figure 7 according to the FS interpretation, all states should be made final.

The following standard results pertaining to pushdown automata are often presented in textbooks on the theory of automata and formal languages:

1. Every language accepted by a pushdown automaton under the acceptance by empty stack interpretation is also accepted by a pushdown automaton under the acceptance by final state interpretation, and vice versa.
2. Every language accepted by a pushdown automaton is accepted by a pushdown automaton that only has *push transitions* (i.e., transitions of the form $s \xrightarrow{\varepsilon,a,d} t$ or $s \xrightarrow{d,a,ed}$) and *pop transitions* (i.e., transitions of the form $s \xrightarrow{d,a,\varepsilon} t$).
3. Every language accepted by a pushdown automaton is also generated by a context-free grammar, and every language generated by a context-free grammar is accepted by a pushdown automaton.

Only push and pop transitions It is easy to see that limiting the set of transitions to push and pop transitions only in the definition of pushdown automaton yields the same notion of pushdown process:

1. Eliminate a transition of the form $s \xrightarrow{\varepsilon,a,\varepsilon} t$ by adding a fresh state s', replacing the transition by the sequence of transitions $s \xrightarrow{\varepsilon,a,d} s' \xrightarrow{d,\tau,\varepsilon} t$ (with d just some arbitrary element in $\mathcal{D}'$).
2. Eliminate a transition of the form $s \xrightarrow{\varepsilon,a,\delta} t$, with $\delta = e_n \cdots e_1$ $(n \geq 1)$, by adding new states $s_2, \ldots, s_n$ and replacing the transition $s \xrightarrow{\varepsilon,a,\delta} t$ by the sequence of transitions
$$s \xrightarrow{\varepsilon,a,e_1} s_2 \xrightarrow{e_1,\tau,e_2e_1} \cdots \xrightarrow{e_{n-2},\tau,e_{n-1}e_{n-2}\cdots e_1} s_n \xrightarrow{e_{n-1},\tau,e_ne_{n-1}\cdots e_1} t \ .$$
3. Eliminate a transition of the form $s \xrightarrow{d,a,\delta} t$, with $\delta = e_n \cdots e_1$ $(n \geq 0)$, by adding new states $s_1, \ldots, s_n$ and replacing the transition $s \xrightarrow{d,a,\delta} t$ by transitions $s \xrightarrow{d,a,\varepsilon} s_1$, $s_1 \xrightarrow{\varepsilon,\tau,e_1} s_2$ and $s_1 \xrightarrow{f,\tau,e_1f} s_2$ for all $f \in \mathcal{D}'$, and the sequence of transitions
$$s_2 \xrightarrow{e_1,\tau,e_2e_1} \cdots \xrightarrow{e_{n-2},\tau,e_{n-1}e_{n-2}\cdots e_1} s_n \xrightarrow{e_{n-1},\tau,e_ne_{n-1}\cdots e_1} t \ .$$

Branching degree In [14] the structure of the labeled transition systems associated with pushdown automata was intensively studied (without termination conditions). In particular, they show these labeled transition systems have bounded branching. However, the pushdown processes that are generated modulo branching bisimulation may still exhibit infinite branching. See for example the pushdown automaton in Figure 9 that generates the pushdown process of Figure 6.

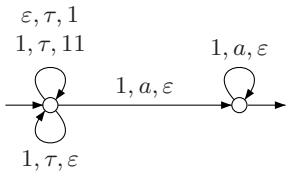

Fig. 9. Pushdown automaton that generates an infinitely branching pushdown process.

Nevertheless, we conjecture that whenever a pushdown process has finite branching, it has bounded branching. More precisely:

Conjecture 1. In every pushdown process there is a bound on the branching degree of those states that have finite branching.

Termination conditions Recall that from a language-theoretic point of view the different approaches to termination of pushdown automata (FS, ES, FSES) are all equivalent, but not from a process-theoretic point of view. First, we argue that the ES and FSES interpretations lead to the same notion of pushdown process.

Theorem 2. *A process is a pushdown process according to the ES interpretation if, and only if, it is a pushdown process according to the FSES interpretation.*

Proof. On the one hand, to see that a pushdown process according to the ES interpretation is also a pushdown process according to the FSES interpretation, let L be the labeled transition system associated with a pushdown automaton M under the ES interpretation, and let M' be the pushdown automaton obtained from M by declaring all states to be final. Then L is the labeled transition system associated with M' under the FSES interpretation.

On the other hand, to see that a pushdown process according to the FSES interpretation is also a pushdown process according to the ES interpretation, let $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ be an arbitrary pushdown automaton. We shall modify M such that the labeled transition system associated with the modified pushdown automaton under the ES interpretation is branching bisimilar to the labeled transition system associated with M under the FSES interpretation. We define the modified pushdown automaton $M' = (\mathcal{S}', \mathcal{A}', \mathcal{D}' \uplus \{\emptyset\}, \rightarrow', \uparrow', \emptyset)$ as follows:

1. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by adding a fresh initial state $\uparrow'$, and also a fresh state $s^{\downarrow}$ for every final state $s \in \downarrow$;
2. $\rightarrow'$ is obtained from $\rightarrow$ by
   (a) adding a transition $(\uparrow', \varepsilon, \tau, \emptyset, \uparrow)$ (the datum $\emptyset$, which is assumed not to occur in M, is used to mark the end of the stack),
   (b) replacing all transitions $(s, \varepsilon, a, \delta, t) \in \rightarrow$ by $(s, \emptyset, a, \delta\emptyset, t) \in \rightarrow'$, and
   (c) adding transitions $(s, \emptyset, \tau, \varepsilon, s^{\downarrow})$ and $(s^{\downarrow}, \varepsilon, \tau, \emptyset, s)$ for every $s \in \downarrow$.

We leave it to the reader to verify that the relation

$$\mathcal{R} = \{((s, \delta), (s, \delta\emptyset)) \mid s \in \mathcal{S} \;\&\; \delta \in \mathcal{D}'^{*}\} \cup \{((\uparrow, \varepsilon), (\uparrow', \varepsilon))\} \cup \{((s, \varepsilon), (s^{\downarrow}, \varepsilon)) \mid s \in \downarrow\}$$

is a branching bisimulation from the labeled transition associated with M under the ES interpretation to M' under the FSES interpretation. □

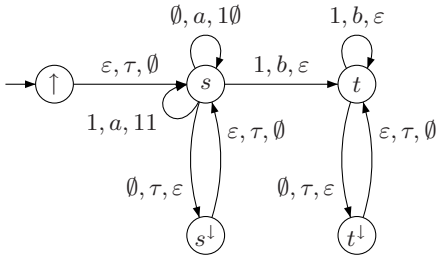

Fig. 10. Example pushdown automaton accepting on empty stack.

If we apply this modification on the pushdown automaton in Figure 7, then we get the result shown in Figure 10 where the states $\uparrow, s^{\downarrow}, t^{\downarrow}$ are added and five transitions, $\uparrow \xrightarrow{\varepsilon,\tau,\emptyset} s$ to put the end-of-stack marker on the stack, $s \xrightarrow{\emptyset,\tau,\varepsilon} s^{\downarrow}$ and $t \xrightarrow{\emptyset,\tau,\varepsilon} t^{\downarrow}$ to remove this marker when in the FSES case termination could occur, and $s^{\downarrow} \xrightarrow{\varepsilon,\tau,\emptyset} s$ and $t^{\downarrow} \xrightarrow{\varepsilon,\tau,\emptyset} t$ to put the end-of-stack marker back.

We proceed to argue that a pushdown process according to the ES interpretation is also a pushdown process according to the FS interpretation, but not vice versa. The classical proof (see, e.g., [11]) that a pushdown language according to the "acceptance by final state" approach is also a pushdown language according to the "acceptance by empty stack" approach coincide employs τ-transitions in a way that is valid modulo language equivalence, but not modulo branching bisimilarity. For instance, the construction that modifies a pushdown automaton M into another pushdown automaton M' such that the language accepted by M by final state is accepted by M' by empty stack adds τ-transitions from every final state of M to a fresh state in M' in which the stack is emptied. The τ-transition introduces, in M', a choice between the original outgoing transitions of the final state in M and termination by going to the fresh state; this choice was not necessarily present in M, and therefore the labeled transition systems associated with M and M' may not be branching bisimilar.

Theorem 3. *A process is a pushdown process according to the ES interpretation only if it is a pushdown process according to the FS interpretation, but not vice versa.*

Proof. On the one hand, to see that a pushdown process according to the ES interpretation is also a pushdown process according to the FS interpretation, let $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ an arbitrary pushdown automaton. We shall modify M such that the labeled transition system associated with the modified pushdown automaton under the ES interpretation is branching bisimilar to the labeled transition system associated with M under the FS interpretation. We define the modified pushdown automaton $M' = (\mathcal{S}', \mathcal{A}', \mathcal{D}' \uplus \{\emptyset\}, \rightarrow', \uparrow', \downarrow')$ as follows:

1. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by adding a fresh initial state $\uparrow'$, and also a fresh state $s^{\downarrow}$ for every state $s \in \mathcal{S}$;
2. $\downarrow'$ is the set $\{s^{\downarrow} \mid s \in \mathcal{S}\}$ of all these newly added states;
3. $\rightarrow'$ is obtained from $\rightarrow$ by
   (a) adding a transition $(\uparrow', \varepsilon, \tau, \emptyset, \uparrow)$ (the datum $\emptyset$, which is assumed not to occur in M, is used to mark the end of the stack),
   (b) replacing all transitions $(s, \varepsilon, a, \delta, t) \in \rightarrow$ by $(s, \emptyset, a, \delta\emptyset, t) \in \rightarrow'$, and
   (c) adding transitions $(s, \emptyset, \tau, \varepsilon, s^{\downarrow})$ and $(s^{\downarrow}, \varepsilon, \tau, \emptyset, s)$ for every $s \in \mathcal{S}$.

We leave it to the reader to verify that the relation

$$\mathcal{R} = \{((s,\delta),(s,\delta\emptyset)) \mid s \in \mathcal{S} \;\&\; \delta \in \mathcal{D}'^{*}\} \cup \{((\uparrow,\varepsilon),(\uparrow',\varepsilon))\} \cup \{((s,\varepsilon),(s^{\downarrow},\varepsilon)) \mid s \in \mathcal{S}\}$$

is a branching bisimulation from the labeled transition associated with M under the ES interpretation to M' under the FS interpretation.

On the other hand, there exist pushdown processes according to the FS interpretation for which there is no equivalent pushdown process according to the ES interpretation. An example is the pushdown automaton shown in Figure 11.

The labeled transition system associated with it according to the FS interpretation is depicted in Figure 12; it has infinitely many terminating configurations. Moreover, no pair of these configurations is branching bisimilar, which can be seen by noting that the nth state from the left can perform at most $n-1$ times a b-transition before it has to perform an a-transition again.

In contrast with this, note that the labeled transition system associated with any pushdown automaton according to the ES interpretation necessarily has finitely many terminating configurations, for the pushdown automaton has only finitely many states and the stack is required to be empty. □

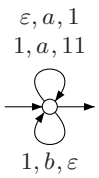

Fig. 11. The counter pushdown automaton.

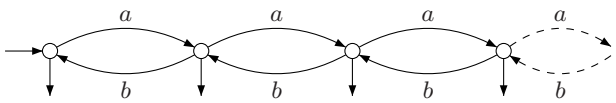

Fig. 12. Labeled transition system associated with automaton of Figure 11 according to the FS interpretation.

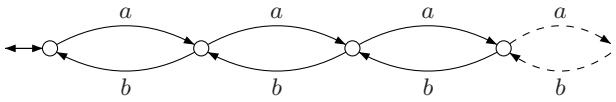

Fig. 13. Labeled transition system associated with automaton of Figure 11 according to the FSES (or ES) interpretation.

Context-free specifications We shall now consider the process-theoretic version of the standard result in the theory of automata and formal languages that the set of pushdown languages coincides with the set of languages generated by context-free grammars. As the process-theoretic counterparts of context-free grammars we shall consider recursive specifications in the subtheory TSP_τ of TCP_τ, which is obtained from BSP_τ by adding sequential composition $\_\cdot\_$. So a TSP_τ recursive specification over a finite subset $\mathcal{N}'$ of $\mathcal{N}$ is a recursive specification over $\mathcal{N}'$ in which only the constructions $\mathbf{0}$, $\mathbf{1}$, N ($N \in \mathcal{N}'$), $a.\_$ ($a \in \mathcal{A}_\tau$), $\_\cdot\_$ and $\_+\_$ occur.

TSP_τ recursive specifications can be used to specify pushdown processes. To give an example, the process expression X defined in the TSP_τ recursive specification

$$X \stackrel{\mathrm{def}}{=} \mathbf{1} + a.X \cdot b.\mathbf{1}$$

specifies the labeled transition system in Figure 13, which is associated with the pushdown automaton in Figure 11 under the FSES interpretation.

Next, we will show by contradiction that the FS interpretation of this pushdown automaton (see Figure 12) cannot be given by a TSP_τ recursive specification. Recall that under this interpretation, there are infinitely many distinct states in this pushdown process and all these states are terminating. This implies that all variables in a possible TSP_τ recursive specification for this process would have a $\mathbf{1}$-summand to ensure termination in all states. On the other hand, we discuss further on in this paper that any state of a TSP_τ recursive specification can be represented by a sequential composition of variables using the Greibach normal form. Each variable in this normal form must be terminating, since all states are terminating, and each variable can do a bounded number of b-transitions without performing a-transitions in between. To get sequences of b-transitions of arbitrary length, variables are sequentially composed. However, since all variables are also terminating this would result in the possibility to skip parts of the intended sequence of b-transitions and hence lead to branching. This branching is not present in the process in Figure 12, hence this process cannot be represented by a TSP_τ recursive specification. Since this impossibility already occurs for a very simple example such as a counter, we restrict ourselves to only use the FSES interpretation in the remainder of this paper.

That the notion of TSP_τ recursive specification still naturally corresponds with with the notion of context-free grammar is confirmed by the following theorem.

Theorem 4. *For every pushdown automaton M there exists a TSP_τ recursive specification E and process expression p such that $\mathcal{T}(M)$ and $\mathcal{T}_E(p)$ are language equivalent, and, vice versa, for every recursive specification E and process expression p there exists a pushdown automaton M such that $\mathcal{T}(M)$ and $\mathcal{T}_E(p)$ are language equivalent.*

We shall see below that a similar result with language equivalence replaced by branching bisimilarity does not hold. In fact, we shall see that there are pushdown processes that are not recursively definable in TSP_τ, and that there are also

TSP_τ recursive specifications that define non-pushdown processes can be defined. We shall present a restriction on pushdown automata and a restriction on TSP_τ recursive specifications that enable us to retrieve the desired equivalence: we shall prove that the set of so-called *popchoice-free* pushdown processes corresponds with the set of processes definable by a *transparency-restricted* TSP_τ recursive specification. We have not yet been able to establish that our result is optimal in the sense that a pushdown process is definable by a recursive TSP_τ specification only if it is popchoice-free, although we conjecture that this is the case.

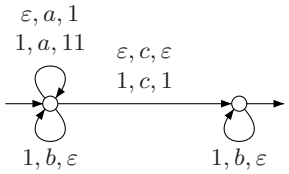

Fig. 14. Pushdown automaton that is not popchoice-free.

Consider the pushdown automaton in Figure 14, which generates the transition system shown in Figure 15. In [13], Moller proved that this transition system cannot be defined with a BPA recursive specification, where BPA is the subtheory of TSP_τ obtained by omitting the τ-prefix and the constant $\mathbf{0}$ and by disallowing $\mathbf{1}$ to occur as a summand in a nontrivial alternative composition. His proof can be modified to show that the transition system is not definable with a TSP_τ recursive specification either. We conclude that not every pushdown process is definable with a TSP_τ recursive specification.

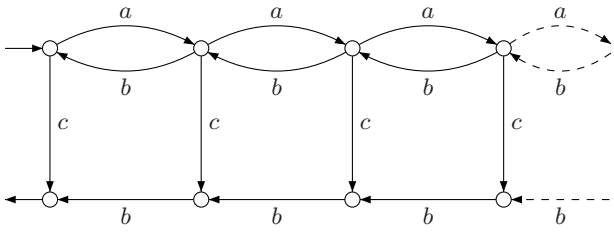

Fig. 15. Transition system of automaton of Figure 14.

Note that a push of a 1 onto the stack in the initial state of the pushdown automaton in Figure 14 can (on the way to termination) be popped again in the initial state or in the final state: the choice of where the pop will take place cannot be made at the time of the push. In other words, in the pushdown automaton in Figure 14 pop transitions may induce a choice in the associated transition system; we refer to such choice through a pop transition as a *popchoice*. We

shall prove below that by disallowing popchoice we define a class of pushdown processes that are definable with a TSP_τ recursive specification.

Definition 12. *Let M be a pushdown automaton that uses only push and pop transitions. A d-pop transition is a transition $s \xrightarrow{d,a,\varepsilon} t$, which pops a datum d. We say M is* popchoice-free *iff whenever there are two d-pop transitions $s \xrightarrow{d,a,\varepsilon} t$ and $s' \xrightarrow{d,b,\varepsilon} t'$, then $t = t'$. A pushdown process is* popchoice-free *if it contains a labeled transition system associated with a popchoice-free pushdown automaton.*

The definition of a pushdown automaton uses a stack as memory. The stack itself can be modeled as a pushdown process, in fact (as we will see shortly) it is the prototypical pushdown process. Given a finite set of data $\mathcal{D}'$, the stack has an input channel i over which it can receive elements of $\mathcal{D}'$ and an output channel o over which it can send elements of $\mathcal{D}'$. The stack process is given by a pushdown automaton with one state $\uparrow$ (which is both initial and final) and transitions $\uparrow \xrightarrow{\varepsilon,i?d,d} \uparrow$, $\uparrow \xrightarrow{e,i?d,de} \uparrow$, and $\uparrow \xrightarrow{d,o!d,\varepsilon} \uparrow$ for all $d, e \in \mathcal{D}'$. As this pushdown automaton has only one state, it is popchoice-free. The transition system of the stack in case $\mathcal{D}' = \{0, 1\}$ is presented in Figure 16. The following recursive specification defines a stack:

$$S \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}'} i?d.S \cdot o!d.S \ ; \tag{1}$$

we refer to this specification of a stack over $\mathcal{D}'$ as E_S.

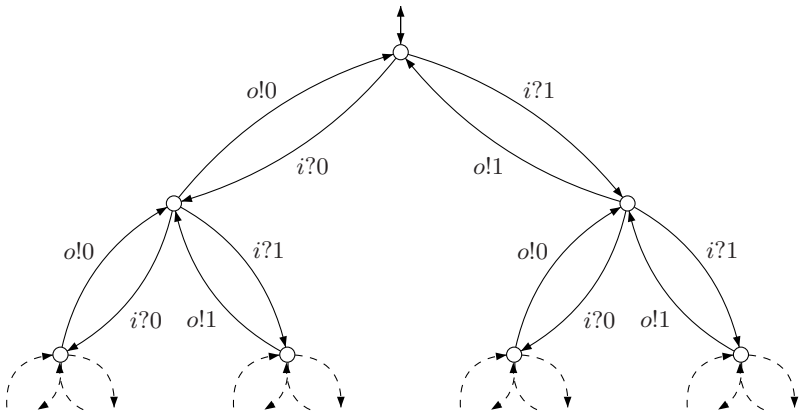

Fig. 16. Stack over $\mathcal{D}' = \{0, 1\}$.

The stack process can be used to make the interaction between control and memory in a pushdown automaton explicit [4]. This is illustrated by the following theorem, stating that every pushdown process is equal to a regular process interacting with a stack.

Theorem 5. *For every pushdown automaton M there exists a* BSP_τ *process expression p and a* BSP_τ *recursive specification E, and for every* BSP_τ *process expression p and* BSP_τ *recursive specification there exists a pushdown automaton M such that*

$$\mathcal{T}(M) \leftrightarrows_b \mathcal{T}_{E\cup E_S}(\tau_{i,o}(\partial_{i,o}(p \parallel S))) \ .$$

Proof. Let $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ be a pushdown automaton; we define the BSP_τ recursive specification E as follows:

- For each $s \in \mathcal{S}$ and $d \in \mathcal{D}' \uplus \{\emptyset\}$ it has a variable $V_{s,d}$ (where $\emptyset$ is a special symbol added to $\mathcal{D}'$ to denote that the stack is empty).
- For each pop transition $t \xrightarrow{d,a,\varepsilon} t$ the right-hand side of the defining equation for $V_{s,d}$ has a summand $a.\sum_{e\in\mathcal{D}'\cup\{\emptyset\}} o?e.V_{t,e}$.
- For each push transition $s \xrightarrow{d,a,ed} t$ the right-hand side of the defining equation for $V_{s,d}$ has a summand $a.i!d.V_{t,e}$, and for each push transition $s \xrightarrow{\varepsilon,a,e} t$ the right-hand side of the defining equation for $V_{s,\emptyset}$ has a summand $a.i!\emptyset.V_{t,e}$.
- For each $s \in \mathcal{S}$ such that $s\downarrow$ the right-hand side of the defining equation for $V_{s,\emptyset}$ has a $\mathbf{1}$-summand.

We present some observations from which it is fairly straightforward to establish that $\mathcal{T}(M) \leftrightarrows_b \mathcal{T}_{E\cup E_S}(\tau_{i,o}(\partial_{i,o}(V_{\uparrow,\emptyset} \parallel S)))$. In our proof we abbreviate the process expression $S\cdot i!d_n.S\cdots i!d_1.S$ by $S_{d_n\cdots d_1}$, with, in particular, $S_\varepsilon = S$.

First, note that whenever $\mathcal{T}(M)$ has a transition $(s,d) \xrightarrow{a} (t,\varepsilon)$, then

$$\partial_{i,o}(V_{s,d} \parallel S_\emptyset) \xrightarrow{a} \partial_{i,o}((\sum_{e\in\mathcal{D}'\uplus\{\emptyset\}} o!e.V_{t,e}) \parallel S_\emptyset) \xrightarrow{o?\emptyset} \partial_{i,o}(V_{t,\emptyset} \parallel S) \ .$$

The abstraction operator $\tau_{i,o}(\_)$ will rename the transition labeled $o?\emptyset$ into a τ-transition. This τ-transition is *inert* in the sense that it does not preclude any observable behavior that was possible before the τ-transition. It is well-known that such inert τ-transitions can be omitted while preserving branching bisimilarity.

Second, note that whenever $\mathcal{T}(M)$ has a transition $(s,d\zeta) \xrightarrow{a} (t,\zeta)$ with ζ nonempty, say $\zeta = e\zeta'$, then

$$\partial_{i,o}(V_{s,d} \parallel S_\zeta) \xrightarrow{a} \xrightarrow{o?e} \partial_{i,o}(V_{t,e} \parallel S_{\zeta'}) \ ,$$

and, since the second transition is the only step possible after the first a-transition, the τ-transition resulting from applying $\tau_{i,o}(\_)$ is again inert.

Third, note that whenever $\mathcal{T}(M)$ has a transition $(s,d\zeta) \xrightarrow{a} (t,ed\zeta)$, then

$$\partial_{i,o}(V_{s,d} \parallel S_\zeta) \xrightarrow{a} \xrightarrow{i?d} \partial_{i,o}(V_{t,e} \parallel S_{d\zeta}) \ ,$$

and again the τ-transition resulting from applying $\tau_{i,o}(\_)$ is inert.

Finally, note that whenever $\mathcal{T}(M)$ has a transition $(s,\varepsilon) \xrightarrow{a} (t,e)$, then

$$\partial_{i,o}(V_{s,\emptyset} \parallel S) \xrightarrow{a} \xrightarrow{i?\emptyset} \partial_{i,o}(V_{t,e} \parallel S_\emptyset) \ .$$

Conversely, let E be a BSP_τ recursive specification, let p be a BSP_τ process expression, and let $M = (\mathcal{S}, \mathcal{A}', \rightarrow, \uparrow, \downarrow)$ be the associated labeled transition system. We define a pushdown automaton M as follows:

- The set of states, the action alphabet, and the initial and final states are the same as those of the finite automaton.
- The data alphabet is the set of data $\mathcal{D}'$ of the presupposed recursive specification of a stack.
- Whenever $s \xrightarrow{a} t$ in M, and $a \neq i!d, o?d$ $(d \in \mathcal{D}')$, then $s \xrightarrow{d,a,d} t$;
- Whenever $s \xrightarrow{i!d} t$ in M, then $s \xrightarrow{\varepsilon,\tau,d} t$ and $s \xrightarrow{e,\tau,de} t$ for all $e \in \mathcal{D}'$.
- Whenever $s \xrightarrow{o?d} t$ in M, then $s \xrightarrow{d,\tau,\varepsilon} t$.

We omit the proof that every transition of $\mathcal{T}_{E \cup E_S}(\tau_{i,o}(\partial_{i,o}(V_{\uparrow,\emptyset} \parallel S)))$ can be matched by a transition in $\mathcal{T}(M)$ in the sense required by the definition of branching bisimilarity. □

In process theory it is standard practice to restrict attention to *guarded recursive specifications*. Roughly, a TSP_τ recursive specification is *guarded* if every occurrence of a name occurs in the argument of an action prefix $a.\_$ $(a \in \mathcal{A})$. For a precise definition of guardedness we refer to [1].

Every guarded recursive specification over TSP_τ can be brought into *restricted Greibach normal form*, that is, satisfying the requirement that every right-hand side of an equation only has summands that are $\mathbf{1}$ or of the form $a.\xi$, where $a \in \mathcal{A}_\tau$ and $\xi = \mathbf{1}$, or ξ is a name, or ξ is a sequential composition of two names. A convenient property of recursive specification in restricted Greibach normal form is that every reachable state in the labeled transition system associated with a name N in such a recursive specification will be denoted by a (generalized) sequential composition of names (see, e.g., the labeled transition system in Figure 17).

Let p be a TSP_τ process expression in the context of a guarded recursive specification E. Then the associated labeled transition system $\mathcal{T}_E(p)$ has finite branching (see, e.g., [1] for a proof). It follows that, e.g., the labeled transition system in Figure 6 is not definable by a guarded recursive specification in restricted Greibach normal form. It is possible with the following unguarded specification:

$$X \stackrel{\mathrm{def}}{=} \mathbf{1} + X \cdot a.\mathbf{1} \,. \tag{2}$$

This should be contrasted with a standard result in the theory of automata and formal languages that, after translation to our process-theoretic setting, states that even if E is *not* guarded, then still there exists a guarded recursive specification E' in Greibach normal such that $\mathcal{T}_E(p)$ and $\mathcal{T}_{E'}(p)$ are language equivalent.

In this paper we choose to follow the standard practice of using guarded recursive specifications, even though this means that we cannot find a complete correspondence with respect to infinite branching pushdown processes. We leave the generalization of our results to an unguarded setting as future work.

Still, restricting to guarded recursive specifications in restricted Greibach normal form is not sufficient to get the desired correspondence between processes definable by TSP_τ recursive specifications and processes definable as a popchoice-free pushdown automaton. Consider the following guarded recursive specification, which is in restricted Greibach normal form:

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

The labeled transition system associated with X, which is depicted in Figure 17, has unbounded branching. So, according to our conjecture, cannot be a pushdown process.

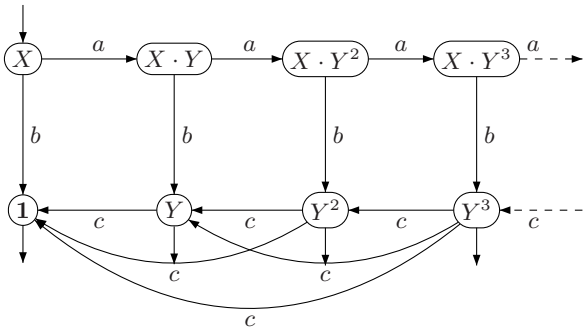

Fig. 17. Process with unbounded branching.

Note that the unbounded branching is due to the $\mathbf{1}$-summand in the defining equation for Y by which $Y^n \stackrel{c}{\longrightarrow} Y^m$ for all $m < n$. A name N in a recursive specification is called *transparent* if its defining equation has a $\mathbf{1}$-summand; otherwise it is called *opaque*. To exclude recursive specifications generating labeled transition systems with unbounded branching, we will require that transparent names may only occur as the *last* element of reachable sequential compositions of names.

Definition 13 (Transparency restricted). *Let E be a recursive specification over* TSP_τ *in restricted Greibach normal form. We call such a specification* transparency-restricted *if for all (generalized) sequential compositions of names ξ reachable from a name in E it holds that all but the last name in ξ is opaque.*

As an example, note that the specification of the stack over $\mathcal{D}'$ defined in (1) above is not transparency restricted, because it is not in Greibach normal form. But the same process can be defined with a transparency-restricted recursive specification: it suffices to add, for all $d \in \mathcal{D}'$, a name T_d to replace $S \cdot o!d.\mathbf{1}$. Thus we obtain the following transparency-restricted specification of the stack

over $\mathcal{D}'$:

$$S \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}'} i?d.T_d \cdot S \ ,$$

$$T_d \stackrel{\text{def}}{=} o!d.\mathbf{1} + \sum_{e \in \mathcal{D}'} i?e.T_e \cdot T_d \ .$$

It can easily be seen that the labeled transition system associated with a name in a transparency-restricted specification has bounded branching: the branching degree of a state denoted by a reachable sequential composition of names is equal to the branching degree of its first name, and the branching degree of a name is bounded by the number of summands of the right-hand side of its defining equation. Since $\mathbf{1}$-summands can be eliminated modulo language equivalence (according to the procedure for eliminating λ- or ϵ-productions from context-free grammars), there exists, for *every* TSP_τ recursive specification E a transparency-restricted specification E' such that $\mathcal{T}_E(p)$ and $\mathcal{T}_{E'}(p)$ are *language equivalent* (with p an arbitrary process expression in the context of E).

For investigations under what circumstances we can extend the set of pushdown processes to incorporate processes with unbounded branching, see [4]. In this paper a (partially) forgetful stack is used to deal with transparent variables on the stack. However, if we allow for τ-transitions in the recursive specifications, we can use the stack as is presented above. Note also that the paper does not require the recursive specifications to be transparency-restricted, but this comes at the cost of using a weaker equivalence (namely contrasimulation [9] instead of branching bisimulation) in some cases.

We are now in a position to establish a process-theoretic counterpart of the correspondence between pushdown automata and context-free grammars.

Theorem 6. *A process is a popchoice-free pushdown process if, and only if, it is definable by a transparency-restricted recursive specification.*

Proof. For the implication from right to left, let E be a transparency-restricted recursive specification, and let I be a name in E. We define a pushdown automaton $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ as follows:

1. The set $\mathcal{S}$ consists of the names occurring in E, the symbol $\mathbf{1}$, an extra initial state $\uparrow$, and an extra intermediate state t.
2. The set $\mathcal{A}'$ consists of all the actions occurring in E.
3. The set $\mathcal{D}'$ consists of the names occurring in E and the symbol $\mathbf{1}$.
4. The transition relation $\rightarrow$ is defined as follows:
   (a) there is a transition $\uparrow \xrightarrow{\varepsilon,\tau,\mathbf{1}} I$;
   (b) if the right-hand side of the defining equation for a name N has a summand $a.\mathbf{1}$, then $\rightarrow$ has transitions $N \xrightarrow{\mathbf{1},a,\varepsilon} \mathbf{1}$ and $N \xrightarrow{N',a,\varepsilon} N'$,
   (c) if the right-hand side of the defining equation for a name N has a summand $a.N'$, then there are transitions $N \xrightarrow{d,a,N'd} t$ and $t \xrightarrow{N',\tau,\varepsilon} N'$ $(d \in \mathcal{D}')$, and

   (d) if the right-hand side of the defining equation for a name N has a summand $a.N' \cdot N''$, then there are transitions $N \xrightarrow{d,a,N''d} N'$ $(d \in \mathcal{D}')$.
5. The set of final states $\downarrow$ consists of $\mathbf{1}$ and all variables with a $\mathbf{1}$-summand.

We leave it to the reader to check that $\mathcal{T}_E(I) \leftrightarrows_{\mathrm{b}} \mathcal{T}(M)$. Using the procedure described earlier in this section, the set of transitions can be limited to include push and pop transitions only. The pushdown automaton resulting from the procedure is popchoice-free, for an N-pop transition leads to state N.

The proof of the implication from left to right is an adaptation of the classical proof that associates a context-free grammar with a given pushdown automaton. Let $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ be a popchoice-free pushdown automaton. We define a transparency-restricted specification E with for every state $s \in \mathcal{S}$ a name $N_{s\varepsilon}$ and for every state s a name N_{sdt} if M has transitions that pop datum d leading to the state t. The defining equations in E for these names satisfy the following:

1. The right-hand side of the defining equation for $N_{s\varepsilon}$ has
   (a) a summand $\mathbf{1}$ if, and only if, $s\downarrow$, and
   (b) a summand $a.N_{tdw} \cdot N_{w\varepsilon}$ whenever $s \xrightarrow{\varepsilon,a,d} t$ and all d-pop transitions lead to w.
2. $N_{s\varepsilon} \stackrel{\mathrm{def}}{=} \mathbf{0}$ if $N_{s\varepsilon}$ has no other summands.
3. The right-hand side of the defining equation for N_{sdt} has
   (a) a summand $a.\mathbf{1}$ if, and only if, $s \xrightarrow{d,a,\varepsilon} t$, and
   (b) a summand $a.N_{uew} \cdot N_{wdt}$ whenever $s \xrightarrow{d,a,ed} u$ and all e-pop transitions lead to state w.
4. $N_{sdt} \stackrel{\mathrm{def}}{=} \mathbf{0}$ if N_{sdt} has no other summands.

It is easy to see that the resulting specification is transparency-restricted, and that $\mathcal{T}_E(N_{\uparrow\varepsilon}) \leftrightarrows_{\mathrm{b}} \mathcal{T}(M)$. □

Consider the pushdown automaton shown in Figure 7. It is easy to see that this pushdown automaton is popchoice-free, since both 1-pop transitions lead to the same state t. Using the method described in the proof of Theorem 6 we can now give the following recursive specification over TSP_τ:

$$
\begin{aligned}
N_{s\varepsilon} &\stackrel{\mathrm{def}}{=} \mathbf{1} + a.N_{s1t} \cdot N_{t\varepsilon} \ , \\
N_{t\varepsilon} &\stackrel{\mathrm{def}}{=} \mathbf{1} \ , \\
N_{s1t} &\stackrel{\mathrm{def}}{=} b.\mathbf{1} + a.N_{s1t} \cdot N_{t1t} \ , \\
N_{t1t} &\stackrel{\mathrm{def}}{=} b.\mathbf{1} \ .
\end{aligned}
$$

We can reduce this specification by removing occurrences of $N_{t\varepsilon}$ (for the right-hand side of the defining equation of this name is just $\mathbf{1}$) and substituting occurrences of N_{t1t} by $b.\mathbf{1}$. We get

$$
\begin{aligned}
N_{s\varepsilon} &\stackrel{\mathrm{def}}{=} \mathbf{1} + a.N_{s1t} \ , \\
N_{s1t} &\stackrel{\mathrm{def}}{=} b.\mathbf{1} + a.N_{s1t} \cdot b.\mathbf{1} \ .
\end{aligned}
$$

Now, we see that $N_{s1t} = (\mathbf{1} + a.N_{s1t}) \cdot b.\mathbf{1} = N_{s\varepsilon} \cdot b.\mathbf{1}$ and therefore we have that $N_{s\varepsilon} \stackrel{\text{def}}{=} \mathbf{1} + a.N_{s\varepsilon} \cdot b.\mathbf{1}$ which is equal to the specification we gave before.

Thus, we have established a correspondence between a popchoice-free pushdown processes on the one hand, and transparency-restricted recursive specification over TSP_τ on the other hand, thereby casting the classical result of the equivalence of pushdown automata and context-free grammars in terms of processes and bisimulation.

5 Computable processes

We proceed to give a definition of a Turing machine that we can use to generate a transition system. The classical definition of a Turing machine uses the memory tape to hold the input string at start up. We cannot use this simplifying trick, as we do not want to fix the input string beforehand, but want to be able to input symbols one symbol at a time. Therefore, we make an adaptation of a so-called *off-line* Turing machine, which starts out with an empty memory tape, and can take an input symbol one at a time. Another important consideration is that we allow termination only when the tape is empty again and we are in a final state: this is like the situation we had for the pushdown automaton.

Definition 14 (Turing machine). *A* Turing machine *M is defined as a six-tuple $(\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ where:*

1. *$\mathcal{S}$ is a finite set of states,*
2. *$\mathcal{A}'$ is a finite subset of $\mathcal{A}$,*
3. *$\mathcal{D}'$ is a finite subset of $\mathcal{D}$,*
4. *$\rightarrow \subseteq \mathcal{S} \times (\mathcal{D}' \cup \{\varepsilon\}) \times (\mathcal{A}' \cup \{\tau\}) \times (\mathcal{D}' \cup \{\varepsilon\}) \times \{L, R\} \times \mathcal{S}$ is a finite set of* transitions *or* steps*,*
5. *$\uparrow \in \mathcal{S}$ is the initial state,*
6. *$\downarrow \subseteq \mathcal{S}$ is the set of final states.*

If $(s, d, a, e, M, t) \in \rightarrow$, we write $s \xrightarrow{d,a,e,M} t$, and this means that the machine, when it is in state s and reading symbol d on the tape, will execute input action a, change the symbol on the tape to e, will move one step left if $M = L$ and right if $M = R$ and thereby move to state t. It is also possible that d and/or e is ε: if d is ε, we are looking at an empty part of the tape, but, if the tape is nonempty, then there is a symbol immediately to the right or to the left; if e is ε, then a symbol will be erased, but this can only happen at an end of the memory string. The exact definitions are given below.

At the start of a Turing machine computation, we will assume the Turing machine is in the initial state, and that the memory tape is empty (denoted by $\Box$).

By looking at all possible executions, we can define the transition system of a Turing machine. Also Caucal [6] defines the transition system of a Turing machine in this way, but he considers transition systems modulo isomorphism, and leaves out all internal τ-moves.

Definition 15. *Let $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ be a Turing machine. The* labeled transition system *of M is defined as follows:*

1. *The set of states is $\{(s, \bar{\Box}) \mid s \in \mathcal{S}\} \cup \{(s, \Box\delta\Box) \mid s \in \mathcal{S}, \delta \in \mathcal{D}'^* - \{\varepsilon\}\}$, where in the second component there is an overbar on one of the elements of the string $\Box\delta\Box$ denoting the contents of the memory tape and the present location. The box indicates a blank portion of the tape.*
2. *A symbol can be replaced by another symbol if the present location is not a blank. Moving right, there are two cases: there is another symbol to the right or there is a blank to the right.*
   - $(s, \Box\delta\bar{d}\Box) \xrightarrow{a} (t, \Box\delta e\bar{\Box})$ *iff* $s \xrightarrow{d,a,e,R} t$ $(d, e \in \mathcal{D}', \delta \in \mathcal{D}'^*)$,
   - $(s, \Box\delta\bar{d}f\zeta\Box) \xrightarrow{a} (t, \Box\delta e\bar{f}\zeta\Box)$ *iff* $s \xrightarrow{d,a,e,R} t$, *for all* $d, e \in \mathcal{D}', \delta, \zeta \in \mathcal{D}'^*$.

   Similarly, there are two cases for a move left.
   - $(s, \Box\bar{d}\delta\Box) \xrightarrow{a} (t, \bar{\Box}e\delta\Box)$ *iff* $s \xrightarrow{d,a,e,L} t$ $(d, e \in \mathcal{D}', \delta \in \mathcal{D}'^*)$,
   - $(s, \Box\delta f\bar{d}\zeta\Box) \xrightarrow{a} (t, \Box\delta\bar{f}e\zeta\Box)$ *iff* $s \xrightarrow{d,a,e,L} t$, *for all* $d, e \in \mathcal{D}', \delta, \zeta \in \mathcal{D}'^*$.
3. *To erase a symbol, it must be at the end of the string. For a move right, there are three cases.*
   - $(s, \Box\bar{d}\Box) \xrightarrow{a} (t, \bar{\Box})$ *iff* $s \xrightarrow{d,a,\varepsilon,R} t$ $(d \in \mathcal{D}')$,
   - $(s, \Box\delta\bar{d}\Box) \xrightarrow{a} (t, \Box\delta\bar{\Box})$ *iff* $s \xrightarrow{d,a,\varepsilon,R} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^* - \{\varepsilon\})$,
   - $(s, \Box\bar{d}f\delta\Box) \xrightarrow{a} (t, \Box\bar{f}\delta\Box)$ *iff* $s \xrightarrow{d,a,\varepsilon,R} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^*)$.

   Similarly for a move left.
   - $(s, \Box\bar{d}\Box) \xrightarrow{a} (t, \bar{\Box})$ *iff* $s \xrightarrow{d,a,\varepsilon,L} t$ $(d \in \mathcal{D}')$,
   - $(s, \Box\bar{d}\delta\Box) \xrightarrow{a} (t, \bar{\Box}\delta\Box)$ *iff* $s \xrightarrow{d,a,\varepsilon,L} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^* - \{\varepsilon\})$,
   - $(s, \Box\delta f\bar{d}\Box) \xrightarrow{a} (t, \Box\delta\bar{f}\Box)$ *iff* $s \xrightarrow{d,a,\varepsilon,L} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^*)$.
4. *To insert a new symbol, we must be looking at a blank. We can only move right, if we are to the left of a (possible) data string. This means there are two cases for a move right.*
   - $(s, \bar{\Box}) \xrightarrow{a} (t, \Box d\bar{\Box})$ *iff* $s \xrightarrow{\varepsilon,a,d,R} t$ $(d \in \mathcal{D}')$,
   - $(s, \bar{\Box}f\delta\Box) \xrightarrow{a} (t, \Box d\bar{f}\delta\Box)$ *iff* $s \xrightarrow{\varepsilon,a,d,R} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^*)$.

   Similarly for a move left.
   - $(s, \bar{\Box}) \xrightarrow{a} (t, \bar{\Box}d\Box)$ *iff* $s \xrightarrow{\varepsilon,a,d,L} t$ $(d \in \mathcal{D}')$,
   - $(s, \Box\delta f\bar{\Box}) \xrightarrow{a} (t, \Box\delta\bar{f}\Box)$ *iff* $s \xrightarrow{\varepsilon,a,d,L} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^*)$.
5. *Finally, looking at a blank, we can keep it a blank. Two cases for a move right.*
   - $(s, \bar{\Box}) \xrightarrow{a} (t, \bar{\Box})$ *iff* $s \xrightarrow{\varepsilon,a,\varepsilon,R} t$,
   - $(s, \bar{\Box}f\delta\Box) \xrightarrow{a} (t, \Box\bar{f}\delta\Box)$ *iff* $s \xrightarrow{\varepsilon,a,\varepsilon,R} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^*)$.

   Similarly for a move left.
   - $(s, \bar{\Box}) \xrightarrow{a} (t, \bar{\Box})$ *iff* $s \xrightarrow{\varepsilon,a,\varepsilon,L} t$,
   - $(s, \Box\delta f\bar{\Box}) \xrightarrow{a} (t, \Box\delta\bar{f}\Box)$ *iff* $s \xrightarrow{\varepsilon,a,\varepsilon,L} t$ $(d \in \mathcal{D}', \delta \in \mathcal{D}'^*)$.
6. *The initial state is $(\uparrow, \bar{\Box})$;*
7. *$(s, \bar{\Box}) \downarrow$ iff $s \downarrow$.*

Now we define a *computable process* as the branching bisimulation equivalence class of a transition system of a Turing machine.

In order to make the internal communications of a Turing machine explicit, we need now *two* stacks, one on the left containing the contents of the memory tape to the left of the current symbol and one on the right containing the contents of the memory tape to the right of the current symbol:

$$S^l \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d\in\mathcal{D}} li?d.S^l \cdot lo!d.S^l \ ,$$
$$S^r \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d\in\mathcal{D}} ri?d.S^r \cdot ro!d.S^r \ .$$

Then, we get the following characterization of computable processes.

Theorem 7. *If process p is a computable process, then there is a regular process q with*

$$p \leftrightarrows_b \tau_{li,lo,ri,ro}(\partial_{li,lo,ri,ro}(q \parallel S^l \parallel S^r)) \ .$$

Proof. Suppose there is a Turing machine $M = (\mathcal{S}, \mathcal{A}', \mathcal{D}', \rightarrow, \uparrow, \downarrow)$ generating a transition system that is branching bisimilar to p. We proceed to define a BSP specification for the regular process q. This specification has variables $V_{s,d}$ for $s \in \mathcal{S}$ and $d \in \mathcal{D}' \cup \{\emptyset\}$. Moreover, there are variables $W_{s,\emptyset}$ denoting that the tape is empty on both sides.

1. The initial variable is $W_{\uparrow,\emptyset}$;
2. Whenever $s \xrightarrow{d,a,e,r} t$ $(d, e \in \mathcal{D}')$, variable $V_{s,d}$ has a summand
$$a.li!e. \sum_{f\in\mathcal{D}'\cup\{\emptyset\}} ro?f.V_{t,f}$$
3. Whenever $s \xrightarrow{d,a,e,L} t$ $(d, e \in \mathcal{D}')$, variable $V_{s,d}$ has a summand
$$a.ri!e. \sum_{f\in\mathcal{D}'\cup\{\emptyset\}} lo?f.V_{t,f}$$
4. Whenever $s \xrightarrow{d,a,\varepsilon,R} t$ $(d \in \mathcal{D}')$, variable $V_{s,d}$ has a summand
$$a.(ro?\emptyset.(lo?\emptyset.W_{t,\emptyset} + \sum_{f\in\mathcal{D}'} lo?f.li!f.ri!\emptyset.V_{t,\emptyset}) + \sum_{f\in\mathcal{D}'} ro?f.V_{t,f})$$
5. Whenever $s \xrightarrow{d,a,\varepsilon,L} t$ $(d \in \mathcal{D}')$, variable $V_{s,d}$ has a summand
$$a.(lo?\emptyset.(ro?\emptyset.W_{t,\emptyset} + \sum_{f\in\mathcal{D}'} ro?f.ri!f.li!\emptyset.V_{t,\emptyset}) + \sum_{f\in\mathcal{D}'} lo?f.V_{t,f})$$

6. Whenever $s \xrightarrow{\varepsilon,a,d,R} t$, variable $V_{s,\emptyset}$ has a summand

$$a.li!d.(ro?\emptyset.ri!\emptyset.V_{t,\emptyset} + \sum_{f\in\mathcal{D}'} ro?f.V_{t,f})$$

and variable $W_{s,\emptyset}$ has a summand $a.li!\emptyset.li!d.ri!\emptyset.V_{t,\emptyset}$;
7. Whenever $s \xrightarrow{\varepsilon,a,d,L} t$, variable $V_{s,\emptyset}$ has a summand

$$a.ri!d.(lo?\emptyset.li!\emptyset.V_{t,\emptyset} + \sum_{f\in\mathcal{D}'} lo?f.V_{t,f})$$

and variable $W_{s,\emptyset}$ has a summand $a.ri!\emptyset.ri!d.li!\emptyset.V_{t,\emptyset}$;
8. Whenever $s \xrightarrow{\varepsilon,a,\varepsilon,R} t$, variable $V_{s,\emptyset}$ has a summand

$$a.(ro?\emptyset.ri!\emptyset.V_{t,\emptyset} + \sum_{f\in\mathcal{D}'} ro?f.V_{t,f})$$

and variable $W_{s,\emptyset}$ has a summand $a.W_{t,\emptyset}$;
9. Whenever $s \xrightarrow{\varepsilon,a,\varepsilon,L} t$, variable $V_{s,\emptyset}$ has a summand

$$a.(lo?\emptyset.li!\emptyset.V_{t,\emptyset} + \sum_{f\in\mathcal{D}'} lo?f.V_{t,f})$$

and variable $W_{s,\emptyset}$ has a summand $a.W_{t,\emptyset}$;
10. Whenever $s\downarrow$, then variable $W_{s,\emptyset}$ has a summand $\mathbf{1}$.

As before, it can checked that this definition of q satisfies the theorem. □

The converse of this theorem does not hold in full generality, as a regular process can communicate with a pair a stacks in ways that cannot be mimicked by a tape. For instance, by means of the stacks, an extra cell on the tape can be inserted or removed. We can obtain a converse of this theorem, nonetheless, if we interpose, between the regular process and the two stacks, an additional regular process *Tape*, that can only perform actions that relate to tape manipulation, viz.

1. $o!d$ ($d \in \mathcal{D}'$), the current symbol can be read;
2. $o!\varepsilon$, we are looking at a blank cell at the end of the string;
3. $i?e$ ($e \in \mathcal{D}'$), the current symbol can be replaced;
4. $i?\varepsilon$, the current symbol can be erased if we are at an end of the string;
5. $i?L$, a move one cell to the left, executed by pushing the current symbol on top of the right-hand stack and popping the left-hand stack;
6. $i?R$, a move one cell to the right, executed by pushing the current symbol on top of the left-hand stack and popping the right-hand stack.

Thus, we have given a characterization of what is a computable process.

In [2], a computable process was defined in a different way. Starting from a classical Turing machine, the internal communication is made explicit just

like we did, by a regular process communicating with two stacks. This shows that a computable function can be described in this way. Next, a computable transition system is coded by means of a computable branching degree function and a computable outgoing edge labeling function. Next, this is again mimicked by a couple of regular processes communicating with a stack. Using this, a similar characterization of computable processes can be reached.

Theorem 8. *A process is computable in the sense of [2] iff it is computable as defined here.*

Proof. Sketch.

If a process is computable in the sense of [2] then we can find a regular process communicating with two stacks such that their parallel composition, after abstraction, is branching bisimilar to it. Moreover, the two stacks together can behave as a tape. Using the theorem above, this means that the process is computable in our sense.

For the other direction, if a process is computable in our sense, then there is a Turing machine for it as defined above. From this Turing machine, we can compute in each state the branching degree and the labels of the outgoing edges. Thus, these functions are computable, and the process is computable in the sense of [2]. □

What remains to be done, is to find a characterization of all recursive specifications over TCP that, after abstraction, give a computable process. In [2], it was found that all guarded recursive specifications over the algebra there yielded computable processes, but that was in the absence of the constant $\mathbf{1}$. We already found, in the previous section, that guardedness is not enough in the presence of $\mathbf{1}$, and we needed to require transparency-restrictedness. It remains to find a way to extend this notion to all of TCP, so including parallel composition.

6 Conclusion

Every undergraduate curriculum in computer science contains a course on automata theory and formal languages. On the other hand, an introduction to concurrency theory is usually not given in the undergraduate program. Both theories as basic models of computation are part of the foundations of computer science. Automata theory and formal languages provide a model of computation where interaction is not taken into account, so a computer is considered as a stand-alone device executing batch processes. On the other hand, concurrency theory provides a model of computation where interaction is taken into account. Concurrency theory is sometimes called the theory of reactive processes.

Both theories can be integrated into one course in the undergraduate curriculum, providing students with the foundation of computing. This paper provides a glimpse of what happens to the Chomsky hierarchy in a concurrency setting, taking a labeled transition system as a central notion, and dividing out bisimulation semantics on such transition systems.

References

1. J.C.M. Baeten, T. Basten, and M.A. Reniers. *Process Algebra (Equational Theories of Communicating Processes)*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2009.
2. J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51:129–176, 1987.
3. J.C.M. Baeten, F. Corradini, and C.A. Grabmayer. A characterization of regular expressions under bisimulation. *Journal of the ACM*, 54(2):6.1–28, 2007.
4. J.C.M. Baeten, P.J.L. Cuijpers, and P.J.A. van Tilburg. A context-free process as a pushdown automaton. In F. van Breugel and M. Chechik, editors, *Proceedings CONCUR'08*, number 5201 in Lecture Notes in Computer Science, pages 98–113, 2008.
5. T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.
6. D. Caucal. On the transition graphs of Turing machines. In M. Margenstern and Y. Rogozhin, editors, *Proceedings MCU 2001*, number 2055 in Lecture Notes in Computer Science, pages 177–189, 2001.
7. R.J. van Glabbeek. The Linear Time – Branching Time Spectrum II. In E. Best, editor, *Proceedings of CONCUR '93*, number 715 in LNCS, pages 66–81. Springer Verlag, 1993.
8. R.J. van Glabbeek. What is Branching Time Semantics and why to use it? *Bulletin of the EATCS*, 53:190–198, 1994.
9. R.J. van Glabbeek. The Linear Time – Branching Time Spectrum I. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. Elsevier, 2001.
10. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.
11. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, 2006.
12. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
13. F. Moller. Infinite results. In U. Montanari and V. Sassone, editors, *Proceedings CONCUR'96*, number 1119 in Lecture Notes in Computer Science, pages 195–216, 1996.
14. D.E. Muller and P.E. Schupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37:51–75, 1985.
15. Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.