iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 2 **Issue:** X **Month of publication:** October 2014

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

ISSN: 2321-9653
International Journal for Research in Applied Science & Engineering Technology(IJRASET)

A Context-Free Process as a Pushdown Automaton

Saurabh Setia[1], Nipun Jain[2], Paras Thakral[3]

DCE,GGN

Abstract: Pushdown automata are used in theories about what can be computed by machines. They are more capable than finite-state machines but less capable than Turing machines. Deterministic pushdown automata can recognize all deterministic context-free languages while nondeterministic ones can recognize all context-free languages. So in the following research we will be discussing about pushdown automation as a context-free process.

I. INTRODUCTION

A finite-state automaton augmented by a stack of symbols that are distinct from the symbols allowed in the input string. Like the finite-state automaton, the PDA reads its input string once from left to right, with acceptance or rejection determined by the final state. After reading a symbol, however, the PDA performs the following actions: change state, remove top of stack, and push zero or more symbols onto stack. The precise choice of actions depends on the input symbol just read, the current state, and the current top of stack. Since the stack can grow unboundedly, a PDA can have infinitely many different configurations – unlike a finite-state automaton. A nondeterministic PDA is one that has a choice of actions for some conditions. The languages recognized by nondeterministic PDAs are precisely the context-free languages. However not every context-free language is recognized by a deterministic PDA.

II. REGULAR PROCESSES

Before we introduce context-free processes, we first consider the notion of a regular process and its relation to regular languages in automata theory. We start with a definition of the notion of transition system from process theory. A finite transition system can be thought of as a non-deterministic finite automaton. In order to have a complete analogy, the transition systems we study have a subset of states marked as final states.
Definition 1 (Transition system). A transition system M is a quintuple $(S,A,\rightarrow,\uparrow,\downarrow)$ where:

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

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

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

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

Definition 2. THE SIGNATURE OF MINIMAL ALGEBRA MA IS AS FOLLOWS:

1. There is a constant 0; this denotes inaction, a deadlock state; other names are δ or stop.

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

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

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

We find that a finite recursive specification over MA can be seen as a rightlinear grammar. Now each finite transition system corresponds directly to a finite recursive specification over MA, using a variable for every state. To go from a term over MA to a transition system, we use structural operational semantics, with rules

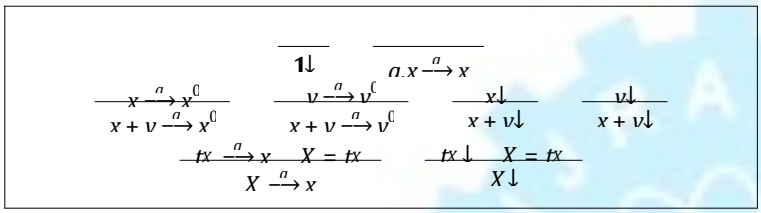


Table 2.1: Operational rules for MA and recursion (a ∈ A,X∈ V).

III. CONTEXT-FREE PROCESSES

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

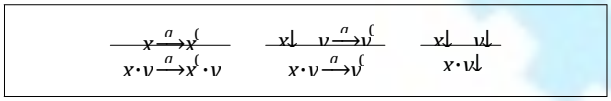


Table 3.1: Operational rules for sequential composition (a ∈ A).

sequential composition · .-We extend the operational rules of Table 2.1 with rules for sequential composition, in Table 3.1. Now consider the following specification

S = 1 + S · a.1.

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

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

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

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

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

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

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

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

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

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

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

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

Now the recursive specification is as follows:

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

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

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

Then it is straightforward to derive the following equations:

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

$\sum_{d\in D}$ $\sum_{e\in D}$

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

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

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

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

Due to the presence of 1, a context-free process may have unbounded branching [8] that we need to mimic with our pushdown automaton. One possible solution is to use forgetfulness of the stack to get this unbounded branching in our pushdown automaton. Note that when using a more restrictive notion of context-free processes we have bounded branching, and thus we don't need the forgetfulness property.

The above presented specifications are still meaningful when D is an infinite data set , but does not represent a term in SA anymore. In this paper, we use infinite summation in some intermediate results, but the end results are finite. Note that the infinite sums also preserve THE NOTION OF CONGRUENCE WE ARE WORKING WITH.

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

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

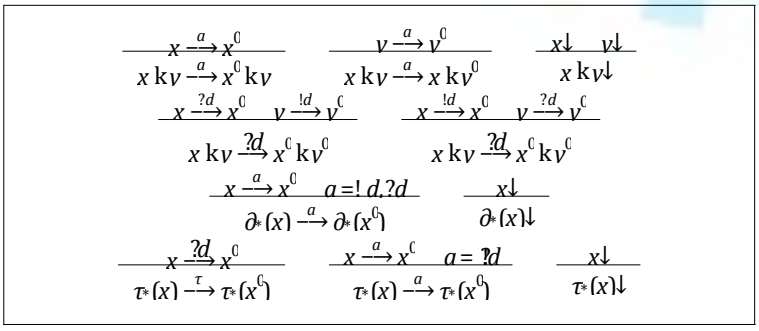


Table 3.3: Operational rules for CA (a ∈ A).

Our finite axiomatization of transition systems of CA modulo rooted branching bisimulation uses the auxiliary operators. See Table 3.4 for the axioms and for an explanation of these axioms.T

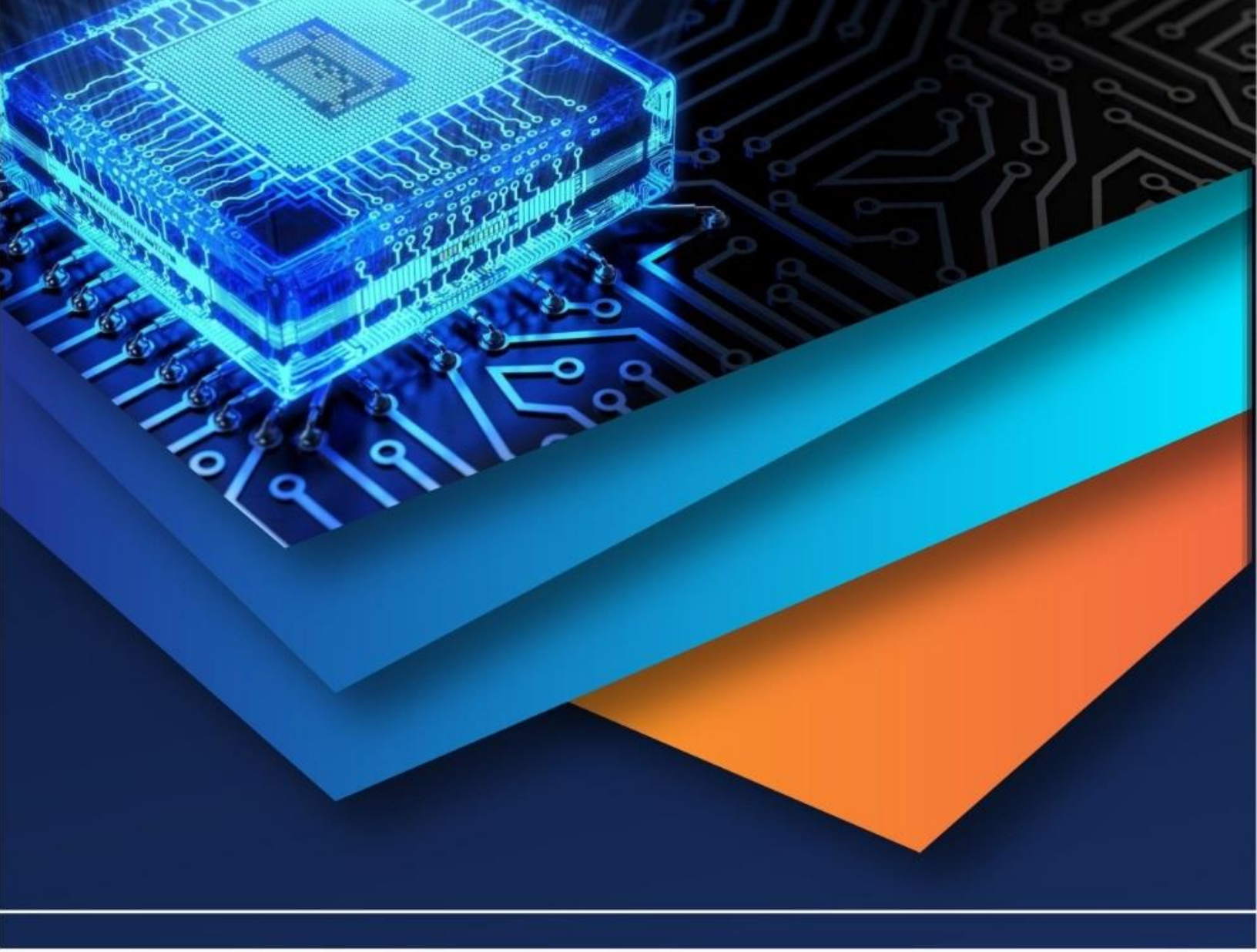
$\mathbf{0} \between x$	$= x \quad y + y \quad x + x \mid y$		$x) = a.(x + y)$
$\mathbf{1} \quad x$	$T \qquad T$	$a.(\tau.(x + y) +$	
x k y			$= y \mid x$
	$= \mathbf{0}$	$x \mid y \; x \; k \; \mathbf{1}$	$= x =$
$\between$	$= \mathbf{0}$	$\mathbf{1} \mid x + \mathbf{1}$	$\mathbf{1}$
$(x + y) \between z$	$= a.(x \; k \; y)$	$(x \; k \; y) \; k \; z$	$= x \; k \; (y \; k \; z)$
$\mathbf{0} \mid x$	$= x \between z + y \between z$	$(x \mid y) \mid z$	$= x \mid (y \mid z) =$
a.xTy	$= \mathbf{0}$	$(x \quad y) \quad z$	$x \; (y \; k \; z)$
$(x + y) \mid z$	$= x \mid z + y \mid z$	$(x \quad \mid Ty) \quad Tz$	$= x \; T \mid (y \qquad z)$
$\mathbf{1} \mid \mathbf{1}$	$= \mathbf{1}$	x	$= x \quad y \; T$
$a.x \mid \mathbf{1}$	$= \mathbf{0}$	$\tau.$	$= \mathbf{0} \; T$
$!d.x \mid$	$= ?!d.(x \; k \; y)$	$yTx \mid T\tau.y$	
$?d.ya.x \mid$	$= \mathbf{0} \qquad$ if $\{a,b\} \; 6= \{!d,?d\}$		
$b.y$			
$\partial_*(\mathbf{0})$	$= \mathbf{0}$	$\tau_*(\mathbf{0}) \quad \tau_*(\mathbf{1})$	$= \mathbf{0}$
$\partial_*(\mathbf{1})$	$= \mathbf{1}$	$\tau_*(?!d.x)$	$= \mathbf{1}$
$\partial_*(!d.x)$	$= \partial_*(?d.x) = \mathbf{0}$	$\tau_*(a.x)$	$= \tau.\tau_*(x)$
$\partial_*(a.x)$	$= a.\partial_*(x) \qquad$ if $a 6\in \{!d,?d\}$	$\tau_*(x + y)$	$= a.\tau_*(x)$
$\partial_*(x + y)$	$= \partial_*(x) + \partial_*(y)$		$= \tau_*(x) + \tau_*(y \quad$ if $a6=$ $) \quad ?!d$

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

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

REFERENCES

[1] Chomsky, Noam. 1962. Context Free Grammar and Pushdown Storage.
[2] Waltrous, Ramond L. and Kuhn, Gary M. 1992. Induction of Finite-State Languages
[3] http://www.jflap.org/tutorial/pda/construct/
[4] Context-free grammars and push-down storage
[5] N. Chomsky, G.A. MillerFinite state languages
[6] Baeten, J.C.M., Bergstra, J.A., Klop, J.W.: Decidability of bisimulation equivalence for processes generating context-free languages. Journal of the ACM 40(3), 653–682 (1993)
[7] Seymour Ginsburg, Sheila A. Greibach and Michael A. Harrison (1967). "Stack Automata and Compiling



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY