Symbolic Model Checking with Rich Assertional Languages [1]

Y. Kesten [a], O. Maler [b], M. Marcus [c], A. Pnueli [c,2], E. Shahar [c]

[a] *Department of Communication Systems Engineering, Ben-Gurion University, Beer-Sheva, Israel.*

[b] *Verimag, Centre Equation, 2, av. de Vignate, 38610 Gières, France.*

[c] *Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel.*

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, and extending the technique to tree architectures.

Keywords: Symbolic model checking; Parametric systems; Tree automata; Regular expressions

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 in the case of systems that consist of regularly connected finite-state processes (a process network). Such a system can be

[1] This research was supported in part by a gift from Intel, and an *Infrastructure* grant from the Israeli Ministry of Science and the Arts.

[2] Corresponding author. E-mail: amir@wisdom.weizmann.ac.il

model checked 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.

In fact, we have had a recent experience with the Futurebus+ system, which has been model checked for many configurations in [CGH$^+$93] and pronounced correct. 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 individually, 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], [LHR97]), methods that can be viewed as abstraction and approximation of network invariants ([BCG86], [SG89], [CGJ95]), and other methods that 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, it was felt that, with some restrictions, the same verification capabilities can be obtained without the elaborate theory developed in [CGJ95]. In particular, in some contexts it may be possible to unify the *network grammar* used in [CGJ95] to define the network topology and structure and the regular language used for representing the dynamic behavior of individual processes. In our approach, we use a *single* regular language to describe both the topology and the local states of the member processes. There is of course a price to pay which, in our case, is that 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 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 [3]. 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 a process 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 a finitary version of the logic WS1S we extend the technique to treat a more general class of rewrite 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 versions 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, and employ bottom-up tree automata to represent them.

The implementation owes much to the MONA system and its underlying principles [HJJ+96]. Similar to MONA, we adopt an WS1S-derived 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. We have also implemented a similar tool for trees.

[3] If we ignore process creation and annihilation.

2 Symbolic Model Checking

In Fig. 1 we present the well-known symbolic model checking procedure for showing that the invariance property $\square\, g$ ($\mathbf{AG}\, g$ 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 [QS82], [EC82], [CES86]), however, 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, and therefore that 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 $\mathit{pred}_{P}\colon \mathcal{L} \mapsto \mathcal{L}$ for 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 heuristic 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 heuristic.

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

3 The Logic FS1S

We use the logic FS1S, (*finitary second-order theory of one successor*) as a specification language for sets of global states of parameterized systems. This logic is derived from the *weak second order logic of one successor* [Tho90] and also resembles the language M2L used in MONA [HJJ+96]. It is well know that this logic has the expressive power of regular expressions, as well as finite automata which are the representation underlying our implementation. Following is a brief definition of the logic.

Syntax

We assume a *signature* $\Xi : \{\Sigma_1, \ldots, \Sigma_k\}$ consisting of a finite set of finite alphabets. The *vocabulary* consists of *position variables* $p_1, p_2, \ldots$ and, for each $\Sigma_i \in \Xi$, a set of Σ_i*-array variables* $X_i, Y_i, Z_i, \ldots$.

- Position (First-order) terms:
  - The constant 0.
  - Any position variable p_i.
  - $t + 1$, where t is a position term.
- Letter terms:
  - Every $a \in \Sigma_i$ is a Σ_i-term.
  - If X is a Σ_i-array variable and t is a position term, then $X[t]$ is a Σ_i-term.
- Atomic Formulas:
  - $t_1 \sim t_2$, where t_1 and t_2 are position terms and $\sim \in \{=, <\}$.
  - $x = y$, where x and y are Σ_i-terms for some $\Sigma_i \in \Xi$.
- Formulas:
  - An atomic formula is a formula.
  - Let φ and ψ be formulas. Then $\neg\varphi$, $\varphi \vee \psi$, $\exists p \cdot \varphi$, $\exists X \cdot \varphi$ are formulas, where p is a position variable and X is an array variable.

Semantics

Let ψ be an FS1S formula. A *model* for ψ is given by $M = \langle N, \nu, \mathcal{A}\rangle$, where $N > 0$ is a positive integer, ν assigns to each position variable p a natural number $\nu(p) \in [0..N-1]$, and $\mathcal{A}$ assigns to each Σ_i-array variable X a Σ_i-word $\mathcal{A}(X) \in \Sigma_i^N$ of size N.

Given a model $M = \langle N, \nu, \mathcal{A}\rangle$, we inductively define the *interpretation* $I_{_M}$ induced by M as follows:

- $I_{_M}$ interprets every *position term* t into a natural number $I_{_M}(t) \in [0..N-1]$, as follows:
  - The constant symbol 0 is interpreted as the natural number 0.
  - For a position variable p, $I_{_M}(p) = \nu(p)$.
  - $I_{_M}(t+1) = I_{_M}(t) + 1$ modulo N
- A Σ_i term is interpreted into a Σ_i-letter, as follows:
  - The constant symbol $a \in \Sigma_i$ is interpreted into the Σ_i-letter a.
  - If $\mathcal{A}(X) = a_0, \ldots, a_{N-1}$ and $I_{_M}(t) = j \in [0..N-1]$, then $I_{_M}(X[t]) = a_j$.
- Formulas are interpreted into boolean values ($\{0, 1\}$), as follows:
  - For position terms t_1 and t_2, $I_{_M}(t_1 \sim t_2)$ evaluates to 1 if the relation $\sim \in \{=, <\}$ holds between $I_{_M}(t_1)$ and $I_{_M}(t_2)$.
  - For Σ_i terms x and y, $I_{_M}(x = y)$ evaluates to 1 if $I_{_M}(x)$ equals $I_{_M}(y)$.
  - $\neg\varphi, \varphi \vee \psi, \varphi \wedge \psi, \varphi \leftrightarrow \psi, \varphi \rightarrow \psi$ — where φ and ψ are formulas, are interpreted in the standard way, after the formulas φ and ψ are interpreted.
  - $\exists p \cdot \varphi$ — is true if there exists a model $M' = \langle N, \nu', \mathcal{A}\rangle$, such that ν and ν' differ at most in the interpretation of the position variable p, and such that $I_{_{M'}}(\varphi) = 1$.
  - $\exists X \cdot \varphi$ — is true if there exists a model $M' = \langle N, \nu, \mathcal{A}'\rangle$, such that $\mathcal{A}$ and $\mathcal{A}'$ differ at most in the interpretation of the array variable X, and such that $I_{_{M'}}(\varphi) = 1$.

4 The Logic FS1S is Adequate

In this section we demonstrate the use of FS1S as an adequate assertional language. As a running example, we will use program MUX of Fig. 2 which implements mutual exclusion by synchronous communication.

The body of the program is a variable-size parallel composition of processes $P[0], \ldots, P[M-1]$. Each process $P[i]$ has two local state variables: a local boolean variable *has* whose initial value is T for $i = 0$ and F for all other processes, 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).

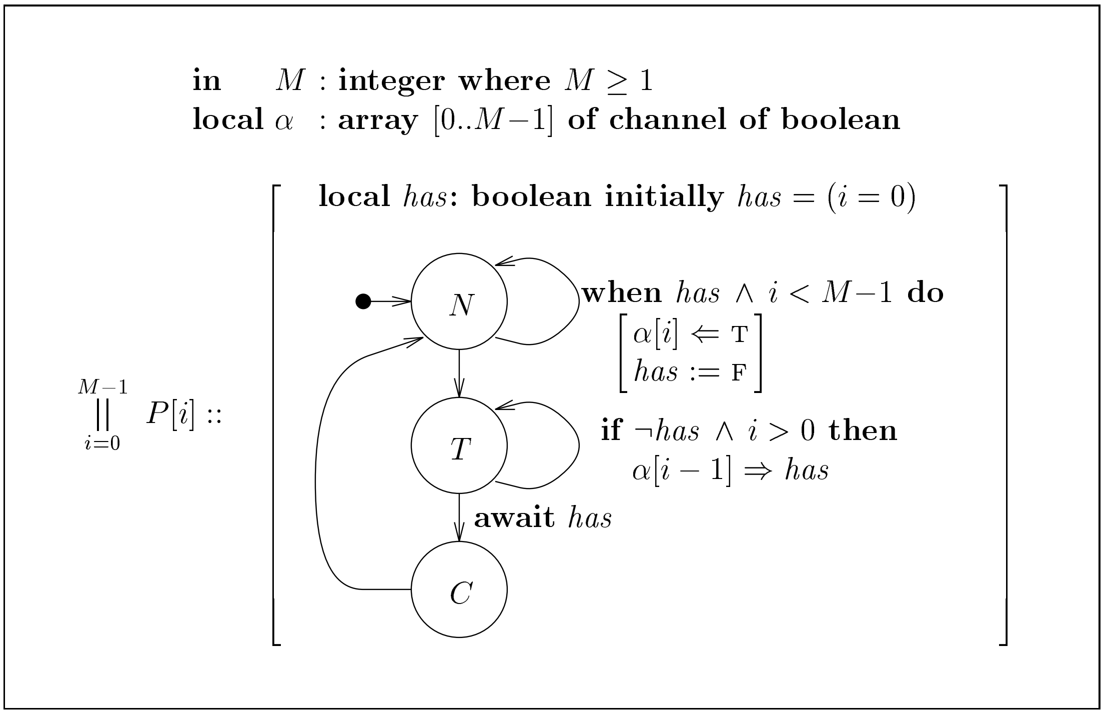


Fig. 2. Parameterized Program MUX.

Process $P[i]$ sends the boolean value T on channel $\alpha[i]$ to its right neighbor (if $i < M - 1$) and reads into variable *has* a boolean value from its left neighbor on channel $\alpha[i-1]$ (if $i > 0$). As seen in the program, process $P[i]$ can enter its critical section only if $P[i].has = \mathrm{T}$.

Our verification goal is to establish that, at any point in the execution, at most one process resides in its critical section.

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

A *global state* (also called a *configuration*) of system MUX is a sequence of local states. For example, the configuration $\langle N,\ \mathrm{T}\rangle\,\langle N,\ \mathrm{F}\rangle\,\langle N,\ \mathrm{F}\rangle$ represents the initial global state of system MUX for the case of $M = 3$. Namely, there are 3 processes, all in their non-critical locations, such that $P[0].has = \mathrm{T}$ and $P[1].has = P[2].has = \mathrm{F}$.

To specify the global state of system MUX in FS1S, we define two alphabets, $\Sigma_1 : \{N, T, C\}$, representing possible valuations of the variable π and $\Sigma_2 : \{\mathrm{F}, \mathrm{T}\}$ representing the domain of variable *has*. We use the Σ_1-array variable Π to store the values of π, and the Σ_2-array variable H to store the values of the local variable *has*, for all processes. The size of both arrays is determined by the number of processes in the verified system. For example, the global valuation

$$\begin{aligned} \Pi &: [N, N, N] \\ H &: [\text{T},\ \text{F},\ \text{F}] \end{aligned}$$

represents the initial global state of system MUX for the case of $M = 3$.

Examining procedure SYMB-MC, we identify two assertions and one predicate transformer which need to be syntactically characterized. We will consider each of these in turn.

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

In the following examples, for an array variable X defined over a boolean alphabet, we use the shorthand notations $X[i]$ for $X[i] = \text{T}$ and $\neg X[i]$ for $X[i] = \text{F}$.

The initial condition for program MUX can be expressed by the FS1S formula

$$init_{\text{MUX}}: \quad \forall i \cdot \Pi[i] = N \wedge (H[i] \leftrightarrow i = 0)$$

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 resides in its critical section at any given instance. This property can be expressed by the FS1S formula

$$g: \quad \neg \exists i, j \cdot i \neq j \wedge \Pi[i] = C \wedge \Pi[j] = C$$

4.2 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 [MP91] corresponding to program MUX 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 of transition involves two contiguous processes, i.e., $P[i]$ and $P[i+1]$ for some $0 \le i \le M-2$. 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*.

Transitions are specified by an assertion $\rho(V, V')$, relating a global-state s to its successor s' by referring to both unprimed and primed versions of the configuration variables. An unprimed version of a configuration variable refers to its value in s, while a primed version of the same variable refers to its value in s'.

We can express both types of transitions by the following FS1S formulas:

$$\rho_s(V,V'):\quad \exists i \cdot \begin{pmatrix} \quad \Pi[i]=N \ \wedge\ \Pi'[i]=T \\ \vee\ \Pi[i]=C \ \wedge\ \Pi'[i]=N \\ \vee\ \Pi[i]=T \ \wedge\ \Pi'[i]=C \ \wedge\ H[i] \end{pmatrix} \wedge\ pres_{\neq i}(\Pi) \ \wedge\ pres(H)$$

and

$$\rho_c(V,V'):\quad \exists i \cdot \begin{pmatrix} \quad i+1 \neq 0 \\ \wedge\ \Pi[i]=N \ \wedge\ \Pi[i+1]=T \\ \wedge\ \ H[i] \quad\ \wedge\ \neg H[i+1] \\ \wedge\ \neg H'[i] \quad \wedge\ \ H'[i+1] \end{pmatrix} \wedge\ pres(\Pi) \ \wedge\ pres_{\neq i,i+1}(H)$$

where, for every array variable X,

$$pres(X):\quad \forall j \cdot X[j]=X'[j]$$

$$pres_{\neq i}(X):\quad \forall j \cdot j \neq i \rightarrow X[j]=X'[j]$$

$$pres_{\neq i,i+1}(X):\quad \forall j \cdot (j\neq i \wedge j \neq i+1) \rightarrow X[j]=X'[j]$$

The formula ρ_s represents a transition of a *single* process, while ρ_c represents a *joint communication* transition of two contiguous processes.

The FS1S formula representing *all* transitions is thus

$$\rho_{\text{MUX}}(V,V'):\quad \rho_s(V,V') \vee \rho_c(V,V')$$

Let $\mathcal{C}(V)$ be an FS1S formula representing a set of configurations of program MUX. The $pred_{\text{MUX}}$ transformer can be expressed by:

$$pred_{\text{MUX}}(\mathcal{C}):\quad \exists V' \cdot \quad \rho_{\text{MUX}}(V,V') \ \wedge\ \mathcal{C}(V')$$

This formula represents the set of all configurations from which a configuration in $\mathcal{C}(V')$ can be reached in a single ρ_{MUX} step.

Applying SYMB-MC *to* MUX

Following is a demonstration of the application of SYMB-MC on the parameterized program MUX, specified in FS1S. We start the iteration with the negation of the property we want to verify:

$$\varphi_0 \ =\ \neg\, g \ \ =\ \exists i,j \cdot i \neq j \ \wedge\ \Pi[i]=C \ \wedge\ \Pi[j]=C$$

Next, we apply $pred_{\text{MUX}}$ to φ_0, as follows

$$\begin{aligned}\varphi_1 &= \varphi_0 \ \vee \ \exists V' \cdot \rho_{\text{MUX}}(V,V') \ \wedge \ \varphi_0(V') \\ &= \varphi_0 \ \vee \ \exists i,j \cdot i \neq j \ \wedge \ \Pi[i] = T \ \wedge \ H[i] \ \wedge \ \Pi[j] = C\end{aligned}$$

We continue iterating, until the result of the iteration converges, as follows:

$$\varphi_2 = \varphi_1 \ \vee \ \exists i,j \cdot i \neq j \ \wedge \ \left(\begin{array}{c}\Pi[i] = N \wedge H[i] \wedge \Pi[j] = C \\ \vee\ \Pi[i] = T \wedge H[i] \wedge \Pi[j] = T \ \wedge \ H[j]\end{array}\right)$$

$$\begin{aligned}\varphi_3 &= \varphi_2 \ \vee \ \exists i,j \cdot i \neq j \ \wedge \ \Pi[i] = N \ \wedge \ H[i] \ \wedge \ \Pi[j] = T \ \wedge H[j] \\ \varphi_4 &= \varphi_3 \ \vee \ \exists i,j \cdot i \neq j \ \wedge \ \Pi[i] = N \ \wedge \ H[i] \ \wedge \ \Pi[j] = N \ \wedge \ H[j]\end{aligned}$$

The iteration sequence converges at φ_5 ($\varphi_5 = \varphi_4$) with the final value

$$\varphi_5\colon \quad \exists i,j \cdot i \neq j \ \wedge \ (H[i] \ \vee \ \Pi[i] = C) \ \wedge \ (H[j] \ \vee \ \Pi[j] = C)$$

Finally, we check the intersection with the initial condition

$$\varphi_5 \ \wedge \ init_{\text{MUX}} = \text{F}$$

Since the intersection is false, a configuration satisfying $\neg g$ can not be reached from an initial configuration. We can thus conclude that g is an invariant of program MUX.

Claim 1 *If P is a system with an encoding of its global state into* FS1S, *and both the global transition relation of P and the goal assertion g can be represented in* FS1S, *then procedure* SYMB-MC *can be applied to the verification of $P \models \Box\, g$, using* FS1S *as the assertional language.*

Claim 1 does not guarantee that the application of SYMB-MC will terminate.

In the following, we present some additional examples of parameterized systems which can be handled by the approach presented in this section. In particular, we show that the case of a process ring can be treated with FS1S, and that the approach can handle both synchronous and asynchronous communication.

5 Additional Examples

Processor Ring

Example MUX considered processes arranged in an array, where tokens could only move from left to right. Once the rightmost process obtains the token, it cannot deliver it to any other process. This, of course, is a degenerate version of the real protocol, in which the processes are arranged in a ring.

The transition relation for the ring configuration is

$$\rho_{\text{RING}}(V, V') : \rho_s(V, V') \ \vee \ \rho_{\text{C-RING}}(V, V'),$$

where

$$\rho_{\text{C-RING}}(V, V'): \quad \exists i \cdot \begin{pmatrix} \Pi[i] = N \wedge \Pi[i+1] = T \\ \wedge\ H[i] \qquad \wedge \neg H[i+1] \\ \wedge \neg H'[i] \qquad \wedge\ H'[i+1] \end{pmatrix} \bigwedge \mathit{pres}(\Pi) \bigwedge \mathit{pres}_{\neq i, i+1}(H)$$

The execution of procedure SYMB-MC converges, and the specification g is found to be an invariant of program PROC-RING.

Asynchronous Communication

In the previous examples, the communication between processes was synchronous. In the following, we transform the ring configuration with synchronous communication to a similar program based on asynchronous communication. We modify the channel declaration in Fig. 2 as follows

local α: **array** $[0..M-1]$ **of channel [1..1] of boolean**.

According to the conventions of [MP91], this declaration identifies $\alpha[0..M-1]$ as an array of asynchronous channels with a buffering capacity 1. We refer to the so modified program as ASYNC-RING.

The initial condition for ASYNC-RING:

$$\mathit{init}_{\text{ASYNC-RING}}: \quad \mathit{init}_{\text{MUX}} \ \wedge \ \forall i \cdot \neg\alpha[i]$$

The transition relation $\rho_{\text{ASYNC-RING}}$:

$$\begin{pmatrix} \rho_s(V, V') \ \wedge \ \mathit{pres}(\alpha) \\ \\ \vee\ \exists i \cdot \begin{pmatrix} \Pi[i] = N \ \wedge \ H[i] \ \wedge \ \neg H'[i] \ \wedge \ \neg\alpha[i] \ \wedge \ \alpha'[i] \\ \wedge\ \mathit{pres}(\Pi) \ \wedge \ \mathit{pres}_{\neq i}(H) \ \wedge \ \mathit{pres}_{\neq i}(\alpha) \end{pmatrix} \\ \\ \vee\ \exists i \cdot \begin{pmatrix} \Pi[i+1] = T \ \wedge \ \neg H[i+1] \ \wedge \ H'[i+1] \\ \wedge\ \alpha[i] \ \wedge \ \neg\alpha'[i] \ \wedge \ \mathit{pres}(\Pi) \\ \wedge\ \mathit{pres}_{\neq i+1}(H) \ \wedge \ \mathit{pres}_{\neq i}(\alpha) \end{pmatrix} \end{pmatrix}$$

This transition relation consists of three disjuncts. The first disjunct represents all local transitions of a single process. The next two disjuncts represent an asynchronous token passing, which is performed in two separate transitions.

For program ASYNC-RING, the execution of procedure SYMB-MC converges, and the specification g is found to be an invariant.

A Protocol with Request Messages

The protocol presented in Fig. 2 satisfies the safety property of mutual exclusion, but does not satisfy the liveness property of *accessibility*. Namely, it does not guarantee that any process wishing to enter its critical section will eventually do so. To see that accessibility is not guaranteed, consider a 3-process configuration of the following form:

$$\langle N, \text{T}\rangle\, \langle N, \text{F}\rangle\, \langle T, \text{F}\rangle$$

In this configuration $P[0]$ has the token, $P[2]$ is interested in entering its critical section, but $P[1]$ is not. Since the token can only be transferred from a process in state $\langle N, \text{T}\rangle$ to a process in state $\langle T, \text{F}\rangle$, $P[2]$ will not be able to obtain the token until $P[1]$ moves to state $\langle T, \text{F}\rangle$.

This situation can be remedied by allowing processes to receive the token from their left neighbor even when they are *indifferent*, i.e. executing at the non-critical location N. Such a version of the protocol may, however, be considered highly inefficient, since it enforces (due to fairness) continuous movement of the token between indifferent processes, even when no process is interested in entering its critical section.

An efficient solution which ensures accessibility, uses an additional local boolean variable *req*, which is true for all processes having some right neighbor who is interested in entering its critical section. In addition to the t token which moves from left to right, the improved protocol introduces an r token which moves from right to left, representing requests for the t token. In Fig. 3, we present program MUX-REQ which implements the management of request tokens.

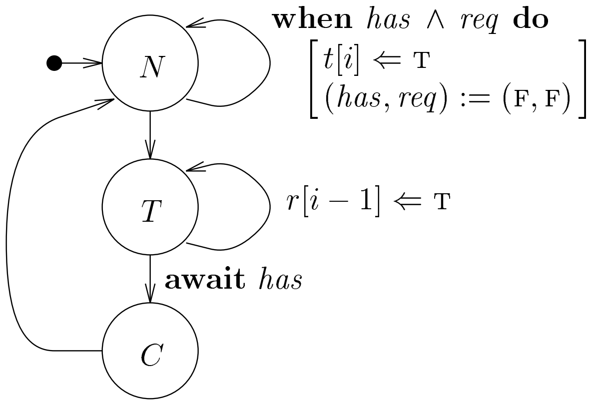
Every process of program MUX-REQ consists of three sub-processes running in parallel and communicating by the shared variables *has* and *req*. The first subprocess performs the main functions of switching between the non-critical and the critical sections. The second sub-process is ever ready to receive a token from channel $t[i-1]$ and store it in variables *has*. The third sub-process is responsible for communicating the request tokens along channels $r[i]$ and $r[i-1]$. If variable *req* is currently false, the sub-process is ready to receive a new request from its right neighbor. If *req* is currently true, the process propagates the request token to its left neighbor.

For the FS1S representation, we use the additional array variable R to represent the value of the variable *req* in all processes. The initial condition for program

in M : **integer where** $M \geq 1$
local t, r : **array** $[0..M{-}1]$ **of channel of boolean**

$\overset{M-1}{\underset{i=0}{||}}$ $P[i]$ ::

local *has* : **boolean initially** $has = (i = 0)$
req : **boolean initially** $req = \text{F}$



||

loop forever do
$t[i-1] \Rightarrow has$

||

loop forever do
if *req* **then** $r[i-1] \Leftarrow \text{T}$
else $r[i] \Rightarrow req$

Fig. 3. Program MUX-REQ with request tokens.

MUX-REQ is given by the FS1S formula

$$init_{\text{MUX-REQ}}: \quad \forall i \cdot \Pi[i] = N \ \wedge \ (H[i] \leftrightarrow i = 0) \ \wedge \ R[i] = \text{F}$$

The transition relation for program MUX-REQ is given by the disjunction:

$$\rho_{\text{MUX-REQ}}: \quad \rho_I \ \vee \ \rho_\ell \ \vee \ \rho_t \ \vee \ \rho_r,$$

where ρ_I is the idling transition, ρ_ℓ describes changes in the control location of sub-process 1, while ρ_t describes transitions related to communications on channel t and ρ_r describes transitions related to communications on channel r.

Transition relation ρ_ℓ is given by:

$$\rho_\ell: \quad \exists i \cdot \left(\begin{array}{c} pres_{\neq i}(\Pi) \ \wedge \ pres(H) \ \wedge \ pres(R) \\ \wedge \left(\begin{array}{l} \ \ \Pi[i] = N \wedge \Pi'[i] = T \\ \vee \ \Pi[i] = T \ \wedge \Pi'[i] = C \ \wedge H[i] \\ \vee \ \Pi[i] = C \ \wedge \Pi'[i] = N \end{array} \right) \end{array} \right)$$

Transition relation ρ_t is given by:

$$\rho_t: \quad \exists i \cdot \begin{pmatrix} pres(\Pi) \ \wedge \ pres_{\neq i,i+1}(H) \ \wedge \ pres_{\neq i}(R) \\ \wedge \ H[i] \ \wedge \ R[i] \ \wedge \ \neg H'[i] \ \wedge \ \neg R'[i] \ \wedge \ H'[i+1] \end{pmatrix}$$

Transition relation ρ_r is given by:

$$\rho_r: \quad \exists i \cdot \begin{pmatrix} pres(\Pi) \ \wedge \ pres(H) \ \wedge \ pres_{\neq i}(R) \\ \wedge \ \neg R[i] \ \wedge \ (\Pi[i+1] = T \ \vee \ R[i+1]) \ \wedge \ R'[i] \end{pmatrix}$$

Applying procedure SYMB-MC to program MUX-REQ and the mutual-exclusion specification

$$g: \quad \neg \exists i, j \cdot i \neq j \ \wedge \ \Pi[i] = C \ \wedge \ \Pi[j] = C,$$

the procedure converges. This proves that the specification g is an invariant of program REQ.

6 Tree Languages

In this section, we extend the method of regular expressions over strings to deal with regular tree languages (see [TW68], [GS84], [Don70]). 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 deal with varying arity. We use the logic FS∗S (*finitary second-order theory of* $\mathbb{N}$ *successors*), a generalization of WS2S, as a specification language for regular sets of trees. For the implementation, expressions in FS∗S are translated into BTA (*bottom-up tree automata*), generalized to deal with varying arity. [4]

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 α by specifying the path from the root to α. Thus, $\alpha = \Lambda \in S$ represents the root, and $(1, 0) \in S$ represents the node which is the first child

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

of the second child of the root. A node $\alpha \in S$ is a leaf, if it is not a prefix of any other member of S.

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

7 The Logic FS*S

Following is a brief definition of the logic.

Syntax

We assume a *signature* $\Xi : \{\Sigma_1, \ldots, \Sigma_k\}$ consisting of a finite set of finite alphabets. The *vocabulary* consists of *position variables* $p_1, p_2, \ldots$ and, for each $\Sigma_i \in \Xi$, a set of Σ_i-*tree variables* $X_i, Y_i, Z_i, \ldots$.

- Position (First-order) terms:
  - The constant Λ.
  - Any position variable p_i.
- Letter terms:
  - Every $a \in \Sigma_i$ is a Σ_i-term.
  - If X is a Σ_i-tree variable and t is a position term, then $X[t]$ is a Σ_i-term.
- Atomic Formulas:
  - $t_1 \sim t_2$, where t_1 and t_2 are position terms and $\sim \in \{=, <, \prec\}$.
  - $x = y$, where x and y are Σ_i-terms for some $\Sigma_i \in \Xi$.
- Formulas:
  - An atomic formula is a formula.
  - Let φ and ψ be formulas. Then $\neg\varphi$, $\varphi \vee \psi$, $\exists p \cdot \varphi$, $\exists X \cdot \varphi$ are formulas, where p is a position variable and X is a tree variable.

Semantics

Let ψ be an FS*S formula. A *model* for ψ is given by $M = \langle S, \nu, \mathcal{A} \rangle$, where S is a tree structure, ν assigns to each position variable p a sequence of natural numbers $\nu(p) \in S$, and $\mathcal{A}$ assigns to each Σ_i-tree variable X a Σ_i-tree with tree structure S.

Given a model $M = \langle S, \nu, \mathcal{A} \rangle$, we inductively define the *interpretation* I_M induced by M as follows:

- I_M interprets every *position term* t into a sequence of natural numbers $I_M(t) \in S$, as follows:
  - The constant symbol Λ is interpreted as the empty sequence.
  - For a position variable p, $I_M(p) = \nu(p)$.
- A Σ_i-term is interpreted into a Σ_i-letter, as follows:
  - The constant symbol $a \in \Sigma_i$ is interpreted into the Σ_i-letter a.
  - If $\mathcal{A}(X) = \langle S, \lambda \rangle$, $I_M(t) = \alpha \in S$ and $\lambda(\alpha) = a$ then $I_M(X[t]) = a$.
- Formulas are interpreted into boolean values ($\{0, 1\}$), as follows:
  - For position terms t_1 and t_2,
    $I_M(t_1 = t_2)$ evaluates to 1 if $I_M(t_1) = I_M(t_2)$.
    $I_M(t_1 < t_2)$ evaluates to 1 if $I_M(t_1)$ is a prefix of $I_M(t_2)$.
    $I_M(t_1 \prec t_2)$ evaluates to 1 if $I_M(t_1)$ is smaller than $I_M(t_2)$ by lexicographic ordering.
  - For Σ_i-terms x and y, $I_M(x = y)$ evaluates to 1 if $I_M(x)$ equals $I_M(y)$.
  - $\neg\varphi, \varphi \vee \psi, \varphi \wedge \psi, \varphi \leftrightarrow \psi, \varphi \rightarrow \psi$ — where φ and ψ are formulas, are interpreted in the natural way, after the formulas φ and ψ are interpreted.
  - $\exists p \cdot \varphi$ — is true iff there exists a model $M' = \langle S, \nu', \mathcal{A} \rangle$, such that ν and ν' differ at most in the interpretation of the position variable p, and such that $I_{M'}(\varphi) = 1$.
  - $\exists X \cdot \varphi$ — is true iff there exists a model $M' = \langle S, \nu, \mathcal{A}' \rangle$, such that $\mathcal{A}$ and $\mathcal{A}'$ differ at most in the interpretation of the tree variable X, and such that $I_{M'}(\varphi) = 1$.

The following FS*S formulas are used as shortcut notations:

$$
\begin{array}{ll}
\textit{son}(x,y) & : x < y \wedge \neg\exists z \cdot x < z \wedge z < y \\
\textit{brothers}(x,y) & : \exists z \cdot \textit{son}(z,x) \wedge \textit{son}(z,y) \\
\textit{elder-brother}(x,y) & : \textit{brothers}(x,y) \wedge x \prec y \wedge \\
& \quad \neg\exists z \cdot \textit{brothers}(x,z) \wedge x \prec z \wedge z \prec y \\
\textit{leaf}(x) & : \neg\exists y \cdot x < y
\end{array}
$$

where $\textit{son}(x, y)$ iff y is the son of x in S, $\textit{brothers}(x, y)$ iff both x and y are sons of the same node in S, $\textit{elder-brother}(x, y)$ iff x and y are brothers and x is the rightmost brother to the left of y. Finally $\textit{leaf}(x)$ iff x represents a leaf node in S.

7.1 Bottom-Up Tree Automata

A (*variable-arity*) *bottom-up tree automaton* (BTA) is a tuple 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 $q \in \Delta(q_1 \cdots q_k, \lambda(n))$. Thus, the annotation at n depends on the annotation of the children of n and on the Σ-letter labeling node n. Note that in the case of unary trees (strings) this definition specializes to the familiar automaton where $\Delta(w,a)$ is always used with either $w = \Lambda$ (in the case of the initial state) or with $w = q$ for some $q \in Q$.

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 said to be *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:

$$\begin{array}{lll}
\Sigma & : & \{a,b\} \\
Q & : & \{q_0, q_1, q_2\} \\
\Delta & : & \text{Defined as follows:} \\
 & & \Delta(q_0^*, a) = \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{q_0\} \\
 & & \Delta(q_0^*, b) = \qquad\qquad \Delta(q_0^* q_1 q_0^*, a) = \qquad\qquad\qquad \{q_1\} \\
 & & \Delta(Q^* q_1 Q^* q_1 Q^*, \{a,b\}) = \Delta(Q^* q_2 Q^*, \{a,b\}) = \Delta(q_0^* q_1 q_0^*, b) = \{q_2\} \\
F & : & \{q_1\}
\end{array}$$

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 heading 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, and the tree should be rejected. ▟

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 2

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

7.2 Translation from FS∗S *to* BTA

The translation from FS∗S to BTA proceeds in two steps. In the first step, a FS∗S formula is reduced to a simpler form, in which only a limited set of atomic formulas are allowed, as follows:

- Eliminate Λ position terms which are not of the form $x = \Lambda$, by rewriting as follows: in each such atomic formula, replace all position terms Λ with a new position variable z and add the condition $z = \Lambda$.
  For example, rewrite
  $$x < \Lambda \qquad \text{as} \qquad \exists z \cdot x < z \wedge z = \Lambda$$
- Atomic formulas of the form $X[x] = Y[y]$, where X, Y have the same alphabet Σ, are rewritten as:
  $$\bigvee_{\sigma \in \Sigma} (X[x] = \sigma \wedge Y[y] = \sigma)$$

The result is a formula with atomic formulas restricted to the following types: $x = y$, $x < y$, $x \prec y$, $x = \Lambda$, $X[x] = \sigma$.

The atomic formulas in the simplified expression are then translated to BTA. In the following examples, translations of these atomic formulas are presented.

We use the following conventions. $q_r \in Q$ is the rejecting state, i.e. if any tree node is annotated with q_r then the tree will be rejected. The state $q_a \in Q$ is the accepting state. For all following BTA's, $F = \{q_a\}$.

- $x = y$

$$
\begin{aligned}
\Sigma &: \{xy, x\overline{y}, \overline{x}y, \bar{x}\bar{y}\} \\
Q &: \{q_0, q_a, q_r\} \\
\Delta &: \text{Defined as follows:} \\
&\quad \Delta(q_0^*, \bar{x}\bar{y}) = \{q_0\} \\
&\quad \Delta(q_0^*, xy) = \Delta(q_0^* q_a q_0^*, \bar{x}\bar{y}) = \{q_a\} \\
&\quad \Delta(Q^* q_a Q^* q_a Q^*, \Sigma) = \Delta(Q^* q_a Q^*, xy) = \{q_r\} \\
&\quad \Delta(Q^* q_r Q^*, \Sigma) = \Delta(Q^*, \overline{x}y) = \Delta(Q^*, x\overline{y}) = \{q_r\}
\end{aligned}
$$

- $x < y$

$$
\begin{aligned}
\Sigma &: \{xy, x\overline{y}, \overline{x}y, \bar{x}\bar{y}\} \\
Q &: \{q_0, q_1, q_a, q_r\} \\
\Delta &: \text{Defined as follows:} \\
&\quad \Delta(q_0^*, \bar{x}\bar{y}) = \{q_0\} \\
&\quad \Delta(q_0^*, \overline{x}y) = \{q_1\} \\
&\quad \Delta(q_0^* q_1 q_0^*, \bar{x}\bar{y}) = \{q_1\} \\
&\quad \Delta(q_0^* q_1 q_0^*, x\overline{y}) = \{q_a\} \\
&\quad \Delta(q_0^* q_a q_0^*, \bar{x}\bar{y}) = \{q_a\} \\
&\quad \text{Otherwise the value of } \Delta \text{ is } q_r
\end{aligned}
$$

- $x \prec y$

$$
\begin{aligned}
\Sigma &: \{xy, x\overline{y}, \overline{x}y, \bar{x}\bar{y}\} \\
Q &: \{q_0, q_1, q_2, q_a, q_r\} \\
\Delta &: \text{Defined as follows:} \\
&\quad \Delta(q_0^*, \bar{x}\bar{y}) = \{q_0\} \\
&\quad \Delta(q_0^*, x\overline{y}) = \{q_1\} \\
&\quad \Delta(q_0^*, \overline{x}y) = \{q_2\} \\
&\quad \Delta(q_0^* q_1 q_0^* q_2 q_0^*, \bar{x}\bar{y}) = \Delta(q_0^* q_1 q_0^*, x\overline{y}) = \{q_a\} \\
&\quad \Delta(q_0^* q_a q_0^*, \bar{x}\bar{y}) = \{q_a\} \\
&\quad \text{Otherwise the value of } \Delta \text{ is } q_r
\end{aligned}
$$

– $X[x] = \sigma$
where X is a Σ_i-tree variable,

$$\begin{array}{ll} \Sigma & : \Sigma_i x \cup \Sigma_i \overline{x} \\ Q & : \{q_0, q_a, q_r\} \\ \Delta & : \text{Defined as follows:} \\ & \quad \Delta(q_0^*, \Sigma_i \overline{x}) = \qquad\qquad \{q_0\} \\ & \quad \Delta(q_0^*, \sigma x) = \qquad\qquad \{q_a\} \\ & \quad \Delta(q_0^* q_a q_0^*, \Sigma_i \overline{x}) = \qquad \{q_a\} \\ & \quad \text{Otherwise the value of } \Delta \text{ is } q_r \end{array}$$

Finally, formulas translated by performing operations on the BTAs resulting from the atomic formulas ([GS84],[Don70],[TW68]).

7.3 Configurations of a Process Tree as a Tree Language

As a running example for a system organized as a process tree, consider program PERCOLATE of Fig. 4.

$$\begin{array}{c} \textbf{in } S : \textbf{tree structure} \\[1em] \mathop{\Big\|}\limits_{\alpha \in S} P[\alpha] :: \left[\begin{array}{l} \textbf{local } val : \{0,1,u\} \textbf{ where } leaf(\alpha) \leftrightarrow val \in \{0,1\} \\[1em] \left[\begin{array}{l} M := \{m \mid \alpha \cdot m \in S\} \\ \textbf{repeat} \\ \quad \textbf{if } \forall m \in M : P[\alpha \cdot m].val \neq u \\ \qquad \textbf{then } val := \bigvee_{m \in M} P[\alpha \cdot m].val \\ \textbf{until } val \neq u \end{array} \right] \end{array} \right] \end{array}$$

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 is interpreted as "undefined yet", which implies that it will eventually change to either 0 or 1. Initially, all the leaf processes have $val \in \{0, 1\}$ and all other processes have $val = u$. The purpose of program PERCOLATE is to percolate to the root 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 $P[\alpha].val$ is not yet defined but all its children's values are defined then $P[\alpha]$ sets its value to the disjunction of the values of its children. The configuration of program PERCOLATE in FS*S is represented by a $\Sigma_{\text{PERCOLATE}}$-tree variable P over the alphabet $\Sigma_{\text{PERCOLATE}}$: $\{0, 1, u\}$.

The initial condition of program PERCOLATE is:

$$init_{percol}: \quad \forall x \cdot \; leaf(x) \leftrightarrow (P[x] = 1 \vee P[x] = 0)$$

The transition relation of program PERCOLATE is:

$$\exists x \cdot \left(\begin{array}{l} \quad \forall y \cdot x \neq y \rightarrow P'[y] = P[y] \\ \wedge \; (\exists z \cdot (son(x,z) \wedge P[z] = u)) \rightarrow P'[x] = P[x] \\ \wedge \left(\begin{array}{l} (\forall z \cdot (son(x,z) \rightarrow P[z] \neq u)) \rightarrow \\ \quad ((P'[x] = 1 \leftrightarrow \exists z \cdot (son(x,z) \wedge P[z] = 1)) \wedge P'[x] \neq u) \end{array} \right) \end{array} \right)$$

The property to be verified can be specified by the following assertion:

$$g\text{:} \quad \forall y \cdot P[y] \neq u \;\; \rightarrow \;\; \Big(P[y] = 1 \leftrightarrow \exists x \cdot y \leq x \wedge leaf(x) \wedge P[x] = 1\Big).$$

This FS*S formula states that for any node α in the process tree, if $\alpha.val \neq u$ then $\alpha.val$ equals the disjunction of all val values at the leaves of its subtree.

The execution of the symbolic model checking algorithm does not converge. To reach convergence we construct a *meta-transition* for PERCOLATE, as described in [WB94]. While in the original transition relation processes update their val variable one at a time, the meta-transition allows all processes to update their val variable simultaneously. As a result, the set of reachable states explored during a single iteration step of SYMB-MC is increased.

Using the meta-transition the execution of SYMB-MC converges and the assertion g is found to be an invariant of program PERCOLATE.

8 The Status of the Implementation

We have constructed two implementations of systems which use the SMV input language extended with a limited subset of either FS1S or FS*S. The internal representations of the two systems are that of finite automata (FSA) and BTAs respectively.

The systems accept as inputs the representations of $init_P$, g and the transition relation $\mathcal{T}_P$, and either confirms that g is a P-invariant, or produces a counterexample, which is a P-computation reaching a $\neg g$-configuration, or fails to terminate.

With the implementation for string languages we have verified the examples presented in this paper. In addition, we verified two of the four safety specifications of the Futurebus+ cache coherency prototcol which were verified in [CGH+93] and [PS96]. These were checked for the single-bus version of the Futurebus+ protocol and were found to be correct. The running time for the simple examples was from seconds, to no more than a minute. For the Futurebus+ the more lengthy verification took less than two hours on a Silicon Graphics Challenge computer.

The representation of automata in the FS1S 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. Our transition function has the type δ: $Q \times \mathit{local\_assertions} \mapsto 2^Q$, where a local assertion is an assertion over the local state variables. In this implementation, we followed many of the ideas suggested in [BK95] and [HJJ+96].

The FS*S implementation can handle simple examples as PERCOLATE.

9 Conclusions and Future Research

The paper extended the method of symbolic model checking to deal with systems with infinitely many states. The notion of *adequate assertional language* is general enough to accommodate many additional decidable theories that can match particular types of parameterized systems. So far, the generalization of the symbolic model checking method was illustrated only for the safety property of state invariance. An interesting and currently investigated question is how to extend the method to apply to other types of temporal properties, in particular, liveness properties.

Another promising line of research is how to handle the case that the iteration does not converge. Other studies of infinite-state systems have used some notion of *widening* which tries to extrapolate an infinite sequence to its limit. Some version of such an extrapolation may prove useful to handle our cases of divergent assertion sequences.

References

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

[BCG86] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many finite state processes. In *Proc. 5th ACM Symp. Princ. of Dist. Comp.*, pages 240–248, 1986.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. 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, *Proc. 7^{th} Intl. Conference on Computer Aided Verification (CAV'95)*, volume 939 of *Lect. Notes in Comp. Sci.*, pages 31–41. Springer-Verlag, 1995.

[BM96] A. Bouajjani and O. Maler. Reachability analysis of push-down automata. In *Workshop on Infinite-state Systems*, Pisa, 1996.

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

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

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, 8:244–263, 1986.

[CGH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the futurebus+ cache coherence protocol. In L. Claesen, editor, *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications.* North-Holland, April 1993.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *6th International Conference on Concurrency Theory (CONCUR'95)*, pages 395–407, Philadelphia, PA, August 1995.

[Don70] J. Doner. Tree acceptors and some of their applications. *J. Comp. Systems Sci.*, 4:406–451, 1970.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comp. Prog.*, 2:241–266, 1982.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22th ACM Conf. on Principles of Programming Languages, POPL'95*, San Francisco, 1995.

[EN96] E.A. Emerson and K.S. Namjoshi. Automatic verification of parameterized synchronous systems. In R. Alur and T. Henzinger, editors, *Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1996.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree automata.* Akadémiai Kiadó (Publishing House of the Hungarian Academy of Sciences), Budapest, 1984.

[HJJ+96] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1996.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7):523–543, 1992.

[ID96] C.N. Ip and D. Dill. Verifying systems with replicated components in Murφ. In R. Alur and T. Henzinger, editors, *Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96)*, Lect. Notes in Comp. Sci. Springer-Verlag, 1996.

[KG96] O. Kupferman and O. Grumberg. Branching time temporal logic and amorphous tree automata. *Information and Computation*, 125(1):62–69, 1996.

[KM89] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In P. Rudnicki, editor, *Proceedings of the 8th Annual Symposium on Principles of Distributed Computing*, pages 239–248, Edmonton, AB, Canada, August 1989. ACM Press.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*, Paris, 1997.

[McM93] K.L McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, 1993.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In R. Alur and T. Henzinger, editors, *Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96)*, Lect. Notes in Comp. Sci., pages 184–195. Springer-Verlag, 1996.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in *cesar*. In M. Dezani-Ciancaglini and M. Montanari, editors, *International Symposium on Programming*, volume 137 of *Lect. Notes in Comp. Sci.*, pages 337–351. Springer-Verlag, 1982.

[SG89] Z. Shtadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lect. Notes in Comp. Sci.*, pages 151–165. Springer-Verlag, 1989.

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

[Tho90] W. Thomas. Automata on infinite objects. *Handbook of theoretical computer science*, pages 165–191, 1990.

[TW68] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Math. Systems Theory*, 2:57–81, 1968.

[WB94] P. Wolper and B. Boigelot. Symbolic verification with periodic sets. In D. Dill, editor, *Proc. 6^{th} Intl. Conference on Computer Aided Verification (CAV'94)*, volume 818 of *Lect. Notes in Comp. Sci.*, pages 55–67. Springer-Verlag, 1994.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lect. Notes in Comp. Sci.*, pages 68–80. Springer-Verlag, 1989.