Electronic Notes in Theoretical Computer Science 89 No. 4 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 17 pages

BMC via On-the-Fly Determinization

Toni Jussila [a,1,2] Keijo Heljanko [b,2,3] Ilkka Niemelä [a,2]

[a] *Helsinki University of Technology*
Laboratory for Theoretical Computer Science
P.O. Box 5400, FIN-02015 HUT, Finland

[b] *University of Stuttgart*
Institute for Formal Methods in Computer Science
Universitätsstr. 38, D-70569 Stuttgart, Germany

Abstract

The paper develops novel bounded model checking (BMC) techniques for labeled transition systems. The aim is to increase the efficiency of BMC by exploiting the inherent concurrency in the product of LTSs in order to cover more executions of the product within a given bound. This is done by considering a non-standard execution model, step executions, where multiple actions can take place simultaneously and where component LTSs are determinized on-the-fly, i.e., a component may be in a set of states in a step instead of in just one as in standard interleaving executions. Step executions can be further restricted to a subclass called process executions without losing reachable states. For bounded model checking of reachability properties of the product of LTSs the paper presents translation schemes from LTSs to a constrained Boolean circuit such that satisfying valuations of the circuit correspond to step (process) executions of the product. The translation schemes have been implemented and some experimental comparisons performed. The results show that the bound needed for step and process executions is in most cases lower than in interleaving executions and that the running time of the model checker using process executions is smaller than using steps. Moreover, the performance compares favorably to a state-of-the-art interleaving BMC implementation in the NuSMV system.

[1] The financial support from Nokia Foundation and Helsinki Graduate School of Computer Science and Engineering is gratefully acknowledged.
[2] The financial support from the Academy of Finland (Project 53695) is gratefully acknowledged.
[3] The financial support from the Academy of Finland (grant for research work abroad) and FET project ADVANCE contract No IST-1999-29082 are gratefully acknowledged.

1 Introduction

Bounded model checking (BMC) is a verification technique that considers only executions of bounded length of the chosen formalism [1]. The general model checking problem for linear temporal logic (LTL) is known to be **PSPACE**-complete, but the bounded case is in **NP** (assuming the used bound to be given in unary encoding). The very idea is to compile the system under verification, the property to be verified and a bound k on the length of the execution to a propositional formula having a model iff the system has an execution of length k that violates the property. The methodology has been successfully applied in industrial setting [2,3].

The aim of the paper is to develop efficient BMC techniques for systems modeled as products of labeled transition systems (LTSs) by exploiting the inherent concurrency in the systems. The basic idea is to cover more executions of a system within a given bound in a way that the size of the encoding is not substantially increased, i.e., it remains linear w.r.t. the bound. The standard approach to BMC is to use interleaving executions where exactly one action is occurring at a time. Here the idea is to encode interleaving executions more compactly by allowing multiple occurrences of actions in different components of the system simultaneously. This kind of an approach has already been investigated using 1-safe Petri nets as the system model and employing step and process executions of Petri nets with encouraging results [9,8].

The novelty in this paper is a technique that exploits independence of actions in a synchronizing system of LTSs so that multiple independent actions can take place in different component LTSs simultaneously. This technique is further combined with an on-the-fly determinization construction where for each component a set of states in which that component can be is maintained. By using determinization the number of different executions the product can have is potentially dramatically reduced, and furthermore invisible transitions do not contribute to the length of an execution. In this work the concurrent executions of independent actions combined with on-the-fly determinization of components are called *step executions*. Without compromising reachable states, step executions can be further restricted to *process executions* satisfying an extra condition on visible actions taking place simultaneously.

Based on these ideas a technique for bounded model checking of reachability properties of the synchronizing product of LTSs is developed by devising a translation scheme from the LTSs to a constrained Boolean circuit [11] such that satisfying valuations of the circuit correspond to step executions of the product. A minor extension of the mapping handles process executions. In both cases the size of the encoding is linear w.r.t. the bound. For the encoding, Boolean circuits are employed for clarity and compactness. Such circuits can be translated to propositional formulae in CNF with a linear blow-up by introducing additional propositional variables using standard techniques [11].

The approach has been applied to a set of examples and the data obtained justify the following points. Firstly, the bound needed for step and process executions is in most cases lower than in the traditional interleaving model. Secondly, the running times using process executions are often smaller than using steps. Finally, the results compare favorably to the running times of a state-of-the-art interleaving BMC implementation [5].

The paper is organized as follows. Section 2 introduces the formalism used as the modeling language and Sect. 3 Boolean circuits. Section 4 presents the encoding schemes for both execution models. Section 5 gives test results comparing step and process executions to NuSMV [4,5] and finally Sect. 6 concludes.

2 System Modeling Formalism

Concurrent systems specified as labeled transition systems (LTS) are studied in this paper. Three execution models for the synchronizing system of LTSs are introduced. The first is the standard interleaving semantics. Thereafter, the step and process models allowing independent actions to take place simultaneously are defined. The section ends with an analysis on the relation between the different models.

Definition 2.1 An LTS is a 4-tuple $L = (S, I, \Gamma, \Delta)$ where

- S is a non-empty set of states,
- $I \subseteq S$ is a non-empty set of *initial* states,
- Γ is a non-empty set of visible actions, and
- $\Delta \subseteq S \times (\Gamma \cup \{\tau\}) \times S$, is the *transition relation*, the elements of which are called transitions of L, where τ is the invisible action.

The transitions, whose middle component is τ are called *internal* or invisible to the environment. LTSs can interact by forming a synchronizing system L, denoted here by $L = (L_1, \ldots, L_n)$. Its semantics is defined in terms of interleaving executions.

Definition 2.2 Let $L = (L_1, \ldots, L_n)$ be a system of synchronizing LTSs. An interleaving execution of S is the sequence

$$S_1 \xrightarrow{a_1} S_2 \xrightarrow{a_2} \cdots \xrightarrow{a_k} S_{k+1} \tag{1}$$

where each $S_i = (s_i^1, \ldots, s_i^n), s_i^j \in S_j$, i.e. s_i^j is a state of LTS L_j and each $a_i \in \Gamma_1 \cup \cdots \cup \Gamma_n \cup \{\tau\}$. Furthermore

(i) For all L_i, $s_0^i \in I_i$.

(ii) If $a_i \neq \tau$, then $(s_i^1, \ldots, s_i^n) \xrightarrow{a_i} (s_{i+1}^1, \ldots, s_{i+1}^n)$ such that for all L_j, if $a_i \in \Gamma_j$ $(s_i^j, a_i, s_{i+1}^j) \in \Delta_j$ otherwise $s_{i+1}^j = s_i^j$.

(iii) If $a_i = \tau$, then $(s_i^0, \ldots, s_i^n) \xrightarrow{a_i} (s_{i+1}^1, \ldots, s_{i+1}^n)$ such that for one L_j $(s_i^j, \tau, s_{i+1}^j) \in \Delta_j$ and $s_{i+1}^k = s_i^k$ for $k \neq j$.

The definition above is usually given by first defining the synchronized product of the components constituting the system and then presenting the executions using that construction. Definition 1 above is equivalent and better suited for comparing the traditional model to the new contributions presented in the paper (Defs. 2.6 and 2.11). These definitions make use of the following concepts.

Definition 2.3 The concatenation of the visible actions in the interleaving execution σ_{I} in the order mandated by σ_{I} is denoted $pr(\sigma_{\mathrm{I}})$.

Definition 2.4 A state S' is *reachable* iff S' is one of the initial states $I = (I_1, \ldots, I_n)$ or there is an execution σ from an initial state S to S'. A state S' is a *deadlock* state iff it is reachable and the system cannot make any transitions from S'.

Definition 2.5 Let $L = (S, I, \Gamma, \Delta)$ be an LTS and $S' \subseteq S$. The τ-closure of S' is the set of states $S'' \subseteq S$ such that $s \in S''$ iff $s \in S'$ or there is an execution from some state in S' to s containing only τ-transitions.

The following definition presents the step executions of the synchronizing system of LTSs. The model is such that while operating on possibly non-deterministic LTSs it determinizes them on-the-fly. Therefore, in each position in the execution each component may be in a set of states instead of just one.

Definition 2.6 Let $L = (L_1, \ldots, L_n)$ be a synchronizing LTS system. A finite *step* execution σ_{S} of L is a sequence

$$V_1 \xrightarrow{A_1} V_2 \xrightarrow{A_2} \ldots \xrightarrow{A_k} V_{(k+1)} \tag{2}$$

such that each V_i is an n-tuple $(S_i^1, \ldots, S_i^n)$, $S_i^j \subseteq S_j$, $1 \leq j \leq n$, i.e., each S_i^j is a set of states of LTS L_j and each $\emptyset \subset A_i \subseteq \Gamma_1 \cup \cdots \cup \Gamma_n$. In addition all of the following conditions hold:

- In V_1 every S_1^j is the τ-closure of I_j.
- For each A_i and L_j, $|A_i \cap \ \Gamma_j| \leq 1$, i.e., in each step at most one visible action is executed from each LTS.
- For each A_i, if $a \in A_i$, then for each L_j such that $a \in \Gamma_j$ there is a transition $(s_j, a, s'_j) \in \Delta_j$ such that $s_j \in S_i^j$. Furthermore $S_{(i+1)}^j$ is the τ-closure of the set of states formed by all states s'' such that $(s', a, s'') \in \Delta_j$ and $s' \in S_i^j$.
- For each A_i and L_j, if $A_i \cap \Gamma_j = \emptyset$ then $S_{(i+1)}^j = S_i^j$.

The length of σ_{S}, denoted by $|\sigma_{\mathrm{S}}|$, is k. Let $lin(\sigma_{\mathrm{S}})$ denote the set of all possible linearizations of σ_{S}, i.e., the set of strings $\alpha_1\alpha_2\ldots\alpha_k$ such that $\alpha_i \in lin(A_i)$, for each $i = 1, \ldots, k$ where $lin(A_i)$ is the set of strings obtained by concatenating the elements in A_i in any order.

Definition 2.7 Let $S = (s_1, \ldots, s_n)$ and $V = (S'_1, \ldots, S'_n)$, Define $S \sqsubset V$ to mean that each $s_j \in S'_j, 1 \leq j \leq n$.

The following theorems characterize the relation between interleaving and step executions. They assume the synchronizing system $L = (L_1, \ldots, L_n)$.

Theorem 2.8 *Let σ_{I} be an (interleaving) execution of the form (1) of L and $|\sigma_{\mathrm{I}}| = k$. Then there is a step execution σ_{S}*

$$V_1 \xrightarrow{\{a_1\}} V_2 \xrightarrow{\{a_2\}} \cdots \xrightarrow{\{a_l\}} V_{(l+1)} \tag{3}$$

of L such that $a_1 a_2 \ldots a_l = pr(\sigma_{\mathrm{I}})$, $l \leq k$ and $S_{(k+1)} \sqsubset V_{(l+1)}$.

Theorem 2.9 *Let σ_{S} be a step execution of L reaching $V_{(k+1)}$. Then for every state $S \sqsubset V_{(k+1)}$ there is an interleaving execution σ_{I} of L reaching S such that $pr(\sigma_{\mathrm{I}}) \in lin(\sigma_{\mathrm{S}})$.*

Corollary 2.10 *A state S of $L = (L_1, \ldots, L_n)$ is reachable iff there is a step execution $V_1 \xrightarrow{A_1} V_2 \xrightarrow{A_2} \cdots \xrightarrow{A_k} V_{(k+1)}$ such that $S \sqsubset V_{(k+1)}$ for some k.*

The set of step executions of a system contains in most cases different elements intuitively corresponding to the same concurrent behavior. The following addition to Definition 2.6 limits the set without compromising reachable states.

Definition 2.11 A *process* execution of L is a step execution of L fulfilling the following condition

- Whenever $a_i \in A_i$ and $i > 1$, then there is an LTS $L_j \in L$ such that $a_i \in \Gamma_j$ and there is an action $a_k \in A_{i-1} \cap \Gamma_j$.

A step execution that is not a process execution would be characterized by the fact that in some global state every LTS participating in an action a would be in a state where it could take place. It would not, though, be chosen for immediate execution, but the relevant components would remain idle in the same states for some steps and only then execute a.

Theorem 2.12 *Let σ_{S} be step execution of reaching state V. Then there is a process execution σ_{P} reaching V such that $|\sigma_{\mathrm{P}}| \leq |\sigma_{\mathrm{S}}|$.*

Corollary 2.13 *A state s of $L = (L_1, \ldots, L_n)$ is reachable iff there is a process execution $V_1 \xrightarrow{A_1} V_2 \xrightarrow{A_2} \cdots \xrightarrow{A_k} V_{(k+1)}$ such that $S \sqsubset V_{(k+1)}$ for some k.*

Intuitively the process executions are step executions which are in a certain canonical normal form. In fact, this canonical normal form corresponds exactly to the so called Foata normal form [7] from the theory of Mazurkiewicz traces, and also to a partial order semantics for 1-safe Petri nets called processes. For more on this connection, see [8] and further references there. Figure 1 gives two LTSs, both having the visible actions $\Gamma_1 = \Gamma_2 = \{a, b\}$. They will be used as a running example when the elements of the encoding are presented. The encoding assumes, without loss of generality that each visible transition is given a unique label. In the figure, that label l_i is given together with the action associated with the transition.

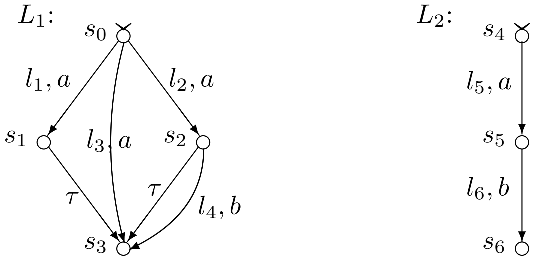

Fig. 1. Running Example

3 Boolean Circuits

The synchronizing systems of LTSs are translated to Boolean circuits. This section, based on the presentation of [11], introduces the concept and the associated terminology. A *Boolean circuit* is a directed acyclic graph where the nodes are called *gates*. The gates can be divided to three categories:

- *input gates* that have no incoming edges nor an associated Boolean function,
- intermediate gates that have both incoming and outgoing edges and an associated Boolean function and
- *output gates* with incoming edges and an associated Boolean function but no outgoing edges.

A *truth valuation* for a circuit with gates $\mathcal{V}$ is a function $\tau : \mathcal{V} \rightarrow \{\text{true}, \text{false}\}$. A valuation is *consistent* with the circuit if $\tau(v) = f(\tau(v_1), \ldots, \tau(v_k))$ for each non-input gate v where f is the Boolean function associated to v and $v_1, \ldots, v_n$ are the gates with edges to v. The *constrained satisfiability problem* for Boolean circuits is formulated as follows: given that gates $c^+ \subseteq \mathcal{V}$ must be true and $c^- \subseteq \mathcal{V}$ must be false, is there a consistent valuation that respects these constraints, i.e., is there a satisfying valuation? The constrained Boolean circuit satisfiability problem is obviously **NP**-complete under the plausible assumption that each Boolean function in the system can be evaluated in polynomial time.

The encoding in the present work applies Boolean circuits where the following standard Boolean functions appear as gates: $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), and $\rightarrow$ (implication). In addition a function of the from $\mathrm{cr}_L^U(v_1, \ldots, v_k)$ is used. It evaluates to true in a valuation τ iff for the cardinality c of the set $\{\tau(v) = \text{true} \mid v \in \{v_1, \ldots, v_k\}\}$ holds that $L \leq c \leq U$ where L and U are fixed constants $0 \leq L \leq U$. The function cr_L^U represents actually a family of functions of which the following two forms are used in the paper: cr_0^1 (at most one true) and cr_1^1 (exactly one true).

4 Encoding

This section presents the structure of the Boolean circuits encoding the step and process executions of the synchronizing system of n LTSs. For representational purposes the gates that appear are given certain illustrative names briefly explained in Table 1. An in-depth description of them follows in subsequent sections with references to figures of gates drawn from the running example.

Table 1
Translation Predicates

Gate	Description
$ex(a,t)$	Action a is executed at time t, input gate.
$in(s,t)$	Execution is in state s at time t.
$sc(L,t)$	Component L scheduled at time t.
$ex(l,t)$	Transition l is executed at time t.
$uv(L,t)$	Unique visible transition from L at time t.
$ni(t)$	Disable idling at time t.
$enok(a,t)$	Execution of action a implies that it is enabled at time t.
$en(a,t)$	Action a is enabled at time t.

The encoding assumes that the LTSs do not have loops containing only τ-transitions involving more than one state. If that is the case, the corresponding component can be preprocessed so that the resulting LTS simulates all the executions of the original. The preprocessing step computes the maximal strongly connected components C_i of the LTS restricted to τ-transitions and replaces each C_i with a single state having as incoming and outgoing transitions the union of those in the set of states in C_i.

The representation follows certain conventions. The variable k is used to denote the length of the execution and the variables s, t, a and l are used to describe arbitrary states, positions in the execution, actions and transition labels, respectively. Based on the division of gates given in Sect. 3, the circuit is composed as follows. Firstly, some gates, namely those labeled with $ex(a,t)$ act as inputs. This special role is marked with two concentric circles. Secondly, the labels $ex(tr,t)$ and $sc(L,t)$ are attached to intermediate gates. Thirdly, the gates $uv(L,t)$ and $ni(t)$ are outputs constrained to true. This is reflected in the figures in which they appear by the symbol $\mathbf{T}$ appearing on the right side of the gate.

The gates labeled $in(s,t)$ can appear in different roles based on the value of t. Gates describing the initial states, i.e. $in(s,1)$ are inputs constrained to true and false depending on whether a state s is an initial state or not. For positions $1 < t \leq k$ the gates are intermediate and for the final position, i.e., $in(s,\ k\ +\ 1)$ they are output gates. When the translation scheme is augmented with a circuit detecting reachability properties, these gates are its inputs. The following subsections present the reasoning for all the gates and the section is concluded by a complete translation algorithm.

4.1 Control Flow

For encoding the control flow of the LTSs the idea is that the $in(s,t)$ gates serve to provide information regarding the progress of execution. For any initial state s of an LTS, $in(s,1)$ is an input gate and is asserted true. This is in accordance with the fact that in the outset the execution in each component is in the initial states. In general, the execution may be in some state at time $t+1$ iff one of the following cases is true.

- The state was reached already at t and not left in step t.
- The state is reached due to it belonging to the τ-closure of some state reached via actions in step t.
- The state is reached by taking some of its incoming visible transitions in step t.

Therefore the gate encoding the progress of control flow is a disjunction of the above three cases. Figure 2 gives an instance encoding the case for the state s_3 of the running example. The disjuncts encode the cases from the left to the right. For instance, taking the leftmost disjunct, if $in(s_3, t-1)$ is true (state reached in the previous step) and $sc(L_1, t)$ is false (component L_1 remained idle in step t), then $in(s_3, t)$ has to be true, i.e. the component remains in state s_3. It should be noted that τ-transitions from a state to itself can (and should) be ignored in the definition.

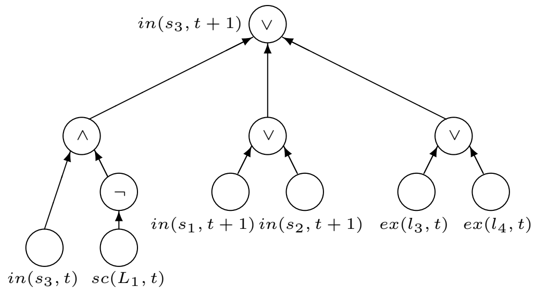

Fig. 2. Progress of Control Flow

The definition makes use of the $sc(L,t)$ and $ex(l,t)$ gates. The former captures the fact that a component L is scheduled iff a visible action in its alphabet is executed.

The reasoning behind the latter, the $ex(l,t)$ gate, is as follows. A transition is traversed in position t iff the action it is labeled with is executed in position

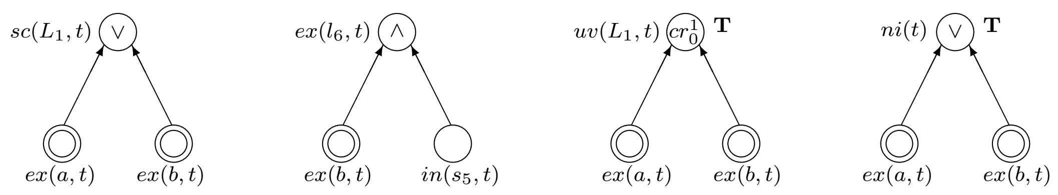

Fig. 3. Elements Illustrating Encoding from Running Example

t and the control flow is in its source state. It should be noted that the definition is not circular, but the control flow in position t together with the executed transitions define the control flow in position $t+1$. The second picture from the left in Fig. 3 illustrates the gate for the transition l_6 in the running example.

So far, the subcircuits presented have been definitions of the elements used in the encoding. To achieve correspondence with step and later process executions additional constraints need to be imposed. A step (process) execution has the property that only a single visible action is allowed to take place in a single component in each step. The arrangement to handle this is by using a cardinality constraint asserted true. An instance is given in Fig. 3 (the second one from the right).

The encoding may be further enhanced with a gate that disables idling. If such a gate is not added, the resulting circuit encodes step executions up to k whereas with it the executions are of precisely length k. Thus the gate limits the search space. As a downside short deadlocks may be missed if the verification process is started with too large a bound. Idling is disabled iff some visible action is executed, for the running example the gate is the rightmost in Fig. 3.

4.2 Synchronization

The synchronization of LTSs mandates that a visible action may be executed iff every LTS whose alphabet contains the action participates. So far, this has not been reflected in the subcircuits containing the input gates $ex(a,t)$. The condition is implemented by demanding that the executed action is enabled in each component having that label in its alphabet. An action a is enabled in a component iff it is in some state with an outgoing transition labeled a. The situation for the running example is illustrated in the two subcircuits on the left in Fig. 4.

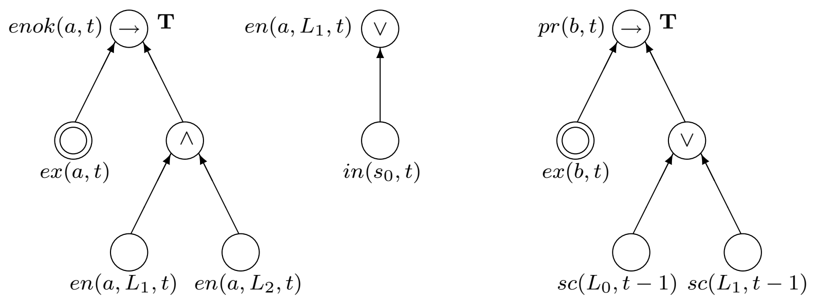

Fig. 4. Constraining the Input Gates (left, middle) and Enforcing Scheduling (right)

4.3 Translation Algorithm for Step Executions

Assume $L = (L_1, \ldots, L_n)$ and a given bound k. Then the algorithm constructing a Boolean circuit encoding step executions of L of length k is as

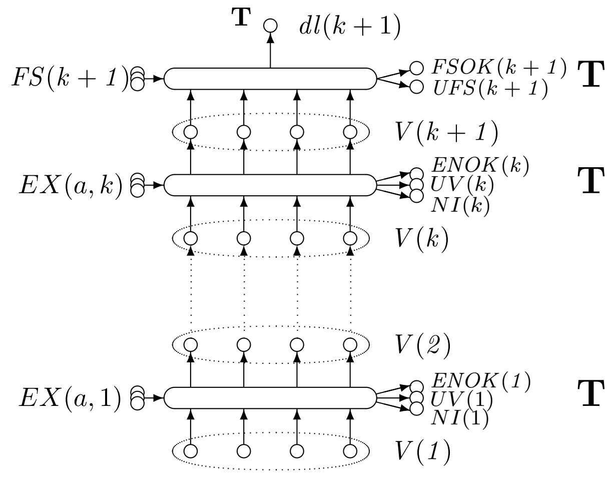

Fig. 5. Schematic Diagram of the Circuit

follows:

(i) To capture the requirement that each L_i is in the τ-closure of its initial states in V_1 add the gate $in(s, 1)$ for all states s and constrain them to true if the above condition holds and false otherwise.

(ii) For all positions $1 \leq t \leq k$, add the following subcircuits:
   (a) For all states $s \in S_1 \cup \ldots \cup S_n$, include the subcircuit for $in(s, t + 1)$.
   (b) For all the components L_j, add the subcircuit for $sc(L_j, t)$.
   (c) For all transitions with visible actions $l \in \Delta_1 \cup \ldots \cup \Delta_n$, add the subcircuit for $ex(l, t)$.
   (d) For each LTS L_j, add the subcircuit for $uv(L_j, t)$ and constrain it to true.
   (e) Add the circuit for $ni(t)$ and constrain it to true.
   (f) For all visible actions a, add the subcircuit for $enok(a, t)$ and constrain it to true.
   (g) For all components L_j, add the subcircuit for $en(a, L_j, t)$ for all its visible actions.

The structure of the circuit is schematically given in Fig. 5. In the bottom is the initial state $V(1)$ and on the top the last state $V(k + 1)$ and a circuit for deadlock detection (introduced in Sect. 4.5). The unconstrained input gates appear on the left and the constrained outputs on the right, the labels capitalized to indicate that they denote several actual gates.

Let $SC(L, k)$ be the (step) circuit obtained by the translation algorithm. Given a satisfying truth valuation α for $SC(L, k)$ call an α-execution the execution $V_1 \overset{A_1}{\rightarrow} \ldots \overset{A_k}{\rightarrow} V_{(k+1)}$ where the elements in each V_i are the states s with $\alpha(in(s, i)) =$ true and the elements in A_i the actions a having $\alpha(ex(a, i)) =$ true.

Theorem 4.1 *If the Boolean circuit $SC(L, k)$ has a satisfying truth valuation α, then there is an α-execution $V_1 \overset{A_1}{\rightarrow} \cdots \overset{A_k}{\rightarrow} V_{k+1}$ which is a step execution.*

Theorem 4.2 *If $V_1 \xrightarrow{A_1} \cdots \xrightarrow{A_k} V_{k+1}$ is a step execution of L, it is an α-execution for some satisfying valuation α of $SC(L,k)$.*

4.4 Process Executions

As can be seen from Definitions 2.6 and 2.11 the difference between step and process executions is rather simple. Indeed, the resulting circuit needs only one additional element. Namely, if an action is executed at $t+1$, then some participating component had to be scheduled in step t. On the right in Fig. 4 is an instance from the running example (executing action b in step 2).

The encoding algorithm needs the following addition for all $1 < t \leq k$.

(h) For all the visible actions $a \in \Sigma_1 \cup \cdots \cup \Sigma_n$ add the subcircuit $pr(a,t)$ and constrain it to true.

Figure 5 illustrates the circuit for step executions. Process executions would be modeled by adding the $PR(t)$ vector to the right hand side of the figure. Let $PR(L,k)$ be the (process) circuit obtained with the augmented algorithm.

Theorem 4.3 *If the Boolean circuit $PR(L,k)$ has a satisfying truth valuation α, then there is an α-execution $V_1 \xrightarrow{A_1} \cdots \xrightarrow{A_k} V_{k+1}$ which is a process execution.*

Theorem 4.4 *If $V_1 \xrightarrow{A_1} \cdots \xrightarrow{A_k} V_{k+1}$ is a process execution of L, it is an α-execution for some satisfying valuation α of $PR(L,k)$.*

4.5 Reachability Properties

In Corollaries 2.10 and 2.13 it is stated that both step and process executions preserve the final states of the executions. Therefore, any state predicate concerning such a state can be studied with the presented approach.[4] A deadlock, i.e., a state with no outgoing transitions, is a particularly interesting case among such properties.

The synchronizing system of LTSs can deadlock as a combination of two conditions. Firstly, components may end up in states with no outgoing transitions. Secondly, single components may indeed be able to proceed, but their synchronizing counterparts are in states where synchronization is not possible.

Thus a deadlock could be detected with circuits encoding such demands only based on the $in(s,k+1)$ gates. The former condition is simple and can be detected by static analysis. The latter is more difficult to encode compactly. Therefore, deadlock detection is implemented by introducing a

[4] There is a subtle issue which should be noted. The presented translation method assumes the following: if in a state s the state predicate to be studied holds, then in all states reachable from s by using only τ-moves the predicate also holds, i.e. you cannot get out of a "bad" state by using only τ-moves. If some τ-moves do not respect this property, they must be converted to visible actions before the verification is started.

new input gate, *fs*(s, L), for each component L and each state s with only visible outgoing actions.

The encoding is based on the reasoning that if there is a deadlocking interleaving execution, then the set of states reached in the associated step or process execution reaches a state $V_{(k+1)}$ such that the deadlocking state $S \sqsubset V_{(k+1)}$. The new input gate captures a single representative s from each component L so that if the gate *fs*(s, L) evaluates to true the state s is the representative from the component L in $V_{(k+1)}$.

The gate has to be constrained in the following way. Firstly, an obvious soundness criterion is that a state has to be in $V_{(k+1)}$ for it to be a candidate. Secondly, to mandate the collection of final states to be a state of the interleaving executions, they have to be constrained to precisely one in each component. Instances from the component L_1 of the running example are given on the left in Fig. 6.[5]

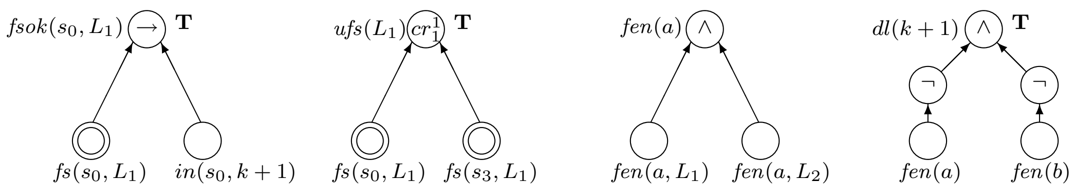

Fig. 6. State Predicate (Deadlock) Analysis

Having defined the *fs* gates a deadlock can be detected by the analysis of enabled actions in the final state. Whether the an action is enabled is computed with the same mechanism as in the previous steps (an instance given in the middle of Fig. 4). However, this is done for the chosen representative of the set of final states. Thus, the *in*(s, t) gate is replaced by the *fs*(s, L) gate and a new gate *fen*(a, L) standing for "action a is enabled in the final state in component L" is introduced.

An action is globally enabled in the final state iff it is enabled in the final states of all the components carrying transitions labeled with that action. Finally, a deadlock is a state where no action is globally enabled. The case for action a in the running example is illustrated on the right of Fig. 6, where the secong image from the right, defining the gate *fen*(a), encodes the cases when action a is enabled in the final state. The rightmost picture is the deadlock detection gate for the entire example, its inputs being the gates labeled *fen*(ac) for all the elements in the alphabet.

It should be noted that compared to the interleaving model, step and process executions may lose some of the intermediate states. However, it is not impossible to reason about them, provided that all state changes of interest to us can be observed through the occurrences of visible actions. The

[5] This idea of guessing a final state combination can be used to compactly encode arbitrary state predicates.

exact details of the following construction are left for further work, here just the main ideas are sketched. An additional component, called an observer automaton, can be added to the system. It observes the visible actions taking place by having all of them in its alphabet. Now any stuttering invariant safety property (which can be expressed as a regular language) can be reduced into the question of whether the observer automaton can reach a particular state. For the syntactic safety subset of LTL_{-X}, the linear temporal logic LTL without the next-time operator X, a finite automaton construction tool is available [13].

5 Test Results

To test the efficiency of the presented method it is compared against the following state-of-the-art implementations:

- NuSMV BMC, a BMC implementation using interleaving semantics,
- NuSMV BDD, an implementation of the BDD-based symbolic model checking.

A set of test cases has been adopted from [6] taking those cases known to deadlock. The test cases are provided as LTS (fsa), Promela and SMV specifications, the input format for NuSMV being the last one. Thus, the comparison task is easy. The results of the tests are given in Table 2 with the following columns:

- Problem instance,
- St. k, bound for step executions, i.e., the smallest number of steps such that a deadlock is reached,
- St. s, running time for step executions as measured by `/usr/bin/time`,
- Pr. k and Pr. s, similarly for process executions,
- SMV k and SMV s, bound and time for NuSMV BMC [5],
- SMV bdd, running time for NuSMV BDD [4].

The tests were carried out with an AMD Athlon machine with a 1400 MHz CPU and 1 Gigabyte of memory running the Linux operating system. With the problem Dartes, no results could be obtained within a reasonable time limit (1 hour) using either NuSMV BMC or NuSMV BDD, therefore the entries are of the form N/A.

The results for the Boolean circuits were obtained by using a tool [12] to translate LTSs to Boolean circuits and then using the BCZChaff system [10] which first translates a circuit to CNF DIMACS form [10] and then solves it with zChaff version 2001.2.17 [15]. The fact that both the presented method and NuSMV BMC use zChaff as the back end adds credibility to their comparison. The running time for the step and process executions is the sum of generating the Boolean circuit from the specifications and solving it for the

Table 2
Test Results (BCZCHAFF)

Problem	St. k	St. s	Pr. k	Pr.s	SMV k	SMV s	SMV BDD s
Dartes	31	2.0	31	0.53	N/A	N/A	N/A
DP(12)	1	0.028	1	0.028	> 8	830	0.12
Elev(1)	3	0.056	3	0.034	8	3.0	0.05
Elev(2)	5	0.16	5	0.11	11	3.1	0.17
Elev(3)	7	0.42	7	0.27	14	410	0.64
Elev(4)	9	1.6	9	0.74	17	120	2.7
Key(2)	35	510	35	200	> 30	2100	0.10
Key(3)	36	200	36	780	> 21	2700	0.27
Key(4)	37	10	37	11	> 19	3200	0.73
Key(5)	38	15	38	140	> 18	1900	3.2
Mmgt(3)	7	0.32	7	0.29	10	14	0.13
Mmgt(4)	8	0.77	8	0.35	12	73	0.25
Q(1)	9	0.25	9	0.25	> 11	1500	2.0
Hart(25)	50	1.2	50	0.71	51	7.0	0.12
Hart(50)	100	5.1	100	3.1	101	130	0.54
Hart(75)	150	12	150	7.6	151	990	1.9
Hart(100)	200	22	200	15	201	4800	5.5
Sentest(25)	33	0.63	33	0.7	38	4.2	0.12
Sentest(50)	58	2.1	58	2.3	63	40	0.45
Sentest(75)	83	4.5	83	5.1	88	220	1.5
Sentest(100)	108	8.0	108	8.3	113	980	4.6
Dac(15)	2	0.014	2	0.014	3	0.27	0.11
Speed(1)	4	0.038	4	0.030	7	0.13	0.07

given bound. The running time for NuSMV BMC is composed of generating the CNF instance and solving it for exactly the given bound.

Even though the test cases do not have a lot of non-determinism it can be seen that the non-standard execution models compare favorably in terms of the bound and running time to those of NuSMV BMC. Compared to BDD-based model checking the results reiterate the fact that BMC is at its best in finding short deadlocks.

Experiments indicate that with these examples it sometimes takes zChaff far longer to prove a formula unsatisfiable than to find a satisfying truth assignment with instances of comparable sizes. The phenomenon is most apparent in the example Key(4) where the time limit of one hour is exceeded with an unsatisfiable instance modeling process executions of length 29. The test cases and the tool translating LTSs to Boolean circuits are available for download at [12].

6 Conclusions and Related Work

The paper studies bounded model checking of reachability properties of a system represented as consisting of synchronizing LTSs. Two nonstandard execution models, step and process executions, are proposed to capture sets of interleaving executions in a compact form.

The paper presents two translation schemes from LTSs to Boolean circuits. In the first case, the resulting circuit encodes precisely the step executions of the system of LTSs under consideration and in the second the process executions. The encoding is compact leading to a circuit linear in the size of the bound k, more precisely $\mathcal{O}((\sum_j(|S_j| + |\Delta_j| + |\Gamma_j|)) \cdot k)$ where S_j, Δ_j and Γ_j are the state space, transition relation and visible actions of LTS L_j, respectively. The encoding uses Boolean functions outside traditional propositional logic, namely cardinality constraints of the form cr^1_0 and cr^1_1, but the bound holds were the use of them disallowed. Such a function with indegree i can namely be simulated using $\mathcal{O}(i)$ new $\vee, \wedge$ and $\neg$ gates. The approach is backed by a set of test cases showing that the running times compare favorably to a state-of-the-art interleaving BMC implementation in the NuSMV system.

The presented approach is considered only for models where the LTSs are presented explicitly. Translations from symbolical representations, like SMV models, is an interesting research problem for future work.

The idea for the paper arose as a comparison to the work done in [8]. The paper presents a BMC procedure to reachability check 1-safe Petri nets with step and process semantics. In addition to the different modeling the approach does not use determinization and neither does it as efficiently handle τ-actions (there they contribute to the execution length). The paper considers some of the same examples presented here. However, a direct comparison was omitted due to some inconsistencies in the state spaces of the fsa and 1-safe Petri net models. The differences could be traced to the fsa to 1-safe Petri net conversion performed in [14].

So far, only the verification of reachability properties has been considered, whereas LTL_{-X} model checking is left for future work. In [9] a translation of LTL_{-X} for step semantics is given using a logic programming approach.

Acknowledgement

The authors would like to warmly thank T. A. Junttila for creating the tool BCZCHAFF, and for fruitful discussions.

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*

(TACAS'1999), volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[2] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *Formal Methods in Computer Aided Design*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer, November 2000.

[3] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in Alpha microprocessor using satisfiability solvers. In *Proceedings of the 13th International Conference of Computer Aided Verification(CAV'2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer, 2001.

[4] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proceeding of the 14th International Conference on Computer-Aided Verification (CAV'2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, July 2002.

[5] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Integrating BDD-based and SAT-based symbolic model checking. In *Proceedings of 4th International Workshop on Frontiers of Combining Systems*, April 2002.

[6] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), 1996.

[7] V. Diekert and Y. Métivier. Partial commutation and traces. In *Handbook of formal languages, Vol. 3*, pages 457–534. Springer, Berlin, 1997.

[8] K. Heljanko. Bounded reachability checking with process semantics. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR'2001)*, pages 218–232, August 2001.

[9] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming (TPLP)*, 2003. Accepted for publication. (CoRR: arXiv:cs.LO/0305040).

[10] T. A. Junttila. Boolean circuit tools (including BCZChaff), May 2003. `http://www.tcs.hut.fi/~tjunttil/circuits`.

[11] T. A. Junttila and I. Niemelä. Towards an efficient tableau method for boolean circuit satisfiability testing. In *Computational Logic - CL 2000; First Internatinal Conference*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 553–567, London, UK, July 2000. Springer, Berlin.

[12] T. Jussila. A BMC tool translating LTSs to boolean circuits, May 2003. `http://www.tcs.hut.fi/~tjussila/otf`.

[13] T. Latvala. Efficient model checking of safety properties. In *Model Checking Software. 10th International SPIN Workshop*, volume 2648 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2003.

[14] S. Melzer and S. Römer. Deadlock checking using net unfoldings. In *Proceeding of 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 352–363. Springer, June 1997.

[15] M. Moskewicz, Y. Madigan, L. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference*, July 2001.