Symbolic Model Checking with Rich Assertional Languages*

Y. Kesten** O. Maler*** M. Marcus† A. Pnueli† E. Shahar†

Abstract. The paper shows that, by an appropriate choice of a rich assertional language, it is possible to extend the utility of symbolic model checking beyond the realm of BDD-represented finite-state systems into the domain of infinite-state systems, leading to a powerful technique for uniform verification of unbounded (parameterized) process networks.
The main contributions of the paper are a formulation of a general framework for symbolic model checking of infinite-state systems, a demonstration that many individual examples of uniformly verified parameterized designs that appear in the literature are special cases of our general approach, verifying the correctness of the Futurebus+ design for all single-bus configurations, extending the technique to tree architectures, and establishing that the presented method is a precise dual to the top-down invariant generation method used in deductive verification.

1 Introduction

The problem of uniform verification of parameterized systems is one of the most thoroughly researched problems in computer-aided verification. The problem seems particularly elusive for systems that consist of regularly connected finite-state processes (a process network). Such a system can be verified for any given configuration, but this does not provide a conclusive evidence for the question of *uniform verification*, i.e., showing that the system is correct for *all* possible configurations.

We have had a recent experience with the Futurebus+ system, which has been verified for many configurations in [CGH+93] . Using the TLV system [PS96], we were able to analyze additional (and larger) configurations and detected a bug that escaped the previous verification efforts. Having corrected the bug, all of the configurations we have been able to check, verified correctly. However, the question of whether the Futurebus+ protocol in its last version contains another lurking bug, which makes its appearance only in a configuration much larger than anyone was able to check, still remains unresolved. One of our main motivations in the research reported in this paper is to develop a method by which uniform verification of parameterized designs such as the Futurebus+ can be algorithmically performed.

Many methods have been proposed for the uniform verification of parameterized systems. These include explicit induction ([EN95], [SG92]), network invariants, which can be viewed as implicit induction ([KM89], [WL89], [HLR92],

* This research was supported in part by a gift from Intel, and an *Infrastructure* grant from the Israeli Ministry of Science and the Arts.
** Future Cad Technology, Intel, Israel. E-mail: ykesten@iil.intel.com
** Verimag, Centre Equation, 2, av. de Vignate, 38610 Gières, France.
E-mail: Oded.Maler@imag.fr
† Weizmann Institute of Science, Rehovot, Israel.
E-mail: {monica, amir, elad}@wisdom.weizmann.ac.il

[LHR97]), methods that can be viewed as abstraction and approximation of network invariants ([BCG86], [SG89], [CGJ95]), and other methods which can be viewed as based on abstraction ([ID96], [EN96]).

In this methodologically simplistic paper, we go back to basics and claim that, with an appropriate choice of an expressive but decidable assertional language, the good old paradigm of symbolic model checking is adequate for uniform verification of parameterized systems. The paper demonstrates this claim by studying in detail symbolic model checking with the assertional languages of regular sets and tree regular sets. For the case of regular sets of strings, we show that many of the examples previously verified using specialized representations or additional theories, such as the examples considered in [CGJ95], [ID96], and [EN96], can be solved by this single and simple approach. The use of regular assertional tree languages is new (except for a brief mention in [HJJ+96]) and its application to a uniform verification of the Futurebus+ system will be a very convincing evidence to the power of the approach advocated here.

One of the inspirations to the work reported here was [CGJ95] (and its predecessor [SG89]), where regular languages was the main instrument used at the end. However, we strongly felt that, with some restrictions, the same verification capabilities can be obtained without the elaborate theory developed in [CGJ95]. In particular, we felt that there exists a redundancy between the *network grammar* used in [CGJ95] just to define the network topology and structure and the additional means for representing the dynamic behavior by another regular language. In our approach, we use a *single* regular language to describe both the topology and the local states of the participating processes. However, we cannot handle as general network topologies as are considered in [CGJ95], and must restrict ourselves to either array or tree topologies. The general principle is still applicable to other topologies but it requires the development of a different assertional language for each family of topologies.

By adopting the idea that a set of possible configurations of an unbounded array of processes can be represented as a set of strings over the process alphabet, we can go further and view the transitions of the system as *rewrite rules* applied to these strings. Hence the model-checking problem for networks can be reduced to the problem of calculating predecessors of a language via a rewriting system consisting of a finite set of *length-preserving* rules[5]. In [BM96], a technique for calculating the reachable states of an alternating push-down process (i.e. an automaton with one unbounded variable, a push-down stack) was presented and used in order to model-check such processes against μ-calculus formulae. This technique (inspired by the construction given in [BO93], pages 91-93) is based on representing a regular set L of stack configurations by an automaton A and then calculating the set of predecessors of L via a rewrite rule by modifying A. In the case of push-down processes the algorithm is guaranteed to converge, but experience shows that it converges in many other cases.

In this paper we generalize this idea in few directions. First, by using finite-state transducers we extend the technique to treat a more general class of rewrite

[5] If we ignore process creation and annihilation.

rules. We transfer the concept from theory to practice by implementing it into a working system and applying it successfully to several examples including all single-bus configurations of the Futurebus+. Secondly, we treat processes arranged in a tree architecture. To this end we define sets of process configurations as regular tree languages, employ bottom-up tree automata to represent them, and use tree transducers in order to define predecessors.

The implementation owes much to the MONA system and its underlying principles [HJJ+96]. Similar to MONA, we adopt an S1S-inspired language for the user interface with the system, which is then translated into finite automata represented with BDD-labeled edges. However, unlike some of the applications to verification reported in [HJJ+96] and [BK95], which are essentially deductive in nature, we use similar tools for symbolic model checking. A similar implementation for trees is in the making, with the intended goal of verifying the Futurebus+ for all multiple-bus configurations.

2 Symbolic Model Checking

In Fig. 1 we present the well-known symbolic model checking procedure for showing that the invariance property $\Box\, g$ ($\mathbf{AG}\, g$ in CTL) is satisfied by system P, where g is an assertion (state formula). This procedure was already formulated in the early 80's (see [CE81], [QS82], [CES86]). It became practical and widely usable only with implementations based on *ordered binary decision diagrams* (OBDDs) [Bry86], such as [BCM+92] and [McM93]. Procedure SYMB-MC attempts to compute an assertion characterizing all the states from which a $\neg g$-state can be reached by a finite number of P-steps. If the search loop terminates at iteration i, then φ_i provides such an assertion. By checking that none of the "bad" states characterized by φ_i are allowed as initial states of P, we verify that there is no $\neg g$-state reachable from a P-initial state, so g is an invariant of system P.

Procedure SYMB-MC(g: **assertion**);
  assertion: $\varphi_0, \varphi_1, \ldots$;
  Let $\varphi_0 := \neg g$;
  For $i = 0, 1, \ldots$ **repeat**
    Let $\varphi_{i+1} := \varphi_i \vee \mathit{pred}_P(\varphi_i)$;
  until $\varphi_{i+1} = \varphi_i$;
  Check that $\varphi_i \wedge \mathit{init}_P = \mathrm{F}$
end procedure

Fig. 1. A procedure for symbolic model checking.

The procedure uses the assertion init_P as a characterization of all the P-initial states, and the predicate transformer pred_P. For an assertion φ, $\mathit{pred}_P(\varphi)$ is an assertion characterizing all states that have a φ-state as a P-successor.

As recommended by the *rich-language symbolic model checking* (RSMC) methodology expounded in this paper, in order to verify that assertion g is an invariant of the (possibly infinite-state) system P, one chooses an assertional language $\mathcal{L}$ and uses it to apply the SYMB-MC procedure. To be applicable, the language $\mathcal{L}$ should satisfy the following minimal requirements:

- The property g and the assertion init_P should be expressible in $\mathcal{L}$.

- The language $\mathcal{L}$ should be effectively closed under the boolean operations of negation and disjunction, and possess an algorithm for deciding equivalence of two assertions.
- There should exist an algorithm for constructing the predicate transformer $pred_P : \mathcal{L} \mapsto \mathcal{L}$ for every system P.

We refer to a language satisfying these three requirements as a language *adequate for symbolic model checking*. Note that identifying an adequate assertional language only guarantees that Procedure SYMB-MC is applicable. It is still only a semi-algorithm which, when terminating, provides either proof of correctness or a counter example, but may fail to terminate. In fact, due to the theoretical results of [AK86], the invariance checking problem for parameterized systems is in general undecidable, and the best we can hope for in the general case is a semi-algorithm.

In the remaining sections, we will consider several useful adequate assertional languages and illustrate their application to parameterized systems of interest.

3 Regular Languages are Adequate

In this section we demonstrate the use of the class of regular languages as adequate assertional languages. As a running example, we will use program MUX of Fig. 2 that implements mutual exclusion by synchronous communication.

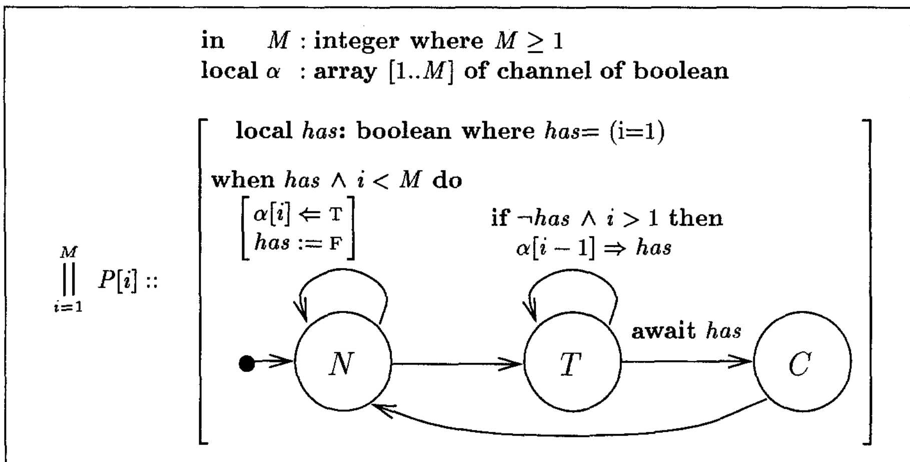

Fig. 2. Parameterized Program MUX.

The body of the program is a variable-size parallel composition of processes $P[1], \ldots, P[M]$. Each process $P[i]$ has two local state variables: a local boolean variable *has* and a control variable π ranging over the set of locations $\{N, T, C\}$ (the noncritical section, the trying section, and the critical section, respectively). Process $P[i]$ sends the boolean value T on channel $\alpha[i]$ to its right neighbor (if $i < M$) and reads into variable *has* a boolean value from its left neighbor on channel $\alpha[i-1]$ (if $i > 1$). As seen in the program, process $P[i]$ can enter its critical section only if $P[i].has = $ T.

A *local state* of process $P[i]$ is a valuation of the local state variables. For example, $\langle \pi : C,\ has : \text{T} \rangle$ is a local state in which $P[i]$ is in its critical section while its variable *has* has the value T. We abbreviate $\langle \pi : C,\ has : \text{T} \rangle$ to $\langle C, \text{T} \rangle$, listing just the values assigned to the variables.

A *global state* (also called a *configuration*) of system MUX, is a sequence of local states. Note that every configuration of system MUX can be viewed as a word over the finite alphabet

Σ_{MUX}: $\{\langle N, \text{F}\rangle, \langle T, \text{F}\rangle, \langle C, \text{F}\rangle, \langle N, \text{T}\rangle, \langle T, \text{T}\rangle, \langle C, \text{T}\rangle\}$.

Consequently, we can view a set of configurations of program MUX as a language over the alphabet Σ_{MUX}. Examining procedure SYMB-MC, we identify two languages and one language transformer which need to be syntactically characterized. We will consider each of these in turn.

3.1 Expressing the Initial Condition $init_P$ and the Desired Invariant g

Our general recommendation is to use as an assertional language for a *process array system* P, such as program MUX, the language of regular expressions over the alphabet Σ_P. A regular expression over Σ_P defines a language which characterizes a set of global states. For example, the initial condition for program MUX can be expressed by the regular expression

$init_{\text{MUX}}$: $\langle N, \text{T}\rangle(\langle N, \text{F}\rangle)^*$.

While we propose to use regular expressions in the user interface with the RSMC support system, the internal representation of the data structure "assertion" used in procedure SYMB-MC, is that of a finite-state automaton (FSA) over Σ_P. We consider such an automaton to be given by A: $\langle \Sigma_P, Q, q_0, \delta, F\rangle$, where Σ_P is the input alphabet, Q is the set of *automaton states*, $q_0 \in Q$ is the *initial automaton state*, $\delta: Q \times \Sigma_P \mapsto 2^Q$ is the *transition function*, and $F \subseteq Q$ is the set of *accepting states*.

Next, we consider the desired property g. For the case of program MUX, the required property is that of *mutual exclusion* requiring that at most one process reside in its critical section at any given instance. This property can be expressed by the regular expression

g: $[x \neq C]^+ + [x \neq C]^*[x = C][x \neq C]^*$,

where we use the abbreviations $[x = C] = \langle C, \text{T}\rangle + \langle C, \text{F}\rangle$ and $[x \neq C] = \Sigma_{\text{MUX}} - [x = C]$.

3.2 Expressing the $pred_P$ Transformer

To express the $pred_P$ transformer, we first attempt to describe the change in configurations as a result of a single program step. Consider our running example, program MUX. The (parameterized) fair transition system [MP95] corresponding to this program has two kinds of transitions. There are transitions that affect only a single process and represent internal movements and variable changes within this process. The other kind is the transition that involves two contiguous processes, i.e., $P[i]$ and $P[i+1]$ for some $i \in \{1, M-1\}$. This transition corresponds to the synchronous communication in which process $P[i]$ sends the boolean value T, which process $P[i+1]$ receives and stores into *has*.

We can summarize the transformation effected by the various transitions by the following list of *rewrite rules*:

$$U: \left\{ \begin{array}{lll} \langle N, \text{F}\rangle \to \langle T, \text{F}\rangle, & \langle C, \text{F}\rangle \to \langle N, \text{F}\rangle, & \langle T, \text{T}\rangle \to \langle C, \text{T}\rangle \\ \langle N, \text{T}\rangle \to \langle T, \text{T}\rangle, & \langle C, \text{T}\rangle \to \langle N, \text{T}\rangle & \end{array} \right\}$$

$$M: \{\langle N, \text{T}\rangle\, \langle T, \text{F}\rangle \to \langle N, \text{F}\rangle\, \langle T, \text{T}\rangle\}$$

where U (the unary rewrites) represents changes that affect only a single process, while M is a binary rewrite rule representing a joint transition of two contiguous processes. For example, applying the rewrite rule $\langle N, \mathrm{T}\rangle\,\langle T, \mathrm{F}\rangle \rightarrow \langle N, \mathrm{F}\rangle\,\langle T, \mathrm{T}\rangle$ to the configuration $\langle N, \mathrm{T}\rangle\,\langle T, \mathrm{F}\rangle\,\langle N, \mathrm{F}\rangle$ yields the successor configuration $\langle N, \mathrm{F}\rangle\,\langle T, \mathrm{T}\rangle\,\langle N, \mathrm{F}\rangle$, representing the result of passing a token from $P[1]$ to $P[2]$.

A precise characterization of the transformation caused by each of these rewrite rules can be provided by a *finite-state transducer* (FST) $\mathcal{T}$: $\langle \Sigma_P \times \Sigma_P, Q, q_0, \delta, F\rangle$, which is an FSA over the alphabet

$$\Sigma_P \times \Sigma_P = \{[a, b] \mid a, b \in \Sigma_P\}.$$

Let $u = a_1 \cdots a_k$ and $v = b_1 \cdots b_k$ be two Σ_P-words of equal length. We define their *cross product* $u \times v$ to be the $\Sigma_P \times \Sigma'_P$-word $([a_1, b_1] \cdots [a_k, b_k])$. We say that word v is a *transduction* of word u by the FST $\mathcal{T}$ if the cross word $u \times v$ is accepted by $\mathcal{T}$.

Consider the FST $\mathcal{T}_2$ presented in Fig. 3. The label *id* appearing in the transducer stands for the set $\{(a, a) \mid a \in \Sigma_{\mathrm{MUX}}\}$, representing the identity transformation. The transducer $\mathcal{T}_2$ represents the rewrite rule $\langle N, \mathrm{T}\rangle\,\langle T, \mathrm{F}\rangle \rightarrow \langle N, \mathrm{F}\rangle\,\langle T, \mathrm{T}\rangle$. For example, the configuration $v = \langle N, \mathrm{F}\rangle\,\langle T, \mathrm{T}\rangle\,\langle N, \mathrm{F}\rangle$ is a $\mathcal{T}_2$-transduction of the configuration $u = \langle N, \mathrm{T}\rangle\,\langle T, \mathrm{F}\rangle\,\langle N, \mathrm{F}\rangle$, because the joint word

$$u \times v \quad = \quad ([\langle N, \mathrm{T}\rangle, \langle N, \mathrm{F}\rangle]\;[\langle T, \mathrm{F}\rangle, \langle T, \mathrm{T}\rangle]\;[\langle N, \mathrm{F}\rangle, \langle N, \mathrm{F}\rangle])$$

is accepted by $\mathcal{T}_2$.

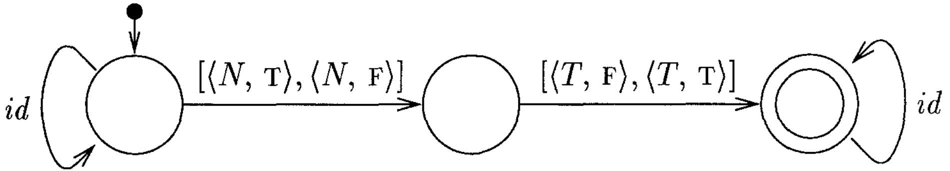

Fig. 3. Transducer $\mathcal{T}_2$ representing a two-state rewrite.

In a similar way, we can construct a transducer corresponding to each of the remaining rewrite rules, expressing the effect of a single transition in program MUX. Since the class of regular languages is closed under union, it is possible to construct a *single* transducer $\mathcal{T}_{\mathrm{MUX}}$ such that the configuration v is a $\mathcal{T}_{\mathrm{MUX}}$-transduction of a configuration u iff v can be obtained from u by a single step of program MUX. We refer to $\mathcal{T}_{\mathrm{MUX}}$ as the *step transducer* for program MUX.

Given a transducer $\mathcal{T} = \langle \Sigma \times \Sigma, R, r_0, \delta_T, F_T\rangle$ and an FSA $A = \langle \Sigma, Q, Q_0, \delta_A, F_A\rangle$, we define their *composition* to be the automaton

$$\mathcal{T} \circ A = \langle \Sigma, R \times Q, [r_0, q_0], \delta, F_T \times F_A\rangle,$$

where $[r_2, q_2] \in \delta([r_1, q_1], a)$ iff there exists a $b \in \Sigma$ such that $r_2 \in \delta_T(r_1, [a, b])$ and $q_2 \in \delta_A(q_1, b)$.

It is possible to establish the following claim:

Claim 1 *The language accepted by the composition $\mathcal{T} \circ A$ consists of all words having a $\mathcal{T}$-transduction which is accepted by A.*

Going back to the use of FSAs as an assertional language, we observe that if A is an automaton characterizing a set of states of system P and $\mathcal{T}_P$ is the step transducer for P, then the precondition transformer $pred_P$ required in procedure SYMB-MC is given by $pred_P(A) = \mathcal{T}_P \circ A$.

3.3 Applicability of FSAs as Assertional Languages

We can summarize the previous discussions by the following claim:

Claim 2 *If P is a system with an encoding of its global states into words over an alphabet Σ_P, such that*

- *the initial condition $init_P$ and the goal assertion g can be represented by Σ_P-automata, and*
- *the global transition relation of P can be represented by a step transducer $\mathcal{T}_P$,*

then procedure SYMB-MC *can be applied to the verification of $P \models \Box g$, using* FSA*'s as the assertional language.*

Claim 2 does not guarantee that the application of SYMB-MC will terminate.

We have constructed an implementation of a system that accepts as inputs the automata representing $init_P$ and g and the step transducer $\mathcal{T}_P$, and checks whether g is a P-invariant, although it may fail to terminate. We managed to verify program MUX and other simple programs including versions of MUX with either synchronous or asynchronous communication where the processes are arranged in a ring rather than an array. Finally, two of the four safety specifications which were verified in [CGH+93] and [PS96] were checked for a single-bus version of the Futurebus+ protocol and were found to be correct.

The representation of automata in our implementation uses OBDD-encoded assertions over the local state variables instead of explicit enumeration of the local states which allow a transition from one automaton state to another. Thus, our transition function has the type δ: $Q \times local_assertions \mapsto 2^Q$, where a local assertion is an assertion over the local state variables.

4 Tree Languages

In this section, we extend the method of regular expressions over strings to deal with regular tree languages (see [TW68], [GS70], [D70]). This will enable us to handle process networks organized in a tree topology. Since process trees may have different out-degrees for different nodes, we have to generalize the notion of tree automata to deal with varying arity.[6]

4.1 Bottom-Up Tree Automata

We define a *tree structure* S to be a finite subset of $\mathbb{N}^*$ (i.e. a finite set of sequences of natural numbers) satisfying:

- S contains the empty sequence Λ.
- If S contains the sequence $(\alpha_1, \ldots, \alpha_k)$, then it also contains the (possibly empty) sequence $(\alpha_1, \ldots, \alpha_{k-1})$ and the sequences $(\alpha_1, \ldots, \alpha_{k-1}, r)$, for every r, $0 \leq r < \alpha_k$.

We refer to the elements of S as the *nodes* of the tree structure S. Obviously, S represents a node by specifying the path that has to be followed from the root in order to get to the node. Thus, in a tree structure, Λ represents the root, and

[6] An extension of tree automata to arbitrary arity was made in [KG96] but in a top-down infinite context.

$(1,0)$ represents the node which is the first child of the second child of the root. A node $\overline{\alpha} \in S$ is a leaf, if it is not a prefix of any other member of S.

Let A be an arbitrary alphabet, i.e. a finite set of symbols. An *A-tree* T: $\langle S, \lambda \rangle$ consists of a tree structure S and a *labeling function* $\lambda: S \mapsto A$, mapping each node of the tree to an A label. We will often refer to nodes in the tree as $n \in T$ and to their labels as $\lambda(n)$.

A *(variable-arity) bottom-up tree automaton* (BTA) B: $\langle \Sigma, Q, \Delta, F \rangle$ where Σ, Q and $F \subseteq Q$ are the standard finite *alphabet*, set of *states*, and set of *accepting states*, while

$$\Delta: Q^* \times \Sigma \mapsto 2^Q$$

is a regular *transition function*, i.e. for every $a \in \Sigma$ and $\widetilde{Q} \subseteq Q$, the set of words $\{w \in Q^* \mid \Delta(w,a) = \widetilde{Q}\}$ is regular. In our presentations of BTAs, we write Δ as a finite number of entries of the form $\Delta(E_i, \Sigma_i) = Q_i$, where E_i is a regular expression over Q, $\Sigma_i \subseteq \Sigma$, and $Q_i \subseteq Q$ indicating that for $q \in Q$, $w \in Q^*$, and $a \in \Sigma$, $q \in \Delta(w,a)$ iff $q \in \Delta(E_i, a)$ for some E_i such that $w \in L(E_i)$.

The way a BTA operates when applied to a Σ-tree T is that it proceeds from the leaves towards the root, annotating the tree nodes with automaton states. A single annotation step can be applied to the tree node $n \in T$ only when all of its children have been already annotated. Assume that the children of n have been annotated with $q_1, \ldots, q_k$. Then, n can be annotated by $q \in Q$ if $q \in \Delta(q_1 \cdots q_k, \lambda(n))$.

More formally, a *run* of the BTA B over the tree $T = \langle S, \lambda \rangle$ is a mapping $r: S \mapsto Q$ satisfying:

For each $n \in S$ with children $n_1, \ldots, n_k$, $r(n) \in \Delta(r(n_1) \cdots r(n_k), \lambda(n))$.

A BTA is *deterministic* if $|\Delta(w,a)| = 1$, for every $w \in Q^*$ and $a \in \Sigma$.

Example: Let us define a BTA B which recognizes all variable-arity trees, labeled by $\Sigma = \{a, b\}$, with the requirement that precisely one node is labeled by b. For the components of B, we choose as follows: $\Sigma : \{a,b\}$, $Q : \{q_0, q_1, q_2\}$, $F : \{q_1\}$,

Δ : Defined as follows:

$$\begin{array}{lll} \Delta(q_0^*, a) = & & \{q_0\} \\ \Delta(q_0^*, b) = & \Delta(q_0^* q_1 q_0^*, a) = & \{q_1\} \\ \multicolumn{3}{l}{\Delta(Q^* q_1 Q^* q_1 Q^*, \{a,b\}) = \Delta(Q^* q_2 Q^*, \{a,b\}) = \Delta(q_0^* q_1 q_0^*, b) = \{q_2\}} \end{array}$$

The BTA B is obviously deterministic. Given an $\{a,b\}$-tree T, automaton B will annotate by q_0 all the nodes n such that the subtree rooted at n is only labeled by a. Nodes rooting a subtree such that precisely one node in the subtree is labeled by b will be annotated by q_1. All other nodes are annotated by q_2. The tree T is accepted by B iff its root is annotated by q_1.

The transition function Δ determines the annotation of a node n, based on the annotation of its children and the Σ-character labeling n. According to the table, n will be annotated by q_0 if all its children are annotated by q_0 and n's label is a. This also takes care of the a-labeled leaves, since the empty word belongs to the language q_0^*. Node n will be annotated by q_1 if either all children are annotated by q_0 and n is labeled by b, or all children are annotated by q_0 except for one child which is annotated by q_1 and n is labeled by a. In all other cases, n will be annotated by q_2 which implies that at least two b's have been detected in the tree. ∎

A tree T is said to be *accepted* by the BTA B if there exists a run r of B over T such that $r(\Lambda) \in F$. We denote by $L(B)$ the set of trees accepted by B. The BTAs B_1 and B_2 are said to be *equivalent* if $L(B_1) = L(B_2)$. By applying the standard subset construction, we can establish the following claim:

Claim 3

1. *Every* BTA *is equivalent to a deterministic* BTA.
2. *The class of tree languages recognizable by a* BTA *is closed under the boolean operations of complementation and union.*

4.2 Configurations of a Process Tree as a Tree Language

As a running example, consider program PERCOLATE of Fig. 4. The assertion $leaf(\alpha, S)$ holds for tree address $\alpha \in S$ iff α is a leaf of S.

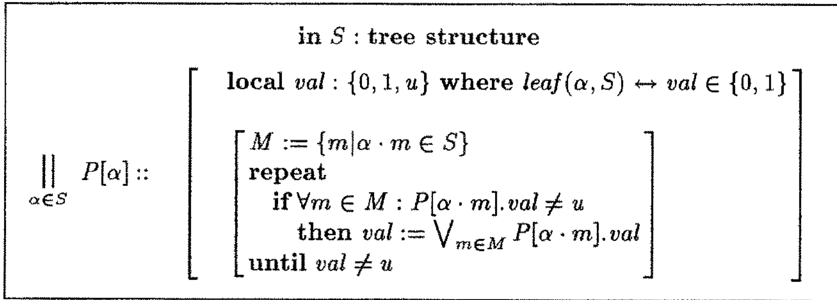

Fig. 4. Process tree program PERCOLATE.

Program PERCOLATE consists of a tree of processes, each having its local variable val, which ranges over the set of values $\{0, 1, u\}$. The value u should be interpreted as "undefined yet", which implies that it will eventually change to either 0 or 1. Initially, all the leaf processes in the tree have $val \in \{0, 1\}$ and all other processes have $val = u$. The purpose of program PERCOLATE is to percolate to the root of the tree a value 1 if at least one of the leaves has value 1, and a value of 0, if all leaves have value 0. If $P[\alpha]$ does not yet have a defined value but all its childrens' values are defined then $P[\alpha]$ sets its value to the disjunction of the values of its children. Consequently, we can represent a configuration of program PERCOLATE as a tree over the alphabet $\Sigma_{\text{PERCOLATE}}$: $\{0, 1, u\}$.

The specification of program PERCOLATE can be given by the formula

g: $\quad P[\Lambda].val \neq u \;\rightarrow\; \big(P[\Lambda].val = \bigvee_{leaf(\alpha,S)} P[\alpha].val\big)$.

This formula states that if the root has a defined value then its value equals the disjunction of all val values at the leaves. It is not difficult to construct a BTA which will accept precisely the trees that have the property specified by g. In a similar way, it is straightforward to construct a BTA which will accept the initial configurations of the program.

To complete the demonstration that the assertional language of BTA's is adequate for symbolic model checking of program PERCOLATE, we should specify a tree transducer that will represent the state transformations due to execution of statements within the individual processes.

Let $T_1 = \langle S, \lambda_1 \rangle$ and $T_2 = \langle S, \lambda_2 \rangle$ be two Σ-trees over the same tree structure S. These can be viewed as two different labeling of the same underlying tree. We define the *cross product* of T_1 and T_2 as the $\Sigma \times \Sigma$ tree $T_1 \times T_2 = \langle S, \lambda_\times \rangle$, where, for each $\alpha \in S$, $\lambda_\times(\alpha) = [\lambda_1(\alpha), \lambda_2(\alpha)]$.

A *tree transducer* (over Σ) is simply a BTA over the product alphabet Σ^2. For trees T_1 and T_2 as described above, we say that T_2 is a $\mathcal{T}$-transduction of T_1 if the tree $T_1 \times T_2$ is accepted by $\mathcal{T}$.

Example: A tree transducer that represents the single transition (parameterized by the process address α) of program PERCOLATE is defined as follows:

Σ: $\Sigma_{\text{PERCOLATE}} \times \Sigma_{\text{PERCOLATE}}$ $\qquad$ Q: $\{\underbrace{q_0, q_1, q_u}_{Q_n}, q_d\}$ $\qquad$ F: $\{q_d\}$

Δ : Defined as follows:

$$\Delta(Q_n^*, [0,0]) = \{q_0\} \qquad \Delta(Q_n^*, [1,1]) = \{q_1\} \qquad \Delta(Q_n^*, [u,u]) = \{q_u\}$$
$$\Delta(q_0^*, [u,0]) = \Delta((q_0+q_1)^* q_1 (q_0+q_1)^*, [u,1]) = \Delta(Q_n^* q_d Q_n^*, id) = \{q_d\}$$

The transducer uses four states. Annotation of node α by the automaton states Q_n: $\{q_0, q_1, q_u\}$ reflects the value of $P[\alpha].val$ and also implies that in the subtree rooted at α, all the Σ^2 labels are the identity id. Annotation of α by q_d such that no descendant of α is annotated by q_d identifies the only allowed node in the tree structure which is labeled by a Σ^2-character different from id. The rules for such annotations are given by the second line in the definition of Δ. This line allows a change of value from u to 0 if all the children of α are annotated by q_0. It allows a change of value from u to 1 if at least one of the descendants is annotated by 1 and all the rest are annotated by 0 or 1.

Once the first (lowest) node is annotated by q_d, this annotation propagates from each node to its parent, provided none of the siblings is annotated by q_d. This guarantees that only one process in the tree changes its value from u to 0 or 1. ∎

Given a tree transducer $\mathcal{T} = \langle \Sigma \times \Sigma, R, \delta_{\mathcal{T}}, F_{\mathcal{T}} \rangle$ and a BTA $A = \langle \Sigma, Q, \delta_A, F_A \rangle$, we define their *composition* to be the BTA

$$\mathcal{T} \circ A = \langle \Sigma, R \times Q, \delta, F_{\mathcal{T}} \times F_A \rangle,$$

where, for every $r \in R$, $q \in Q$, $v \in R^*$, and $w \in Q^*$,

$$[r,q] \in \delta(v \times w, a) \quad \text{iff} \quad \exists b \in \Sigma \text{ such that } r \in \delta_{\mathcal{T}}(v, [a,b]) \text{ and } q \in \delta_A(w, b).$$

Claim 4 *The tree language accepted by the composition $\mathcal{T} \circ A$ consists of all trees having a $\mathcal{T}$-transduction which is accepted by A.*

Going back to the use of BTAs as an assertional language, we observe that if A is a BTA characterizing a set of configurations of system P and $\mathcal{T}_P$ is the step tree transducer for P, then the precondition transformer $pred_P$ required in procedure SYMB-MC is given by $pred_P(A) = \mathcal{T}_P \circ A$.

5 Symbolic Model Checking is Dual to Invariant Generation

An important component in all the modern support systems for deductive verification, such as STeP [MAB+94] and PVS [SOR93], consists of algorithms and heuristics for the automatic generation of invariants. Several of these techniques have been presented in [MP95] and efficiently implemented as reported in [BBM95]

and [BLS96]. Perhaps the most powerful and widely applicable is the technique called *top-down invariant generation*. As described in [MP95] and [BBM95], the method starts with a goal assertion g, whose invariance we wish to prove, and applies a series of strengthening steps, until we obtain a stronger assertion ψ which implies g and is *inductive*. Using our notation, the strengthening procedure can be described as in Fig. 5. The predicate transformer $pred_P^\forall$ appearing in the procedure is dual to the $pred_P$ transformer used in procedure SYMB-MC of Fig. 1. It can be defined either by the duality relation $pred_P^\forall(\psi) = \neg pred_P(\neg\psi)$, or by saying that a state s satisfies $pred_P^\forall(\psi)$ iff *all*[7] P-successors of s satisfy ψ.

Procedure STRENGTHEN(g: **assertion**);
  assertion: $\psi_0, \psi_1, \ldots$;
  Let $\psi_0 := g$;
  For $i = 0, 1, \ldots$ **repeat**
    Let $\psi_{i+1} := \psi_i \wedge pred_P^\forall(\psi_i)$;
  until $\psi_{i+1} = \psi_i$;
  Check that $init_P \rightarrow \psi_i$
end procedure

Fig. 5. A procedure for top-down invariant generation.

Procedure STRENGTHEN is a perfect dual of procedure SYMB-MC. One of the procedures terminates iff the other does and, when they terminate, they terminate after precisely the same number of steps. Furthermore, for every $i = 0, 1, \ldots$, reached in the application of these procedures, $\psi_i = \neg\varphi_i$, and one of them reports success (implying that g is a P-invariant) iff the other does.

So presenting the considered procedure as symbolic model checking or as part of the deductive set of tools is a matter of taste. The successful verification cases reported in [BBM95] and [BLS96] will work equally well in the approach of symbolic model checking suggested here. Symmetrically, it shows that the two assertional languages of regular languages and regular tree languages analyzed here can be imported into the invariant generation methodology with equal success.

References

[AK86] K. R. Apt and D. Kozen. Limits for automatic program verification of finite-state concurrent systems. *Information Processing Letters*, 22(6), 1986.

[BBM95] N. Bjørner, I.A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In *LNCS 976*, 1995.

[BO93] R.V. Book and F. Otto. *String-Rewriting Systems*, Springer, 1993.

[BM96] A. Bouajjani and O. Maler, Reachability Analysis of Push-down Automata. *Workshop on Infinite-state Systems*, Pisa, 1996.

[BCG86] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many finite state processes. In *PODC'86.*

[BCM+92] J.R. Burch, E.M. Clarke, et al. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BK95] D.A. Basin and N. Klarlund. Hardware verification using 2nd-order logic. In P. Wolper, editor, *CAV'95, LNCS 939*, 1995.

[7] In comparison, s satisfies $pred_P(\psi)$ iff *some* P-successor of s satisfies ψ.

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. *CAV'96*, 1996.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[CE81] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Logics of Programs, LNCS 131*, 1981.

[CES86] E.M. Clarke, et al. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, 1986.

[CGH+93] E.M. Clarke, O. Grumberg, et al. Verification of the futurebus+ cache coherence protocol. *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications.* 1993.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks using abstraction and regular languages. In *CONCUR'95*, 1995.

[D70] J. Doner. Tree Acceptors and some of their applications, *JCSS* 4, 1970.

[EN95] E. A. Emerson, K. S. Namjoshi. Reasoning about rings. *POPL'95*, 1995.

[EN96] E.A. Emerson and K.S. Namjoshi. Automatic verification of parameterized synchronous systems. In R. Alur and T. Henzinger, editors, *CAV'96*, 1996.

[GS70] F. Gecseg and M. Steinby. *Tree Automata* Akademiai Kiado, 1984.

[HLR92] N. Halbwachs, F. Lagnier, et al. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7), 1992.

[HJJ+96] J.G. Henriksen, J. Jensen, et al. Mona: Monadic second-order logic in practice. In *TACAS '95, LNCS 1019*, 1996.

[ID96] C.N. Ip and D. Dill. Verifying systems with replicated components in Murφ. In R. Alur and T. Henzinger, editors, *CAV'96*, 1996.

[KG96] O. Kupferman and O. Grumberg. Branching Time Temporal Logic and Amorphous Tree Automata. *Information and Computation* 125, 1996.

[KM89] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In P. Rudnicki, editor, *PODC'89*, Edmonton, AB, Canada, 1989.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *POPL'97*, Paris, 1997.

[MAB+94] Z. Manna, A. Anuchitanukul, et al. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Stanford University, 1994.

[McM93] K.L McMillan. *Symbolic Model Checking.* 1993.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, 1995.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In R. Alur and T. Henzinger, editors, *CAV'96*, 1996.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in *cesar*. *International Symposium on Programming,LNCS 137*, 1982.

[SG89] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In *LNCS 407*, 1989.

[SG92] A.P. Sistla and S.M. German. Reasoning about systems with many processes. *J. ACM*, 39:675–735, 1992.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual (draft). Technical report, SRI International, Menlo Park, CA, 1993.

[TW68] J.W. Thatcher, J.B. Wright. Generalized finite automata with application to a decision procedure in second order logic. *Math. Sys. Theory* 2, 1968.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *LNCS 407*, 1989.