Turing Meets Milner

Jos C.M. Baeten[1,2], Bas Luttik[2,3], and Paul van Tilburg[2]

[1] Centrum Wiskunde & Informatica (CWI),
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands,
[2] Division of Computer Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands,
[3] Department of Computer Science, Vrije Universiteit Amsterdam,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{j.c.m.baeten,s.p.luttik,p.j.a.v.tilburg}@tue.nl

Abstract. We enhance the notion of a computation of the classical theory of computing with the notion of interaction from concurrency theory. In this way, we enhance a Turing machine as a model of computation to a Reactive Turing Machine that is an abstract model of a computer as it is used nowadays, always interacting with the user and the world.

1 Introduction

What is a computation? This is a central question in the theory of computing, dating back to the work of Alan Turing in 1936 [?]. The classical answer is that a computation is given by a Turing machine, with the input given on its tape at the beginning, after which a deterministic sequence of steps takes place, leaving the output on the tape at the end. A computable function is a function of which the transformation of input to output can be computed by a Turing machine.

A Turing machine can serve in this way as a basic model of a computation, but cannot serve as a basic model of a computer. Well, it could up to the advent of the terminal in the 1970s. Before that, input was given as a stack of punch cards at the start, and output of a computation appeared as a printout later. The terminal made direct interaction with the computer possible. Nowadays, a computer is interacting continuously, with the user at the click of a mouse or with many other computers all over the world through the Internet.

An execution of a computer is thus not just a series of steps of a computation, but also involves interaction. It cannot be modeled as a function, and has inherent nondeterminism. In this paper, we make the notion of an execution precise, and compare this to the notion of a computation. To illustrate the difference between a computation and an execution, we can say that a Turing machine cannot fly an airplane, but a computer can. An automatic pilot cannot know all weather conditions en route beforehand, but can react to changing conditions run-time.

Computability theory is firmly grounded in automata theory and formal language theory. It progresses from the study of finite automata to pushdown automata and Turing machines. Of these different classes of automata, it studies the languages, the sets of strings, induced by them. We can view a language as an equivalence class of automata (under language equivalence).

The notion of interaction has been studied extensively in concurrency theory and process theory exemplified by the work of Robin Milner [6]. It embodies a powerful parallel composition operator that is used to compose systems in parallel, including their interaction. The semantics of concurrency theory is mostly given in terms of transition systems, that are almost like automata. However, there are important differences.

First of all, a notion of final state, of termination, is often missing in concurrency theory. The idea is that concurrency theory often deals with so-called *reactive systems*, which need not terminate but are always on, reacting to stimuli from the environment. As a result, termination is often neglected in concurrency theory, but is nevertheless an important ingredient, as shown and fully worked out in [1]. Using this presentation of concurrency theory as a starting point, we obtain a full correspondence with automata theory: a finite transition system is exactly a finite automaton. On the other hand, we stress that we fully incorporate the reactive systems approach of concurrency theory: non-terminating behaviour is also relevant behaviour, that is taken into account.

A second difference between automata theory and concurrency theory is that transition systems need not be finite. Still, studying the subclass of finite transition systems yields useful insights for the extension to pushdown automata and Turing machines.

The third and main difference between automata theory and concurrency theory is that language equivalence is too coarse to capture a notion of interaction. Looking at an automaton as a language acceptor, acceptance of a string represents a particular computation of the automaton, and the language is the set of all its computations. 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 some form of interaction of the automaton with its environment (user, other automata running in parallel, etc.) may influence the course of computation.

Therefore, other notions of equivalence are studied in concurrency theory, capturing more of the branching structure of an automaton. Prominent among these is bisimulation equivalence [**?**]. When silent steps are taken into account, the preferred variant is *branching bisimilarity*, arguably preserving all relevant moments of choice in a system [4]. Moreover, it is important to keep track of possible divergencies as advocated in the work on CSP [5].

In this paper we study the notion of a computation, taking interaction into account. We define, next to the notion of a computable function, the notion of an executable process. An executable process is a behaviour that can be exhibited by a computer (interacting with its environment). An executable process is a divergence-preserving branching bisimulation equivalence class of transition systems defined by a Reactive Turing Machine. A Reactive Turing Machine is an adaptation of the classical Turing Machine that can properly deal with ubiquitous interaction. Leading up to the definition of the Reactive Turing Machine, we reconsider some of the standard results from automata theory when automata are considered modulo *divergence-preserving branching bisimilarity* instead of language equivalence.

This paper is an update of [?] with results from [?], [?] and [?].

In Section 3 we consider *finite-state processes*, defined as divergence-preserving branching bisimulation equivalence classes of finite labeled transition systems that are finite automata. The section illustrates the correspondence between finite automata and linear recursive specifications that can be thought of as the process-theoretic counterpart of regular grammars.

In Section 4 we consider *pushdown processes*, defined as divergence-preserving branching bisimulation equivalence classes of labeled transition systems associated with pushdown automata. We investigate the correspondence between pushdown processes and processes definable by sequential recursive specifications, which can be thought of as the process-theoretic counterpart of context-free grammars. We show this correspondence is not optimal, and define a new grammar that fits better, based on the universality of the stack process.

In Section 5 we define *executable processes*, defined as divergence-preserving branching bisimulation equivalence classes of labeled transition systems associated with Reactive Turing Machines. We highlight the relationship of computable functions and executable processes, laying the foundations of executability theory alongside computability theory. We define a new grammar for executable processes, based on the universality of the queue process.

2 Process theory

In this section we briefly recap the basic definitions of the process algebra TCP$^*_\tau$ (Theory of Communicating Processes with silent step and iteration). This process algebra has a rich syntax, allowing to express all key ingredients of concurrency theory, including termination that enables a full incorporation of regular expressions. It also has a rich theory, fully worked out in [1].

Syntax. We presuppose a finite *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. We furthermore presuppose a finite *data alphabet* $\mathcal{D}$, a finite set $\mathcal{C}$ of *channels*, and assume that $\mathcal{A}$ includes special actions $c?d$, $c!d$, $c!?d$ ($d \in \mathcal{D}$, $c \in \mathcal{C}$), 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:

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

Let us briefly comment on the operators in this syntax. The constant $\mathbf{0}$ denotes inaction or *deadlock*, the unsuccessfully terminated process. It can be thought of as the automaton with one initial state that is not final and no transitions. The constant $\mathbf{1}$ denotes the successfully terminated process. It can be thought of as the automaton with one initial state that is final, without transitions. 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. We write $\mathcal{A}_\tau$ for $\mathcal{A} \cup \{\tau\}$. The binary operator $\cdot$ denotes *sequential composition.* The unary operator * is iteration or *Kleene star.* 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 $c!?d$ 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 $c!?d$ 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 $N \overset{\text{def}}{=} p$ 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.

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 (Labeled 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 see that a labeled transition system with a finite set of states is exactly a finite (nondeterministic) automaton.

We use Structural Operational Semantics [?] 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 the operational rules in Table 1.1. Note that the operational rules presuppose a recursive specification E.

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

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

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

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

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

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

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

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

Table 1.1: Operational rules for TCP$^*_\tau$ and 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 $s \overset{w}{\twoheadrightarrow} s'$ 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. ε denotes the empty word. We say a state $t \in \mathcal{S}$ is *reachable* from a state $s \in \mathcal{S}$ if there exists $w \in \mathcal{A}^*$ such that $s \overset{w}{\twoheadrightarrow} t$.

Definition 2. *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 1.1, 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., $\downarrow_p = \downarrow \cap \mathcal{S}_p$.*

If we start out from a process expression not containing a name, then the transition system defined by this construction is finite and so is a finite automaton.

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 [4]. For motivations to use branching bisimulation as the preferred notion of equivalence, see [**?**]. Moreover, by taking divergence into account, as advocated e.g. by [5], most of our results do not depend on fairness assumptions. Divergence-preserving branching bisimulation is called branching bisimulation with explicit divergence in [4].

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 3 (Divergence-preserving 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 $s_2 \overset{\varepsilon}{\twoheadrightarrow}_2 s_2'' \xrightarrow{(a)}_2 s_2'$, $s_1 \mathcal{R} s_2''$ 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 $s_1 \overset{\varepsilon}{\twoheadrightarrow}_1 s_1'' \xrightarrow{(a)}_1 s_1'$, $s_1'' \mathcal{R} s_2$ and $s_1' \mathcal{R} s_2'$;*
3. *if $s_1\downarrow_1$, then there exists s_2' such that $s_2 \overset{\varepsilon}{\twoheadrightarrow}_2 s_2'$ and $s_2'\downarrow_2$; and*
4. *if $s_2\downarrow_2$, then there exists s_1' such that $s_1 \overset{\varepsilon}{\twoheadrightarrow}_1 s_1'$ and $s_1'\downarrow_1$.*

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

A branching bisimulation $\mathcal{R}$ from L_1 to L_2 is divergence-preserving *if for all states s_1, s_2, $s_1 \mathcal{R} s_2$ implies*

5. *if there exists an infinite sequence $s_{1,0}, s_{1,1}, s_{1,2}, \ldots$ such that $s_1 = s_{1,0}$, $s_{1,i} \xrightarrow{\tau}_1 s_{1,i+1}$ and $s_{1,i}\mathcal{R}s_2$ for all i, then there exists a state $s_2{}'$ such that $s_2 \overset{\varepsilon}{\twoheadrightarrow}_2 s_2{}'$ with at least one step and $s_{1,i}\mathcal{R}s_2{}'$ for some i; and*
6. *if there exists an infinite sequence $s_{2,0}, s_{2,1}, s_{2,2}, \ldots$ such that $s_2 = s_{2,0}$, $s_{2,i} \xrightarrow{\tau}_2 s_{2,i+1}$ and $s_1\mathcal{R}s_{2,i}$ for all i, then there exists a state $s_1{}'$ such that $s_1 \overset{\varepsilon}{\twoheadrightarrow}_1 s_1{}'$ with at least one step and $s_1{}'\mathcal{R}s_{2,i}$ for some i.*

Labeled transition systems L_1 and L_2 are divergence-preserving branching bisimilar *(notation: $L_1 \leftrightarrow_b^\Delta L_2$) if there exists a divergence-preserving branching bisimulation from L_1 to L_2.*

(Divergence-preserving) branching bisimilarity is an equivalence relation on labeled transition systems [?,?]. A branching bisimulation from a transition system to itself is called a branching bisimulation *on* this transition system. Each transition system has a maximal branching bisimulation, identifying as many states as possible, found as the union of all possible branching bisimulations. Dividing out this maximal branching bisimulation, we get the quotient of the transition system w.r.t. the maximal branching bisimulation. We define the *branching degree* of a state as the cardinality of the set of outgoing edges of its equivalence class in the maximal divergence-preserving branching bisimulation.

A transition system has *finite branching* if all states have a finite branching degree. We say a transition system has *bounded branching* if there exists a natural number $n \geq 0$ such that every state has a branching degree of at most n. Divergence-preserving branching bisimulations respect branching degrees.

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 nondeterminism is relevant and basic in concurrency theory, we look at a nondeterministic finite automaton. As a nondeterministic finite automaton is exactly a finite labeled transition system, we refer to Definition 1 for the definition.

Two examples of finite automata are given in Figure 1.

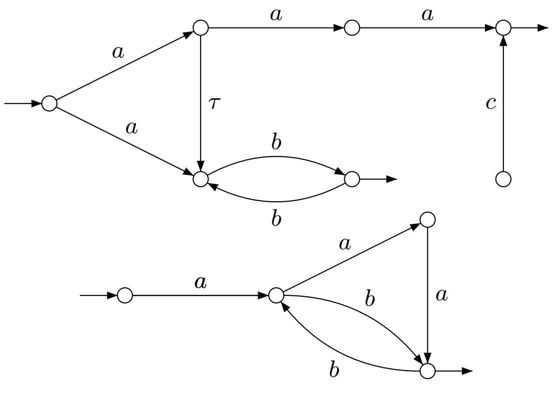

Fig. 1. Two examples of finite automata.

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 $a \in \mathcal{A}'$, $s \xrightarrow{a}\!\!\twoheadrightarrow t_1$ and $s \xrightarrow{a}\!\!\twoheadrightarrow t_2$ implies $t_1 = t_2$.

In the theory of automata and formal languages, it is usually also required in the definition of a deterministic automaton 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 1 is nondeterministic 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 4 (Language equivalence). *The* language $\mathcal{L}(L)$ *accepted by a labeled transition system* $L = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ *is defined as*

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

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

The language of both automata in Figure 1 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 divergence-preserving branching bisimilarity class of labeled transition systems that contains a finite automaton.

In automata theory, every silent step τ and all nondeterminism can be removed from a finite automaton. These results are no longer valid when we consider finite automata modulo branching bisimulation. 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 1. Nor does there exist a deterministic finite automaton branching bisimilar with the upper finite automaton in Figure 1.

Regular expressions A *regular expression* is a process expression using only the first 7 items in the definition of process syntax above, so does not contain parallel composition or recursion. Not every regular process is given by a regular expression, see [2]. We show a simple example in Figure 2 of a finite transition system that is not bisimilar to any transition system that can be associated with a regular expression.

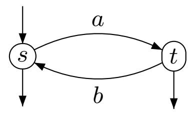

Fig. 2. Not bisimilar to a regular expression.

However, if we can also use parallel composition and encapsulation, then we can find an expression for every finite automaton, see [**?**]. Abstraction and recursion are not needed for this result. We can illustrate this with the finite automaton in Figure 2, but need to replace the label a by $st!?a$ and label b by $ts!?b$. Then, we can define the following expressions for states s, t:

$$s = (ts?b.(st!a.\mathbf{1} + \mathbf{1}))^*, \qquad t = (st?a.(ts!b.\mathbf{1} + \mathbf{1}))^* \ .$$

The expressions show the possibilities to enter a state, followed by the possibilities to leave a state, and then iterate. Then, we compose the expressions of the states in a parallel composition:

$$\partial_{st,ts}(((st!a.\mathbf{1} + \mathbf{1}) \cdot s) \parallel \mathbf{1} \cdot t) \ .$$

Using the operational rules on this resulting expression gives again the finite automaton in Figure 2.

Regular grammars 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.

If we use only the syntax elements $\mathbf{0}$, $\mathbf{1}$, N ($N \in \mathcal{N}'$), $a.\_$ ($a \in \mathcal{A}_\tau$) and $\_+\_$ of the definition above, then we get so-called *linear* recursive specifications. Thus, we do not use sequential composition, parallel composition, encapsulation and abstraction.

We have the result that every linear recursive specification by means of the operational rules defined generates a finite automaton, but also conversely, every finite automaton can be specified, up to isomorphism, by a linear recursive specification. We illustrate the construction with an example.

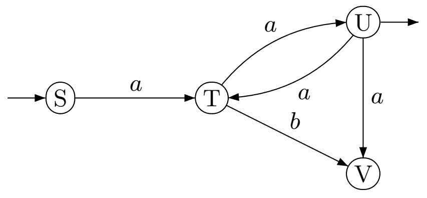

Fig. 3. Example automaton.

Consider the automaton depicted in Figure 3. 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 an equation, in such a way that the labeled transition system $\mathcal{T}_E(S)$ generated by the operational semantics in Table 1.1 is isomorphic (so certainly divergence-preserving branching bisimilar) with the automaton in Figure 3.

The recursive specification for the finite automaton in Figure 3 is:

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

This result 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. If we use *action postfix* instead of action prefix, then on the one hand not every finite automaton can be specified, and on the other hand, by means of a simple recursive equation we can specify an infinite transition system (see [**?**]).

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 5 (Pushdown automaton). *A* pushdown automaton *M is defined as a sixtuple $(\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ where:*

1. *$\mathcal{S}$ a finite set of states,*
2. *$\mathcal{A}$ is a finite action alphabet,*
3. *$\mathcal{D}$ is a finite data alphabet, and $\emptyset \notin \mathcal{D}$ is a special symbol denoting an empty stack,*
4. *$\rightarrow \subseteq \mathcal{S} \times \mathcal{A}_\tau \times (\mathcal{D} \cup \{\emptyset\}) \times \mathcal{D}^* \times \mathcal{S}$ is a $\mathcal{A}_\tau \times (\mathcal{D} \cup \{\emptyset\}) \times \mathcal{D}^*$-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 $(s, a, d, \delta, t) \in \rightarrow$, we write $s \xrightarrow{a[d/\delta]} t$.

The pair of a state together with particular stack contents will be referred to as the *configuration* of a pushdown automaton. Intuitively, a transition $s \xrightarrow{a[d/\delta]} t$ (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 $s \xrightarrow{a[\emptyset/\delta]} t$ 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 $s \xrightarrow{\tau[d/\delta]} t$ or $s \xrightarrow{\tau[\emptyset/\delta]} t$ 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. We also have a third approach in which a configuration is terminating if it consists of a terminating state *and* an empty stack (FSES). We note 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. We choose the FS approach here, as this gives us more flexibility, allows us to define more pushdown processes. For further details, see [**?**,**?**].

Definition 6. *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 $s \xrightarrow{a[d/\delta]} t$ 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 $s \xrightarrow{a[\emptyset/\delta]} t$;*
3. *the initial state of $\mathcal{T}(M)$ is $(\uparrow, \varepsilon)$; and*
4. *the set of final states is $\{(s, \zeta) \mid s\downarrow, \zeta \in \mathcal{D}^*\}$.*

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 divergence-preserving branching bisimilarity class of labeled transition systems containing a labeled transition system associated with a pushdown automaton.

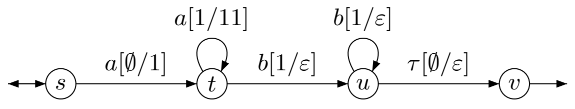

Fig. 4. Example pushdown automaton.

As an example, the pushdown automaton in Figure 4 defines the infinite transition system in Figure 5, that accepts the language $\{a^n b^n \mid n \geq 0\}$.

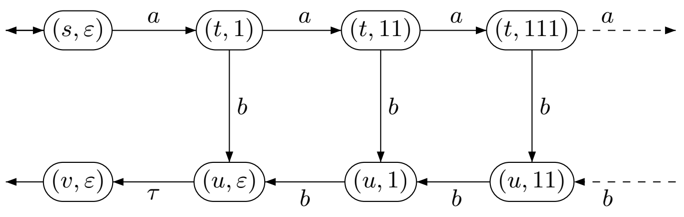

Fig. 5. A pushdown process.

Context-free grammars 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 so-called *sequential* recursive specifications in which only the constructions $\mathbf{0}$, $\mathbf{1}$, N ($N \in \mathcal{N}'$), $a.\_$ ($a \in \mathcal{A}_\tau$), $\_ \cdot \_$ and $\_ + \_$ occur, so adding sequential composition to linear recursive specifications.

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

$$X \overset{\text{def}}{=} \mathbf{1} + a.X \cdot b.\mathbf{1}$$

has a labeled transition system that is divergence-preserving branching bisimilar to the one in Figure 5, which is associated with the pushdown automaton in Figure 4.

The notion of a sequential recursive specification naturally corresponds with with the notion of context-free grammar: for every pushdown automaton there exists a sequential recursive specification such that their transition systems are language equivalent, and, vice versa, for every sequential recursive specification there exists a pushdown automaton such that their transition systems are language equivalent.

A similar result with language equivalence replaced by divergence-preserving branching bisimilarity does not hold. There are pushdown processes that are not recursively definable by a sequential recursive specification, and also there are sequential recursive specifications that define non-pushdown processes. Extra restrictions are necessary in order to retrieve the desired equivalence, see [**?**,**?**]. Here, we limit ourselves by just giving examples about the difficulties involved.

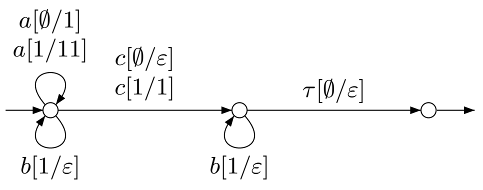

Fig. 6. Pushdown automaton that does not have a sequential recursive specification.

Consider the pushdown automaton in Figure 6, which generates the transition system shown in Figure 7 (omitting the τ-step, this preserves divergence-preserving branching bisimilarity). In [7], Moller proved that this transition system cannot be defined with a BPA recursive specification, where BPA is the restriction of sequential recursive specifications 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 sequential recursive specification either. We conclude that not every pushdown process is definable with a sequential recursive specification.

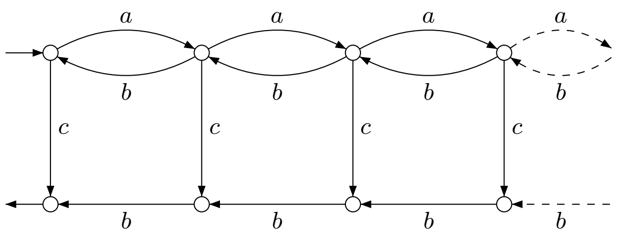

Fig. 7. Transition system of automaton of Figure 6.

Another example of a pushdown automaton that does not have a sequential recursive specification is the stack itself, used as memory in the definition of a pushdown automaton. The stack can be modeled as a pushdown process, in fact (as we will see shortly) it is the prototypical pushdown process. Given a finite nonempty data set $\mathcal{D}$, the stack St^{io} has an input channel i over which it can receive elements of $\mathcal{D}$ and an output channel o over which it can signal it is empty, and 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 $\uparrow \xrightarrow{i?d[\emptyset/d]} \uparrow$, $\uparrow \xrightarrow{i?d[e/de]} \uparrow$, and $\uparrow \xrightarrow{o!\emptyset[\emptyset/\varepsilon]} \uparrow$, $\uparrow \xrightarrow{o!d[d/\varepsilon]} \uparrow$ for all $d, e \in \mathcal{D}$. As the transition system generated by this pushdown automaton has infinitely many final states that are not branching bisimilar, it can be shown there is no sequential recursive specification for it. If we allow termination only when the stack is empty, then we can find the following sequential recursive specification:

$$S \overset{\text{def}}{=} \mathbf{1} + o!\emptyset.\mathbf{1} + \sum_{d \in \mathcal{D}} i?d.T \cdot o!d.S \qquad\qquad T \overset{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.T \cdot o!d.T.$$

Conversely, not every sequential recursive specification defines a pushdown process. To give an example, the sequential recursive equation $X \overset{\text{def}}{=} X \cdot a.\mathbf{1}$ generates an infinitely branching transition system, which can only be given a pushdown automaton at the cost of introducing divergencies. This infinite branching is due to the unguardedness of the equation, but even for guarded equations, not always is pushdown process is defined. To give an example, consider the following recursive specification:

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

The labeled transition system associated with X, which is depicted in Figure 8, has finite but unbounded branching. We claim this cannot be a pushdown process.

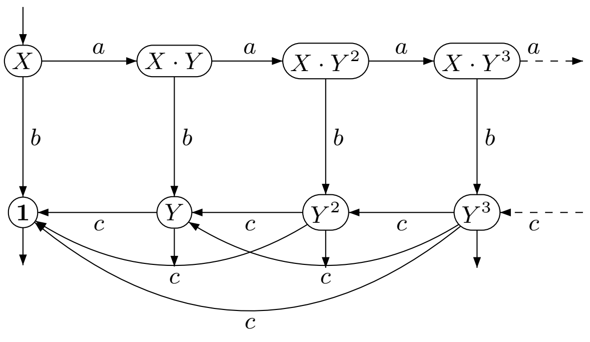

Fig. 8. Process with unbounded branching.

Because of these difficulties with the correspondence between pushdown processes and sequential recursive specifications, we consider another grammar for pushdown processes. This grammar starts by giving a specification for the stack process St^{io} defined above (that can terminate irrespective of the contents).

We start out from the observation that an unbounded stack can be seen as a buffer of capacity one (for the top of the stack) communicating with a copy of the unbounded stack. An unbounded stack with input port i and output port o is equal to a regular *Top* process with external input i, internal input j, external output o, internal output p, communicating with an unbounded stack with input port j and output port p. See Figure 9.

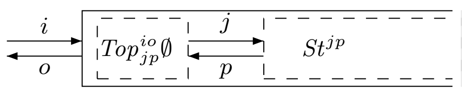

Fig. 9. Intuition for specification of always terminating stack.

In a formula, we want to achieve

$$St^{io} \mathrel{\underline{\leftrightarrow}}_{\mathrm{b}}^{\Delta} \tau_{jp}(\partial_{jp}(Top^{io}_{jp}\emptyset \parallel St^{jp})).$$

In turn, the stack with input j and output p will satisfy

$$St^{jp} \mathrel{\underline{\leftrightarrow}}_{\mathrm{b}}^{\Delta} \tau_{io}(\partial_{io}(Top^{jp}_{io}\emptyset \parallel St^{io})).$$

We do this by the following specification over names St^{io} and St^{jp}, with auxiliary variables $Top^{io}_{jp}\emptyset$, $Top^{jp}_{io}\emptyset$ and $Top^{io}_{jp}d$, $Top^{jp}_{io}d$ for every $d \in \mathcal{D}$:

$$\begin{aligned}
St^{io} &\stackrel{\text{def}}{=} \mathbf{1} + o!\emptyset.St^{io} + \sum_{d\in\mathcal{D}} i?d.\tau_{jp}(\partial_{jp}(Top^{io}_{jp}d \parallel St^{jp}))\\
St^{jp} &\stackrel{\text{def}}{=} \mathbf{1} + p!\emptyset.St^{jp} + \sum_{d\in\mathcal{D}} j?d.\tau_{io}(\partial_{io}(Top^{jp}_{io}d \parallel St^{io}))\\
Top^{io}_{jp}\emptyset &\stackrel{\text{def}}{=} \mathbf{1} + o!\emptyset.Top^{io}_{jp}\emptyset + \sum_{d\in\mathcal{D}} i?d.Top^{io}_{jp}d\\
Top^{jp}_{io}\emptyset &\stackrel{\text{def}}{=} \mathbf{1} + p!\emptyset.Top^{jp}_{io}\emptyset + \sum_{d\in\mathcal{D}} j?d.Top^{jp}_{io}d\\
Top^{io}_{jp}d &\stackrel{\text{def}}{=} \mathbf{1} + o!d.(p?\emptyset.Top^{io}_{jp}\emptyset + \sum_{e\in\mathcal{D}} p?e.Top^{io}_{jp}e) + \sum_{f\in\mathcal{D}} i?f.j!d.Top^{io}_{jp}f\\
Top^{jp}_{io}d &\stackrel{\text{def}}{=} \mathbf{1} + p!d.(o?\emptyset.Top^{jp}_{io}\emptyset + \sum_{e\in\mathcal{D}} o?e.Top^{jp}_{io}e) + \sum_{f\in\mathcal{D}} j?f.i!d.Top^{jp}_{io}f
\end{aligned}$$

The last two equations occur for every $d \in \mathcal{D}$. Notice that the subspecification of the *Top* processes define these as regular processes.

On the basis of this specification, the divergence-preserving branching bisimilarities above are straightforward to prove (even strong bisimilarity holds).

The stack process can be used to make the interaction between control and memory in a pushdown automaton explicit [3]. This is illustrated by the following theorem, stating that every pushdown process is equal to a regular process interacting with a stack.

Theorem 1. *For every pushdown automaton M there exists a regular process expression p and a linear recursive specification E, and for every regular process expression p and linear recursive specification there exists a pushdown automaton M such that*

$$\mathcal{T}(M) \leftrightarroweq_b \mathcal{T}_{E\cup E_S}(\tau_{i,o}(\partial_{i,o}(p \parallel St^{io}))) \ .$$

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.

Definition 7 (Reactive Turing Machine). *A* Reactive 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 action alphabet, $\mathcal{A}_\tau$ also includes the silent step τ,*
3. *$\mathcal{D}$ is a finite data alphabet, we add a special symbol $\Box$ standing for a blank and put $\mathcal{D}_\Box = \mathcal{D} \cup \{\Box\}$,*
4. *$\rightarrow \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathcal{D}_\Box \times \mathcal{D}_\Box \times \{L, R\} \times \mathcal{S}$ 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 $(s, a, d, e, M, t) \in \rightarrow$, we write $s \xrightarrow{a[d/e]M} t$, 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 $\Box$: if d is $\Box$, the reading head is looking at an empty cell on the tape and writes e; if e is $\Box$ and d is not, then d is erased, leaving an empty cell. 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 (only contains blanks).

By looking at all possible executions, we can define the transition system of a Turing machine. The states of this transition system are the configurations of the Reactive Turing Machine, consisting of a state, the current tape contents, and the position of the read/write head. We represent the tape contents by an element of $\mathcal{D}_\Box^*$, replacing exactly one occurrence of a type symbol d by a marked symbol $\bar{d}$, indicating that the read/write head is on this symbol. We denote by $\bar{\mathcal{D}}_\Box = \{\bar{d} \mid d \in \mathcal{D}_\Box\}$ the set of marked tape symbols; a *tape instance* is a sequence $\delta \in (\mathcal{D}_\Box \cup \bar{\mathcal{D}}_\Box)$ such that δ contains exactly one element of $\bar{\mathcal{D}}_\Box$.

A tape instance thus is a finite sequence of symbols that represents the contents of a two-way infinite tape. We do not distinguish between tape instances that are equal modulo the addition or removal of extra occurrences of a blank at the left or right extremes of the sequence. The set of configurations of a Reactive Turing Machine now consists of pairs of a state and a tape instance. In order to concisely describe the semantics of a Reactive Turing Machine in terms of transition systems on configurations, we use some additional notation.

If $\delta \in \mathcal{D}_\Box$, then δ^{-} is the tape instance obtained by placing the marker on the right-most symbol of δ if this exists, and $\bar{\Box}$ otherwise. Likewise, ${}^{-}\delta$ is the tape instance obtained by placing the marker on the left-most symbol of δ if this exists, and $\bar{\Box}$ otherwise.

Definition 8. *Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a Reactive Turing Machine. The* labeled transition system *of M, $\mathcal{T}(M)$, is defined as follows:*

1. *The set of states is the set of configurations $\{(s, \delta) \mid s \in \mathcal{S}, \delta$ a tape instance$\}$.*
2. *The transition relation $\rightarrow$ is the least relation satisfying, for all $a \in \mathcal{A}_\tau, d, e \in \mathcal{D}_\Box, \delta, \zeta \in \mathcal{D}_\Box^*$:*
   - $(s, \delta\bar{d}\zeta) \xrightarrow{a} (t, \delta^{-}e\zeta)$ *iff* $s \xrightarrow{a[d/e]L} t$,
   - $(s, \delta\bar{d}\zeta) \xrightarrow{a} (t, \delta e\,{}^{-}\zeta)$ *iff* $s \xrightarrow{a[d/e]R} t$.
3. *The initial state is $(\uparrow, \bar{\Box})$;*
4. *$(s, \delta) \downarrow$ iff $s \downarrow$.*

Now we define an *executable process* as the divergence-preserving branching bisimulation equivalence class of a transition system of a Reactive Turing Machine.

As an example of a Reactive Turing Machine, we define the (first-in first-out) queue over a data set $\mathcal{D}$. It has the initial and final state at the head of the queue. There, output of the value at the head can be given, after which one move to the left occurs. If an input comes, then the position travels to the left until a free position is reached, where the value input is stored, after which the position travels to the right until the head is reached again. We show the Turing machine in Figure 10 in case $\mathcal{D} = \{0, 1\}$. A label containing an n, like $\tau[n/n]L$ means there are two labels $\tau[0/0]L$ and $\tau[1/1]L$.

The queue process is an executable process, but not a pushdown process.

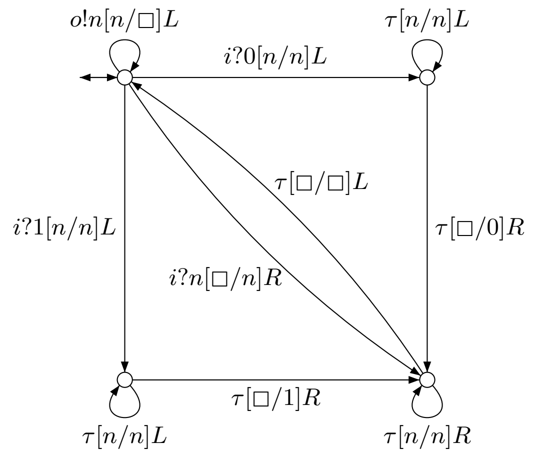

Fig. 10. Reactive Turing Machine for the FIFO queue.

We call a transition system *computable* if it is finitely branching and there is a coding of the states such that the set of final states is decidable and for each state, we can determine the set of outgoing transitions. A transition system is *effective* if its set of transitions and set of final states are recursively enumerable. The following results are in [**?**].

Theorem 2. *The transition system defined by a Reactive Turing Machine is computable.*

Theorem 3. *Every boundedly branching computable transition system is executable.*

Theorem 4. *The parallel composition of two executable transition systems is again executable.*

Theorem 5. *For each n, Reactive Turing Machine exists, that is universal for all Reactive Turing Machines that have a transition system with branching degree bounded by n.*

A truly universal Reactive Turing Machine can only be achieved at the cost of introducing divergencies.

As in the case of the pushdown automaton, we can make the interaction between the finite control and the memory explicit, and turn this into a recursive specification.

Theorem 6. *For every Reactive Turing Machine M there exists a regular process expression p and a linear recursive specification E, and for every regular process expression p and linear recursive specification there exists a Reactive Turing Machine M such that*

$$\mathcal{T}(M) \leftrightarroweq_b^{\Delta} \mathcal{T}_{E \cup E_Q}(\tau_{i,o}(\partial_{i,o}(p \parallel Q^{io}))) \ .$$

In this theorem, we use the queue process as defined above, and its specification E_Q to be defined next. By putting a finite control on top of a queue, we can simulate the tape process of a Reactive Turing Machine. The control of the Turing machine together with this control, can be specified as a finite-state process.

We finish by giving a finite recursive specification E_Q for a queue with input channel i and output channel o, Q^{io}. It follows the same pattern as the one for the stack in Figure 9: a queue with input port i and output port o is the same as the queue with input port i and output port ℓ communicating with a regular *head* process with input ℓ and output o. See Figure 11.

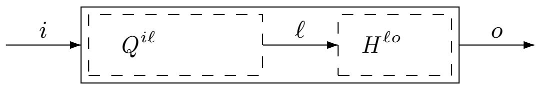

Fig. 11. Intuition for specification of always terminating queue.

.

In a formula, we want to achieve

$$Q^{io} \leftrightarroweq_{\mathrm{b}}^{\Delta} \tau_{\ell}(\partial_{\ell}(Q^{i\ell} \parallel H^{\ell o})).$$

In turn, the queue with input i and output ℓ will satisfy

$$Q^{i\ell} \leftrightarroweq_{\mathrm{b}}^{\Delta} \tau_{o}(\partial_{o}(Q^{io} \parallel H^{o\ell})).$$

We do this by the following specification over names Q^{io} and $Q^{i\ell}$, with auxiliary variables $H^{\ell o}$ and $H^{o\ell}$, where the *Head* processes are just always terminating one place buffers:

$$Q^{io} \overset{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.\tau_\ell(\partial_\ell(Q^{i\ell} \parallel (\mathbf{1} + o!d.H^{\ell o})))$$

$$Q^{i\ell} \overset{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.\tau_o(\partial_o(Q^{io} \parallel (\mathbf{1} + \ell!d.H^{o\ell})))$$

$$H^{\ell o} \overset{\text{def}}{=} \mathbf{1} + \sum_{e \in \mathcal{D}} \ell?e.(\mathbf{1} + o!e.H^{\ell o})$$

$$H^{o\ell} \overset{\text{def}}{=} \mathbf{1} + \sum_{e \in \mathcal{D}} o?e.(\mathbf{1} + \ell!e.H^{o\ell}).$$

Now, the theorem above implies that recursive specifications over our syntax (even omitting sequential composition and iteration) constitute a grammar for all executable processes.

6 Conclusion

We established the notion of an execution in this paper, that enhances a computation by taking interaction into account. We do this by marrying computability theory, moving up from finite automata through pushdown automata to Turing machines, with concurrency theory, not using language equivalence but divergence-preserving branching bisimilarity on automata.

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, see [**?**]. 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, F. Corradini, and C.A. Grabmayer. A characterization of regular expressions under bisimulation. *Journal of the ACM*, 54(2):6.1–28, 2007.

3. 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.
4. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.
5. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
6. R. Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer Verlag, 1980.
7. 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.