BMC via On-the-Fly Determinization

Toni Jussila*, **Keijo Heljanko** **, **Ilkka Niemelä*****

Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O. Box 5400, FI-02015 HUT, Finland

The date of receipt and acceptance will be inserted by the editor

Abstract. This paper develops novel bounded model checking (BMC) techniques for asynchronous parallel systems. The aim is to increase the ef£ciency of BMC by exploiting the inherent concurrency in such systems. This added ef£ciency is gained by covering more reachable states within a given bound using two techniques. Firstly, a non-standard execution model, step executions, where multiple actions can take place simultaneously is applied. Secondly, the number of executions the system can have is reduced by modeling the execution of the system components as if they were determinized. This determinization technique also enables the removal of the internal transitions of the components. Step executions can be further restricted to a subclass called process executions without losing any reachable states.

The paper presents a translation scheme for bounded model checking of reachability properties. The translation is from an asynchronous system where the components are modeled as labeled transition systems (LTSs) to a propositional formula. The models of the formula correspond to the step executions of the original system where each component is replaced with its determinized counterpart. The formula for step executions can be easily extended in such a way that its models correspond to the process executions of the system. The translation scheme has been implemented and some experimental comparisons performed. The results show that the bound needed to detect a violation of a reachability property is for step and process executions 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 Introduction

The purpose of the paper is to develop ef£cient bounded model checking (BMC) techniques for asynchronous systems modeled as labeled transition systems (LTSs). BMC is a veri£cation technique that considers only executions of bounded length of the chosen formalism [1]. The general model checking problem for properties speci£ed in linear temporal logic (LTL) is known to be **PSPACE**-complete w.r.t. the system description given, for instance, as an LTS system studied in this paper. However, the bounded case is in **NP** (assuming the used bound is given in unary encoding). The very idea is to compile the system under veri£cation, the property to be veri£ed 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 this work is to increase the ef£ciency of BMC by exploiting the inherent concurrency in asynchronous systems. The standard approach to such systems is to use interleaving executions, where exactly one action is occurring at a time. For example, consider the system in Figure 1. It presents $2n$ LTSs following the standard notation of presenting the states as circles and the transitions from a state to another as arrows. The arrows are labeled with symbols like a_1 or τ.

The $2n$ components form a system whose global states (denoted by **s**) are $2n$-tuples of local states, one local state from each component. The initial global state is the tuple
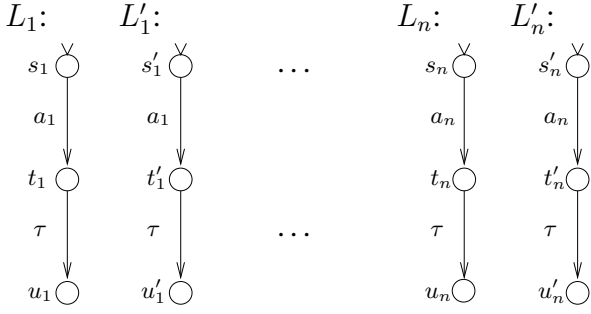
Fig. 1. Example system

Fig. 2. On-the-£y determinization

where each component is in its initial state (marked in the picture with a wedge and labeled s_i or s'_i where $1 \leq i \leq n$).

The system can move from the global state $\mathbf{s}$ to the global state $\mathbf{s}'$ with label a_i (denoted $\mathbf{s} \xrightarrow{a_i} \mathbf{s}'$) iff every component having a_i among its labels can execute a transition labeled a_i. That means that before the transition each such component is in a local state having an outgoing transition labeled a_i. In the next global state each of these components is in the target state of that transition while each other component remains in its original state. However, the label τ is an exception to the rule. It denotes an internal transition and thus a single component can execute its τ transitions in isolation. An interleaving execution of the example system reaching the global state $\langle u_1, u'_1, \ldots, u_n, u'_n \rangle$ from the initial state $\langle s_1, s'_1, \ldots, s_n, s'_n \rangle$ is as follows:

$$\langle s_1, s'_1, \ldots, s_n, s'_n \rangle \xrightarrow{a_1} \langle t_1, t'_1, \ldots, s_n, s'_n \rangle \xrightarrow{a_2}$$
$$\cdots \xrightarrow{a_n} \langle t_1, t'_1, \ldots, t_n, t'_n \rangle \xrightarrow{\tau} \langle u_1, t'_1, \ldots, t_n, t'_n \rangle \xrightarrow{\tau}$$
$$\cdots \xrightarrow{\tau} \langle u_1, u'_1, \ldots, u_n, u'_n \rangle$$

Here the idea is to encode interleaving executions more compactly by allowing multiple occurrences of actions in different components of the system simultaneously. For the example in Figure 1 with the interleaving model the number of steps needed to reach every state of the system is $3n$. If simultaneous executions of independent actions are allowed only 2 steps are needed (independent of the value of n).

The approach of allowing independent actions to take place simultaneously is further combined with an on-the-£y determinization construction where for each component a set of states in which that component can be is maintained. The situation is illustrated in Figure 2. On the left hand side is a non-deterministic LTS and on the right hand side its determinized counterpart. The purpose of this construction is to (i) reduce the number of executions of the system, and (ii) shorten their length by removing internal transitions.

In this work, the concurrent execution of independent actions combined with on-the-£y determinization of components is referred to as *step executions*. The idea is illustrated in the following example execution of the system from Figure 1:

$$\langle \{s_1\}, \{s'_1\}, \ldots, \{s_n\}, \{s'_n\} \rangle \xrightarrow{\{a_1, \ldots, a_n\}} \quad (1)$$
$$\langle \{t_1, u_1\}, \{t'_1, u'_1\}, \ldots, \{t_n, u_n\}, \{t'_n, u'_n\} \rangle$$
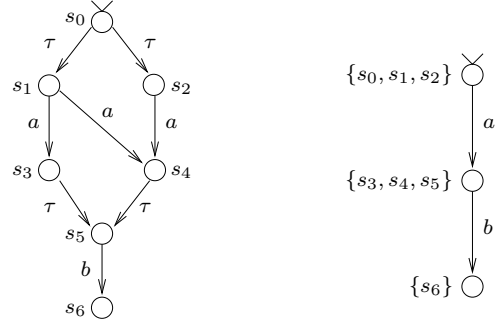
The execution is different from the interleaving model in that the actions $a_1, \ldots, a_n$ are executed simultaneously. Secondly, due to on-the-£y determinization component L_1, for instance, reaches the set of states $\{t_1, u_1\}$. Without compromising reachable states, step executions can be further restricted to *process executions* satisfying an extra condition on visible actions.

Based on these ideas, a bounded model checking procedure of reachability properties of an LTS system is developed by devising a translation scheme from the LTSs to a propositional formula. The novelty point of the translation is that the models of the formula are the step executions of the system, i.e., in the models (interpreted as executions):

– several independent actions can be executed simultaneously and
– in each execution state each component can be in a *set* of its local states, i.e., the formula models the executions of the determinized version of the component.

Yet, the size of the formula remains linear both w.r.t. the bound and the system description. In addition, the translation can be done without constructing explicitly the determinized versions of the components, i.e., handling determinization on-the-£y. A simple addition to the formula modeling step executions results in a formula modeling process executions.

The approach has been applied to a set of deadlock checking problems and the data obtained justify the following points. Firstly, step and process executions need in most cases a lower bound to detect a deadlock than 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 Section 3 presents the encoding schemes for both execution models. Section 4 gives test results comparing step and process executions to NuSMV [4,5] and £nally Section 5 concludes.

2 System Modeling Formalism

This paper studies asynchronous concurrent systems speci-£ed as labeled transition systems (LTS). Three execution mod-

els for a system of LTSs are introduced. The £rst is the standard interleaving semantics. Thereafter, the step and process models allowing independent actions to take place simultaneously are de£ned. The section ends with an analysis of the relation between the different models.

De£nition 1. An LTS is a 4-tuple $L = (S, I, \Gamma, \Delta)$ where

- S is a non-empty set of (local) 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 (local) transitions of L, where $\tau \notin \Gamma$.

The transitions whose middle component is τ are called *internal* or invisible to the environment. LTSs can interact by forming a synchronizing system, denoted here by $L = \langle L_1, \ldots, L_n \rangle$. Its states are n-tuples of local states and semantics is de£ned in terms of interleaving executions.

De£nition 2. Let $L = \langle L_1, \ldots, L_n \rangle$ be a system of synchronizing LTSs, where $L_i = (S_i, I_i, \Gamma_i, \Delta_i), 1 \le i \le n$. An *interleaving* execution of L is a sequence

$$\mathbf{s}_1 \xrightarrow{a_1} \mathbf{s}_2 \xrightarrow{a_2} \cdots \xrightarrow{a_k} \mathbf{s}_{k+1} \qquad (2)$$

such that each *global state* $\mathbf{s}_i$ is an n-tuple $\langle s_i^1, \ldots, s_i^n \rangle \in S_1 \times \cdots \times S_n$, i.e., s_i^j is a local state of LTS L_j and each $a_i \in \Gamma_1 \cup \cdots \cup \Gamma_n \cup \{\tau\}$. Furthermore, the following holds:

1. $\langle s_1^1, \ldots, s_1^n \rangle \in I_1 \times \cdots \times I_n$.
2. For all $1 \le i \le k$, if $a_i \ne \tau$, then for the transition $\langle s_i^1, \ldots, s_i^n \rangle \xrightarrow{a_i} \langle s_{i+1}^1, \ldots, s_{i+1}^n \rangle$ it holds that for all L_j, if $a_i \in \Gamma_j$, $(s_i^j, a_i, s_{i+1}^j) \in \Delta_j$ otherwise $s_{i+1}^j = s_i^j$.
3. For all $1 \le i \le k$, if $a_i = \tau$, then for the transition $\langle s_i^1, \ldots, s_i^n \rangle \xrightarrow{a_i} \langle s_{i+1}^1, \ldots, s_{i+1}^n \rangle$ it holds that there is an L_j such that $(s_i^j, \tau, s_{i+1}^j) \in \Delta_j$ and $s_{i+1}^k = s_i^k$ for $k \ne j$.

When explicit presentation of intermediate states is not of great importance, the notation $\mathbf{s}_i \xrightarrow{a_i, \ldots, a_j} \mathbf{s}_{j+1}$ is used to denote a part of an interleaving execution from $\mathbf{s}_i$ to $\mathbf{s}_{j+1}$ such that the executed actions are $a_i, \ldots, a_j$ in that order.

The de£nition above is usually given by £rst de£ning the synchronized product of the components constituting the system and then presenting the executions using that construction. De£nition 2 above is equivalent and better suited for comparing the traditional model to the new contributions presented below (De£nitions 6 and 8). These de£nitions make use of the following concepts.

De£nition 3. The concatenation of the visible actions in interleaving execution σ_I in the order mandated by σ_I is denoted $vis(\sigma_I)$.

De£nition 4. A global state $\mathbf{s}'$ is *reachable* in the LTS system $L = \langle L_1, \ldots, L_n \rangle$ iff $\mathbf{s}'$ is one of the global initial states $\mathbf{s}' \in I_1 \times \cdots \times I_n$ or there is an interleaving execution σ_I from a global initial state $\mathbf{s}$ to $\mathbf{s}'$. A global state $\mathbf{s}'$ is a *deadlock* state iff it is reachable and no execution reaching $\mathbf{s}'$ can be extended with some transition.

Any LTS can be seen as a labeled graph. Thus, a labeled path in an LTS is just a labeled path in the LTS seen as a graph.

De£nition 5. Let $L = (S, I, \Gamma, \Delta)$ and $S' \subseteq S$. The τ-*closure* of S' is the maximal set of states $S'' \subseteq S$ such that $s \in S''$ iff $s \in S'$ or there is a path from some state in S' to s labeled only with τ transitions.

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

De£nition 6. Let $L = \langle L_1, \ldots, L_n \rangle$ be a system of synchronizing LTSs, where $L_i = (S_i, I_i, \Gamma_i, \Delta_i), 1 \le i \le n$. A £nite *step* execution σ_S of L is a sequence

$$\mathbf{S}_1 \xrightarrow{A_1} \mathbf{S}_2 \xrightarrow{A_2} \cdots \xrightarrow{A_k} \mathbf{S}_{k+1} \qquad (3)$$

such that each $\mathbf{S}_i$, called a *determinized global state*, is an n-tuple $\langle S_i^1, \ldots, S_i^n \rangle, S_i^j \subseteq S_j, 1 \le j \le n$, 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:

1. In $\mathbf{S}_1$ every S_1^j is the τ-closure of I_j.
2. For each A_i and L_j, $|A_i \cap \Gamma_j| \le 1$, i.e., in each step at most one visible action is executed from each LTS.
3. 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 $s' \in S_i^j$.
4. 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 $|\sigma_S|$, is k. Let $lin(\sigma_S)$ 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 for each $1 \le i \le k$, $\alpha_i \in lin(A_i)$ where $lin(A_i)$ is the set of strings obtained by concatenating the elements in A_i in all possible orders.

The above execution model based on on-the-¤y determinization could be alternatively de£ned by requiring the components of the LTS system to be deterministic or by determinizing them using the standard subset construction [17]. However, the subset construction is potentially expensive since the determinized version of a non-deterministic £nite automaton of n states can have as many as 2^n states. On-the-¤y determinization avoids this without substantially increasing the complexity of the BMC encoding.

De£nition 7. Given an LTS system $L = \langle L_1, \ldots, L_n \rangle$, its global state $\mathbf{s} = \langle s_1, \ldots, s_n \rangle$ and its determinized global state $\mathbf{S} = \langle S_1, \ldots, S_n \rangle$, de£ne $\mathbf{s} \sqsubset \mathbf{S}$ to mean that for all $1 \le j \le n, s_j \in S_j$.

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

Theorem 1. *Let σ_{I} be an (interleaving) execution of L of the form (2) and $|\sigma_{\mathrm{I}}| = k$. Then there is a step execution σ_{S}*

$$\mathbf{S}_1 \xrightarrow{\{a'_1\}} \mathbf{S}_2 \xrightarrow{\{a'_2\}} \dots \xrightarrow{\{a'_{l-1}\}} \mathbf{S}_l \xrightarrow{\{a'_l\}} \mathbf{S}_{l+1} \tag{4}$$

of L such that $a'_1 a'_2 \dots a'_l = vis(\sigma_{\mathrm{I}})$, $l \leq k$ and $\mathbf{s}_{k+1} \sqsubset \mathbf{S}_{l+1}$.

Proof. Any interleaving execution of length k can be represented in the following form:

$$\mathbf{s}_{1,0} \xrightarrow{\tau^*} \mathbf{s}_{1,n_1} \xrightarrow{a'_1} \mathbf{s}_{2,0} \xrightarrow{\tau^*} \mathbf{s}_{2,n_2} \cdots \xrightarrow{a'_l} \mathbf{s}_{l+1,0} \xrightarrow{\tau^*} \mathbf{s}_{l+1,n_{l+1}}$$

The notation τ^* is a regular expression meaning zero or more τ-actions. Each global state in the interleaving execution is represented in the form $\mathbf{s}_{i,j}$. Index i is incremented every time a visible action is executed. Index j, on the other hand, is incremented every time τ is executed and reset to zero every time a visible action is executed. The idea is to match the global states $\mathbf{s}_{i,j}$ to the determinized global state $\mathbf{S}_i$ (compressing the τ transitions between the different $\mathbf{s}_{i,j}, 0 \leq j \leq n_i$ to $\mathbf{S}_i$ containing the τ-closure).

The actions in the execution can be represented as the string $\tau^* a'_1 \tau^* a'_2 \dots a'_l \tau^*$. The corresponding step execution is constructed by omitting all the τ transitions in the execution and for each visible transition executing the step consisting of the singleton set $\{a'_i\}$, where $1 \leq i \leq l$. Thus, the execution is of the form (4) above.

The proof proceeds by induction over the determinized global states $\mathbf{S}_i$ of the step execution establishing that the step execution can be constructed and proving the following stronger invariant for all the visited states $\mathbf{s}_{i,j}$ of the interleaving execution:

for all $i, 1 \leq i \leq l+1$, $\mathbf{s}_{i,j} \sqsubset \mathbf{S}_i, 0 \leq j \leq n_i$.

1. **Base Case.** In any step execution $\mathbf{S}_1$ is the τ-closure of the initial states of the system. For every interleaving execution $\mathbf{s}_{1,j} \sqsubset \mathbf{S}_1$, where $0 \leq j \leq n_1$, because $\mathbf{s}_{1,0}$ is one of the system's initial states and any subsequent state is reached from it by executing only τ transitions.
2. **Induction Hypothesis.** Assume $\mathbf{s}_{i,j} \sqsubset \mathbf{S}_i$ for some i and for all $0 \leq j \leq n_i$.
3. **Induction Step.** Consider the case $i + 1$. By induction hypothesis, $\mathbf{s}_{i,n_i} \sqsubset \mathbf{S}_i$. Thus, if $\mathbf{s}_{i,n_i} \xrightarrow{a'_i} \mathbf{s}_{i+1,0}$ is a valid transition in σ_I, then $\mathbf{S}_i \xrightarrow{\{a'_i\}} \mathbf{S}_{i+1}$ is a valid step in the sense that in every component having a'_i in their alphabet at least one transition labeled a'_i is enabled. By definition of a step execution, $\mathbf{S}_{i+1}$ contains the τ-closure of all the states reachable from the states in $\mathbf{S}_i$ by executing transitions labeled a'_i. Thus, $\mathbf{s}_{i+1,0} \sqsubset \mathbf{S}_{i+1}$. Every subsequent $\mathbf{s}_{i+1,j}$ is reached by executing τ transitions. Thus, by definition of a τ-closure $\mathbf{s}_{i+1,j} \sqsubset \mathbf{S}_{i+1}, 0 \leq j \leq n_{i+1}$.

It should be easy to see that $vis(\sigma_{\mathrm{I}}) = a'_1 \dots a'_l$ and that the length of the resulting step execution can only shrink compared to σ_{I}. $\quad\square$

Theorem 2. *Let σ_{S} be a step execution of a system L reaching $\mathbf{S}_{k+1}$. Then for any global state $\mathbf{s}$ of the system L such that $\mathbf{s} \sqsubset \mathbf{S}_{k+1}$ there is an interleaving execution σ_{I} of L reaching $\mathbf{s}$ such that $vis(\sigma_{\mathrm{I}}) \in lin(\sigma_{\mathrm{S}})$.*

Proof. Consider a step execution σ_{S} of the form:

$$\mathbf{S}_1 \xrightarrow{A_1} \mathbf{S}_2 \xrightarrow{A_2} \dots \xrightarrow{A_k} \mathbf{S}_{k+1},$$

where each $A_i = \{a_{i,1}, \dots, a_{i,m_i}\}$. It is shown that from σ_{S} it is possible to construct an interleaving execution σ_{I} of the form

$$\mathbf{s} \xrightarrow{\tau^*} \mathbf{s}_{1,0} \xrightarrow{a_{1,1}\cdots a_{1,m_1}\tau^*}$$
$$\mathbf{s}_{2,0} \xrightarrow{a_{2,1}\cdots a_{2,m_2}\tau^*} \cdots \xrightarrow{a_{k,1}\cdots a_{k,m_k}\tau^*} \mathbf{s}_{k+1,0}$$

such that $\mathbf{s}_{k+1,0}$ is the state the interleaving should reach. Similarly as in the proof of Theorem 1 the states of the interleaving execution are represented in the form $\mathbf{s}_{i,j}$. However, since the proof constructs the interleaving backwards, the indexing has been set to better adhere to the construction. In this case, the index i is incremented and j reset when the interleaving execution is in a state that is immediately followed by the execution of the *first* action in the chosen linearization of a particular step in the given step execution. Otherwise, the index j is incremented.

It should be noted that compared to the proof of Theorem 1 there may be several interleaving executions of the form above (with the same actions). The construction proceeds inductively from the end of σ_{S} and proves that from any $\mathbf{s}_{i,0}$ (the suffix of) σ_{I} can be constructed and the following invariant holds:

for all $i = k+1, \dots, 1$, $\mathbf{s}_{i,0} \sqsubset \mathbf{S}_i$.

1. **Base Case.** Trivially, the interleaving execution from a state to itself can be constructed. In addition, by definition $\mathbf{s}_{k+1,0} \sqsubset \mathbf{S}_{k+1}$.
2. **Induction Hypothesis.** Assume that (the suffix of) σ_{I} can be constructed from $\mathbf{s}_{i,0}$ and that $\mathbf{s}_{i,0} \sqsubset \mathbf{S}_i$.
3. **Induction Step.** Consider the case $i - 1$. Consider all the states reachable by executing from $\mathbf{s}_{i,0}$ action sequences of the form $\tau^* a_{i,m_i} \dots a_{i,1}$ *backwards*. Since the step execution has a step $\mathbf{S}_{i-1} \xrightarrow{\{a_{i,1}, \dots, a_{i,m_i}\}} \mathbf{S}_i$ and by induction hypothesis $\mathbf{s}_{i,0} \sqsubset \mathbf{S}_i$, there is at least one state $\mathbf{s}'$ (backwards) reachable from $\mathbf{s}_{i,0}$ with such an action sequence such that $\mathbf{s}' \sqsubset \mathbf{S}_{i-1}$. It follows naturally that the construction proceeds by letting $\mathbf{s}_{i-1,0} = \mathbf{s}'$. The intermediate states $\mathbf{s}_{i-1,1}, \dots, \mathbf{s}_{i-1,n_{i-1}}$ are assigned according to the visited states during the backwards processing of the sequence. This defines σ_{I} from $\mathbf{s}_{i-1,0}$ to $\mathbf{s}_{i,0}$ continuing from there by the induction hypothesis. If there are several possibilities for choosing $\mathbf{s}'$, an arbitrary one can be picked.

Thus, σ_{I} can be constructed from $\mathbf{s}_{1,0}$ and $\mathbf{s}_{1,0} \sqsubset \mathbf{S}_1$. Since $\mathbf{S}_1$ is the τ-closure of the initial states, it is possible to find a τ-path from the initial state $\mathbf{s}$ of the interleaving to $\mathbf{s}_{1,0}$

and the construction is complete. It should be noted that the several visible actions in each step in σ_S can be interleaved in *any* order since they are independent. Changing the order obviously changes the intermediate states in the interleaving execution. However, $vis(\sigma_I) \in lin(\sigma_S)$ as stated in the theorem. $\quad\square$

Corollary 1. *A global state* $\mathbf{s}$ *of* $L = \langle L_1, \ldots, L_n \rangle$ *is reachable iff for some* k *there is a step execution* $\mathbf{S}_1 \overset{A_1}{\to} \mathbf{S}_2 \overset{A_2}{\to} \cdots \overset{A_k}{\to} \mathbf{S}_{k+1}$ *such that* $\mathbf{s} \sqsubset \mathbf{S}_{k+1}$.

The set of step executions for a system contains in most cases different executions intuitively corresponding to the same concurrent behavior. The following addition to Definition 6 limits the number without compromising reachable states.

Definition 8. A *process* execution of system L is a step execution of L where every action a fulfills the following *process condition*:

1. $a \in A_1$ or
2. $i > 1$, $a \in A_i$ and L has a component L_j such that $a \in \Gamma_j$ 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 component having action a in its alphabet would be in a state where a could take place. It would not, though, be chosen for immediate execution but the components would remain idle in the same states for some steps and only then execute a. For instance, the execution

$$\langle \{s_1\}, \{s_1'\}, \ldots, \{s_n\}, \{s_n'\} \rangle \overset{\{a_1, \ldots, a_{n-1}\}}{\to}$$
$$\langle \{t_1, u_1\}, \ldots, \{t_{n-1}', u_{n-1}'\} \{s_n\}, \{s_n'\} \rangle \overset{\{a_n\}}{\to}$$
$$\langle \{t_1, u_1\}, \{t_1', u_1'\}, \ldots, \{t_n, u_n\}, \{t_n', u_n'\} \rangle$$

of the system in Figure 1 is not a process execution since nothing prevents a_n from occurring in the first step. The execution in (1), on the other hand, is both a step and a process execution.

Theorem 3. *Let* σ_S *be a step execution reaching the determinized global state* $\mathbf{S}$. *Then there is a process execution* σ_P *reaching* $\mathbf{S}$ *such that* $|\sigma_P| \leq |\sigma_S|$.

Proof. The construction of the corresponding process execution is as follows:

1. If the step execution is a process execution, then terminate.
2. Take the smallest i such that A_i contains some action a for which the process condition is not fulfilled.
3. Push action a one step earlier and modify the determinized global state accordingly. If the remaining step A_i is empty, then remove it from the step execution. Goto step 1.

By definition of the process condition, if action a violates the process condition, then no component with a in its alphabet executes a transition in A_{i-1}. However, this implies that
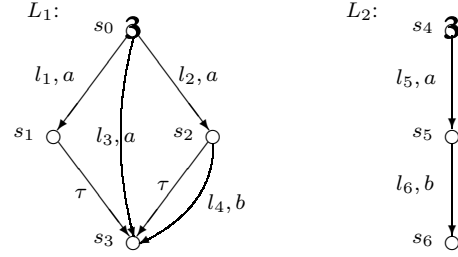

Fig. 3. Running Example

every such component is in the same local state before and after A_{i-1}. Thus, the execution obtained by moving a from set A_i to set A_{i-1} and possibly removing A_i is a valid step execution.

The procedure above is bound to terminate. This due to the fact that the violating action a is pushed one step earlier. Thus, in the worst case it is pushed all the way back to A_1. However, in A_1 every action fulfills the process condition. Secondly, there is only a finite number of actions to be pushed.

Finally, in both executions all the components execute the same transitions in the same order. Thus, they are bound to reach the same state. $\square$

Corollary 2. *A global state* $\mathbf{s}$ *of* $L = \langle L_1, \ldots, L_n \rangle$ *is reachable iff for some* k *there is a process execution* $\mathbf{S}_1 \overset{A_1}{\to} \mathbf{S}_2 \overset{A_2}{\to} \cdots \overset{A_k}{\to} \mathbf{S}_{k+1}$ *such that* $\mathbf{s} \sqsubset \mathbf{S}_{k+1}$.

Intuitively, 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 3 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 l_i. In the figure, that label is given together with the action associated with the transition.

3 Encoding

This section presents the structure of the Boolean formula encoding the step and process executions of a system of LTSs. For representational purposes the literals that appear are given certain names collected in Table 1 for reference. An in-depth description of them follows in subsequent sections.

The encoding assumes that the LTSs do not have loops containing only τ transitions involving more than one state. To guarantee that this is the case, each component can be preprocessed in a way which ensures that the resulting LTS simulates all the executions of the original. The preprocessing step computes the maximal strongly connected components

Table 1. Translation Atoms

Atoms	Description
$ex(ac, t)$	Action ac is executed at time t.
$in(s, t)$	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.
$en(ac, L, t)$	Action ac is enabled in component L at time t.

C_i of the LTS restricted to τ transitions and replaces each such τ-*component* C_i with a single state s'_i having as incoming and outgoing transitions the union of those in the set of states in C_i. Let $repr()$ be a function from the set of local states to the set of states representing τ-components such that for each $s \in C_i, repr(s) = s'_i$. Then, if the original system has a global state $\mathbf{s} = \langle s_1, \ldots, s_n \rangle$, the modified system simulates this with the global state $\mathbf{s}' = \langle repr(s_1), \ldots, repr(s_n) \rangle$.

In addition, the presentation applies the following subformulas for reasons of compactness and readability. The formulas

$$[0, 1]\{a_1, \ldots, a_n\}$$
$$[1, 1]\{a_1, \ldots, a_n\}$$

evaluate to true iff exactly zero or one (the former) or precisely one (the latter) of the literals in the set $\{a_1, \ldots, a_n\}$ are true. Note that they can be simulated using $\mathcal{O}(n)$ connectives and new variables with traditional propositional logic.

The details of the encoding are presented in the rest of the section. The formulas are given as propositional schemes that have free variables. Those variables have to be instantiated for particular elements of the system model, a procedure described together with the formulas. However, due to the fact that every scheme has to be instantiated for the same time steps, from 1 to k, the unrolling depth, that description is omitted for the free variable t. The presentation also follows the convention of abbreviating certain conjunctions and disjunctions as universally and existentially quantified formulas, respectively.

3.1 Control Flow

For encoding the control flow of the LTSs the idea is that the $in(s, t)$ literals serve to provide information regarding the progress of the execution. For the (statically computed) τ-closure of the initial states of an LTS the literal $in(s, 1)$ is asserted true. This is in accordance with the fact that in the outset the execution in each component is in the τ-closure of its 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 executing some of its incoming visible transitions in step t.

The resulting scheme is a disjunction encoding the three cases above, the disjuncts below appearing in the same order as in the list above.

$$in(s, t + 1) \leftrightarrow (in(s, t) \wedge \neg sc(L, t)) \vee$$
$$(\exists s_1 in(s_1, t + 1) \wedge sc(L, t)) \vee$$
$$\exists s_2 (in(s_2, t) \wedge \exists l\, ex(l, t)) \qquad (5)$$

where
- s varies over the states of the system,
- L refers to the component where s is in,
- s_1 quantifes over each state from which s is reachable via a τ transition,
- s_2 quantifes over all the states from which there is a transition with a visible action to s, and
- l quantifes over the visible transitions from s_2 to s.

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

$$sc(L, t) \leftrightarrow \exists ac_1 ex(ac_1, t) \qquad (6)$$

where
- L varies over the components of the system and
- ac_1 quantifes over the actions in L.

The reasoning behind the latter, the $ex(l, t)$ literal, is as follows. A transition is executed at time t iff the action it is labeled with is executed 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 $t + 1$. Formally:

$$ex(l, t) \leftrightarrow ex(ac, t) \wedge in(sr, t) \qquad (7)$$

where
- l varies over the visible transitions of the system,
- ac refers to the action with which l is labeled, and
- sr refers to the source states of l.

Instantiating the schemes for the running examples yields for instance the following formulas. Firstly, the formula encoding that local state s_3 in component L_1 is reached in global state 2 is the following:

$$in(s_3, 2) \leftrightarrow (in(s_3, 1) \wedge \neg sc(L_1, 1)) \vee$$
$$((in(s_1, 2) \vee in(s_2, 2)) \wedge sc(L_1, 1)) \vee$$
$$((in(s_0, 1) \wedge ex(l_3, 1)) \vee$$
$$(in(s_2, 1) \wedge ex(l_4, 1)))$$

The formula encoding the fact that component L_1 is scheduled in the first step is as follows:

$$sc(L_1, 1) \leftrightarrow ex(a, 1) \vee ex(b, 1)$$

Finally, the execution of transition l_3 translates to the following instance:

$$ex(l_3, 1) \leftrightarrow ex(a, 1) \wedge in(s_0, 1)$$

3.2 Executed Actions

So far, the subformulas 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 execution has the property that at most one single visible action is allowed to take place in a single component in each step. The arrangement to handle this is to use a cardinality constraint as follows:

$$[0, 1]\{ex(ac_1, t), \ldots, ex(ac_n, t)\} \tag{8}$$

of which

- an instance is needed for all the components of the system and
- $ac_1, \ldots, ac_n$ refer to actions in a *particular* component.

The encoding may be further enhanced with a propositional scheme that disables idling i.e., steps where no visible action is executed. The formula is the following:

$$\exists ac\, ex(ac, t) \tag{9}$$

where ac quantifies over the visible actions of the system.

If the element above is not added, the resulting formula encodes step executions up to k whereas with it the executions are of precisely length k. Thus, the formula limits the search space. As a downside short deadlocks may be missed if the verification process is started with too large a bound.

3.3 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 subformulas containing the variables $ex(ac, t)$. Therefore, additional constraints are needed. The condition is implemented by demanding that the executed action is enabled in each component having that label in its alphabet. An action ac is enabled in a component iff it is in some state with an outgoing transition labeled ac. Formally:

$$en(ac, L, t) \leftrightarrow \exists s\, in(s, t) \tag{10}$$

where

- L varies over the components of the system,
- ac varies over the visible actions of component L, and
- s quantifies over the states in L having an outgoing transition labeled ac.

The variable denoting the execution of a single action is then constrained as follows:

$$ex(ac, t) \rightarrow \forall L\, en(ac, L, t) \tag{11}$$

where

- ac varies over the visible actions of the system, and
- L quantifies over the components having ac in their alphabet.

Taking instances from the running example, the fact that action a is enabled in component L_1 in the initial state is encoded by the following instance:

$$en(a, L_1, 1) \leftrightarrow in(s_0, 1)$$

Furthermore, executing action b can only take place if it is enabled in both the components:

$$ex(b, 1) \rightarrow en(b, L_1, 1) \wedge en(b, L_2, 1)$$

Having thus presented all the propositional schemes, the next subsection gathers them to a complete translation algorithm.

3.4 Translation Algorithm for Step Executions

Assume $L = \langle L_1, \ldots, L_n \rangle$ with $L_i = (S_i, I_i, \Gamma_i, \Delta_i)$, $1 \leq i \leq n$ and a given bound k. Then the algorithm constructing a Boolean formula encoding step executions of L of length k is as follows:

1. To capture the requirement that each L_i is in the τ-closure of its initial states in $\mathbf{S}_1$ add for each L_i the literal $in(s, 1)$ for all the states S forming the τ-closure of I_i and the literal $\neg in(s', 1)$ for the rest of the states s'.
2. For all time steps $1 \leq t \leq k$, instantiate the following propositional schemes:
   (a) For all states $s \in S_1 \cup \cdots \cup S_n$, instantiate proposition scheme (5), the progress of control flow.
   (b) For all the components L_j, instantiate scheme (6) that encodes scheduling.
   (c) For all transitions with visible actions $l \in \Delta_1 \cup \cdots \cup \Delta_n$, instantiate scheme (7) encoding the execution of single transitions.
   (d) For each LTS L_j, limit the number of visible actions by instantiating scheme (8).
   (e) (Optionally) instantiate the scheme (9) ruling out idle steps.
   (f) For all visible actions ac, instantiate the scheme (10) defining when an action is enabled.
   (g) For all the visible actions require synchronization by instantiating scheme (11) .

Let $ST(L, k)$ be the (step) formula obtained by the translation algorithm. Given a satisfying truth valuation α for the formula $ST(L, k)$ call $\mathbf{S}_1 \xrightarrow{A_1} \cdots \xrightarrow{A_k} \mathbf{S}_{k+1}$ an α-execution, where $\mathbf{S}_i = \langle S_i^1, \ldots S_i^n \rangle$, each $S_i^j = \{s \mid s \in S_j$ and $\alpha(in(s, i)) = $ true$\}$ and $A_i = \{ac \in A_i \mid \alpha(ex(ac, i)) = $ true$\}$. The following theorem establishes that any α-execution is indeed a valid step execution.

Theorem 4. *If the formula $ST(L, k)$ has a satisfying truth valuation α, then the α-execution $\mathbf{S}_1 \xrightarrow{A_1} \cdots \xrightarrow{A_k} \mathbf{S}_{k+1}$ is a step execution.*

Proof. The proof is by induction over the time steps. The goal is to show that a valid step execution can be constructed from α by mapping the elements as stated above.

1. **Base Case.** The literal $in(s, 1)$ is true for the τ-closure of the initial states and false for the other states. This adheres to condition (1) of the translation algorithm. Thus, $\mathbf{S}_1$ is trivially correct.

2. **Induction Hypothesis.** Assume that the step execution is valid up to the time step i.

3. **Induction Step.** By induction hypothesis, α is such that for all $t \leq i, A_t = \{ac \mid \alpha(ex(ac, t)) = \text{true}\}$ correspond to executed actions and the result is a step execution. Consider $A_{i+1} = \{ac \mid \alpha(ex(ac, i+1)) = \text{true}\}$. By condition (2d) in the translation algorithm, the actions for which $ex(ac, i + 1)$ is true have to be limited to at most one in each component.

   Secondly, if $ex(ac, i + 1)$ is true, then by condition (2g) the literals $en(ac, L, i + 1)$ have to be true for all the components L having ac in their alphabet. Due to condition (2f), this can only be the case if for all such L, $in(s, i)$ is true for some state s having an outgoing transition labeled ac.

   Since at least one $in(s, i)$ has to be true, then the instances of the scheme (2c) have to enforce at least one $ex(l, i+1)$ from all the components L to be true for some l corresponding to a transition labeled ac. Indeed, the scheme enforces that for all such transitions.

   Thus, for the set of actions A_{i+1} it holds that:

   - At most one visible action can be executed from each component.
   - If action ac is executed, then every component L having ac in its alphabet has to participate by executing a non-empty set of transitions labeled ac.
   - If action ac is executed, then each component having it in its alphabet has to execute every enabled transition labeled ac.

   Thus, the actions taken in A_{i+1} are of the correct form. It remains to show that the progress of the control flow is correct. Condition (2b) mandates that the literal $sc(L, i+1)$ is true for components with executed actions and false for all the rest. Therefore, the proposition scheme (2a) reduces to

   $$in(s, i + 1) \leftrightarrow in(s, i) \tag{12}$$

   for components remaining idle and

   $$in(s, i + 1) \leftrightarrow \exists s_1 in(s_1, i + 1) \vee \\ \exists s_2(in(s_2, i) \wedge \exists l\, ex(l, i+1)) \tag{13}$$

   for components for which $sc(L, i)$ is true. By (12), the $in(s, i + 1)$ literals have to be true for the same states for which they are true in step i for the idle components. The latter disjunct on the right-hand side of the equivalence (13) requires the $in(s, i + 1)$ literal to be true for all the successor states of the executed transitions. The first existentially quantified disjunct mandates the same for the τ-closure of the immediate successor. By the argument above (no τ-loops), the literal can be true for only those states. Hence, the control flow is correctly updated, which completes the proof. $\square$

Theorem 5. *If* $\mathbf{S}_1 \overset{A_1}{\to} \cdots \overset{A_k}{\to} \mathbf{S}_{k+1}$ *is a step execution of* L, *it is an* α-*execution for some satisfying valuation* α *of* $ST(L, k)$.

Proof. The goal is to prove that given any step execution, a truth assignment α satisfying $ST(L, k)$ can be constructed from it by starting from the reached states and executed actions i.e., taking the step execution to be the α-execution.

- The conditions for the initial states is satisfied since $\mathbf{S}_1$ is in any step execution the τ-closure of the system's initial states.
- In subsequent steps, to satisfy scheme (2f), the $en(ac, L, i)$ literal has to be true whenever action ac is executed in A_i for all the components L such that ac is in their alphabet.
- Scheme (2g), on the other hand, defines the $en(ac, L, t)$ literal based on the reached states. This definition is not violated by the step execution and the scheme above since to be able to execute ac in A_i, each component has to be in a state where a transition labeled ac is possible.
- The schemes (2d) and (2e) are satisfied by definition of a step execution.
- By scheme (2c), the step execution requires the $ex(l, t)$ literal be true for every transition leaving a reached state when the action l it is labeled with is executed. For every other transition the literal is false. Thus, $ex(l, t)$ is true iff only transition l is executed.
- The instances of scheme (2b) are equivalences and the step execution mandates the $sc(L, t)$ literal to be true iff component L is scheduled, i.e., a visible action is executed from its alphabet.
- By the schemes analyzed above, the step execution has defined truth values for the literals $ex(l, t)$ and $sc(L, t)$ used in scheme (2a). It remains to be shown that these truth values are not in conflict with the control flow scheme. This is not the case since based on the step execution, the $sc(L, t)$ literal evaluates in such a way that the scheme reduces to

  $$in(s, t + 1) \leftrightarrow in(s, t)$$

  for idle components and in any step execution idle components remain in the same states. Furthermore, for non-idle components the scheme reduces to

  $$in(s, t + 1) \leftrightarrow \exists s_1 in(s_1, t + 1) \vee \\ \exists s_2(in(s_2, t) \wedge \exists l\, ex(l, t))$$

  The second disjunct mandates $in(s, t+1)$ literal to be true for successor states for transitions l where the $ex(l, t)$ literal is true. The first one extends this to their τ-closure. Thus, the scheme and the step execution agree on the reached state. $\square$

3.5 Process Executions

As can be seen from Definitions 6 and 8 the difference between step and process executions is rather simple. Indeed, the resulting formula needs only one additional propositional

scheme. Namely, if an action is executed at $t + 1$, then some participating component had to be scheduled in step t, i.e.

$$ex(ac, t + 1) \rightarrow \exists L\, sc(L, t) \qquad (14)$$

where

- ac varies over the visible actions of the system, and
- L quantifies over the components having ac in their alphabet.

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

(h) For all the visible actions $a \in \Sigma_1 \cup \cdots \cup \Sigma_n$ instantiate the scheme (14).

The following theorem establish the soundness and completeness of the augmented encoding with respect to process executions.

Theorem 6. *If the formula $PR(L, k)$ has a satisfying truth valuation α, the α-execution $\mathbf{S}_1 \overset{A_1}{\rightarrow} \cdots \overset{A_k}{\rightarrow} \mathbf{S}_{k+1}$ is a process execution.*

Proof. Every satisfying truth assignment α of $PR(L, k)$ satisfies also $ST(L, k)$. Thus, by Theorem 4 every truth valuation corresponds to a step execution. Therefore, it remains to be proven that with the additional scheme the models are limited to process executions. That is straightforward since the scheme (14) is a verbatim translation of the additional criterion given in Definition 8. □

Theorem 7. *If $\mathbf{S}_1 \overset{A_1}{\rightarrow} \cdots \overset{A_k}{\rightarrow} \mathbf{S}_{k+1}$ is a process execution of L, it is an α-execution for some satisfying valuation α of $PR(L, k)$.*

Proof. Similarly as in Theorem 6 above, process executions adhere to condition (h) of the augmented translation algorithm since the scheme is a verbatim translation of the process criterion. □

3.6 Reachability Properties

In this work, checking a reachability property equals answering the question whether an LTS system can reach a *global* state in which some global state predicate holds. It follows from Theorems 4 to 7 that the presented encodings for step and process executions preserve reachable global states. Any global state predicate can therefore be studied with the presented approach.

However, some care is needed when expressing a global state predicate when the non-trivial τ-components are eliminated as described in the beginning of Section 3. Given a preprocessed LTS system L, the reached determinized global state inferred from a model of $ST(L, k)$ and $PR(L, k)$ is a tuple of sets. Each set contains local states from a particular component of the preprocessed system. However, each local state of the preprocessed system represents a set of local states of the original system. For expressing a global state

predicate on the states of the original system, a new literal $fs(s, k + 1)$ is introduced for every local state s of the original system. These literals capture reached final states. The idea is to augment the formula $ST(L, k)$ (or $PR(L, k)$) with constraints on the new literal $fs(s, k + 1)$ so that then the global state predicates can be expressed in terms of the local states s. These constraints guarantee that a global state $\mathbf{s} = \langle s_1, \ldots, s_n \rangle$ of the original system is reached by a step (process) execution of length k iff there is a model of the augmented formula where $fs(s_i, k + 1)$ is true for exactly those s_i which form $\mathbf{s}$. The added constraints are as follows:

$$fs(s, k + 1) \rightarrow in(repr(s), k + 1) \qquad (15)$$
$$[1, 1]\{fs(s_1, k + 1), \ldots, fs(s_n, k + 1)\} \qquad (16)$$

where

- s varies over the local states of the original system,
- $repr(s)$ is the representative state of the τ-component the state s belongs to, and
- $s_1, \ldots, s_n$ are the local states of a *particular* component from the original system. Thus, an instance of the formula is needed for all the components.

The former constraint requires that the τ-component that s is an element of is reached. The latter, on the other hand, allows exactly one state from each component. Any global state predicate can now be evaluated in the global state encoded by the new $fs(s, k + 1)$ literals.

A deadlock state predicate which holds for exactly those global states from which no execution can be extended is a particularly interesting case among global state predicates. A deadlock can be captured as a disjunction of conjunctions where each conjunction is of the form $(fs(s_1, k + 1) \wedge \cdots \wedge fs(s_n, k + 1))$ such that $s_1, \ldots, s_n$ are local states, one from each component, and the global state that they form enables no transition. However, the number of such global states is potentially large and most of them are probably not even reachable.

A more compact approach is to use additional literals $en(ac, L_i, k + 1)$ and $en(ac, k + 1)$ evaluated based on the $fs(s, k+1)$ literals. The former captures the fact that action ac is enabled in component L_i in state $k + 1$. The latter encodes the condition that the action is globally enabled. Thereafter, the deadlock state predicate is completed by requiring that in the final state, no action is enabled. The formulas to encode the deadlock state predicate are then:

$$\forall s_1 \neg fs(s_1, k + 1) \qquad (17)$$
$$en(ac_1, L_1, k + 1) \leftrightarrow \exists s_2\, fs(s_2, k + 1) \qquad (18)$$
$$en(ac_2, k + 1) \leftrightarrow \forall L_2\, en(ac_2, L_2, k + 1) \qquad (19)$$
$$\forall ac_3 \neg en(ac_3, k + 1) \qquad (20)$$

where

- s_1 quantifies over the local states of the system having outgoing τ-transitions,
- ac_1 and ac_2 vary over the visible actions of the system,
- L_1 and L_2 refer to the components having ac, resp. ac_2 in their alphabet,

- s_2 quanti£es over the states in L_1 having an outgoing transition labeled ac, and
- ac_3 quanti£es over all the actions of the system.

The presented approach can be extended to reason about reachability of sequences of global states. Even though step and process executions may lose some of the intermediate states, sequences can be analyzed provided that all state changes of interest can be observed through the occurrences of visible actions. The 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 a syntactic safety subset of LTL_{-X}, the linear temporal logic LTL without the next-time operator X, a £nite automaton construction tool is available [14].

4 Test Results

4.1 Implementation

The approach has been implemented as a translator which maps a given synchronizing system of LTSs into a CNF formula in DIMACS format accepted by most propositional satis£ability (SAT) solvers. The translation is based on the encoding presented in Section 3 but it is optimized in the following way. First, the resulting formula is represented as a Boolean circuit (explained below). Second, the circuit is reduced by simpli£cation rules, substructure sharing and cone of in¤uence reduction [12]. The reduced circuit is transformed to CNF using a linear size translation introducing a new atom for each gate in the circuit [12]. The circuit reduction and mapping to CNF are done using a tool called BCZChaff [11], which is integrated with the ZChaff SAT solver so that ZChaff can be run directly on the generated CNF formula.

The Boolean circuit representation of an encoding formula is straightforward: for each atom and connective in the formula, a gate is introduced in the circuit. This is easy as the tool used supports directly the extended set of Boolean functions used in the translation including the cardinality constraints (see, e.g. (8)). For example, the subcircuit for an instance of scheme (8) is shown in Figure 4 (right hand side). The letter 'T' marks a constraint (true) for the gate $UV(L_1, t)$. So each instance of the propositional schemes in the encoding is represented by this kind of a simple subcircuit constrained to true.

However, this straightforward representation is optimized when translating equivalences such as (6), for which no gate for the atom on the left nor for the equivalence connective are introduced. Rather, the atom on the left is identi£ed with the gate for the main connective on the right hand side of the equivalence. For example, the subcircuit for an instance of the equivalence scheme (6) is shown in Figure 4 (left hand
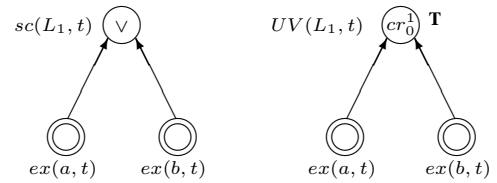

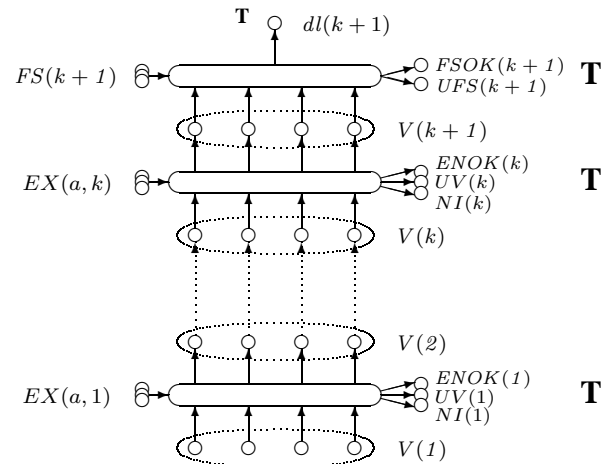**Fig. 4.** Elements Illustrating Encoding from Running Example

Fig. 5. Schematic Diagram of the Circuit

side). Notice that equivalence subcircuits are not constrained to true.

The complete process of translating an LTS system to a Boolean circuit results in a circuit whose schematic diagram can be seen in Figure 5. The circuit is a faithful representation of the encoding as a propositional formula because the satisfying truth valuations of the encoding formula and satisfying valuations of the circuit coincide. A satisfying valuation of the circuit is a truth value assignment for the input gates (gates with no incoming edges) of the circuit such that the resulting value of each constrained gate matches its speci£ed value (true).

4.2 Test Cases

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

- punroll [8], a BMC implementation translating 1-safe Petri nets to Boolean circuits applying process semantics,
- NuSMV (version 2.1.2) BMC [5], a bounded model checker for SMV speci£cations using interleaving semantics,
- NuSMV (version 2.1.2) BDD [4], a BDD-based symbolic model checker for SMV speci£cations, and
- SPIN (version 4.0.7) [10], an explicit-state model checker for PROMELA programs applying partial-order reduction methods.

A set of test cases has been adopted from [6] taking those cases known to deadlock. The examples are provided as LTS

(fsa), Promela and SMV speci£cations and therefore the comparison task is feasible. It should be noted that the Petri nets are created from scratch using the fsa speci£cations. 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,
– punroll k and punroll s, similarly for the encoding in [8] using 1-safe Petri Nets with process semantics,
– SMV k and SMV s, the bound and the time in seconds for NuSMV BMC [5],
– SMV bdd, running time for NuSMV BDD [4], and
– SPIN, the results for SPIN [10].

The tests were carried out using an AMD Athlon machine with a 1400 MHz CPU and 1 Gigabyte of memory running the Linux operating system. In the £rst three BMC methods (the presented encodings and the punroll tool), the running time is the sum of the time needed to create the Boolean circuit and the time needed to obtain a satisfying truth assignment for the bound given in the table using the SAT solver zChaff version 2001.2.17 [16].

For NuSMV BMC, the running time is the execution time of NuSMV using the solver above. If the number is of the form $> n$, no deadlock could be found and n refers to the greatest number of execution steps for which the unsatis£ability could be established within a time limit of one hour.

For NuSMV BDD, the running time is the sum of the time needed to solve the problem and to create a counterexample. For SPIN, the running time is the execution time of the veri£er compiled from the generated C-code. The time needed to create the veri£er and to compile it are omitted. The search is limited to invalid end states and in the attempt to £nd short deadlocks, breadth £rst search is used (using the switch `-DBFS` in the compilation process).

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. Similarly, SPIN ran out of memory with the examples Dartes and DP(12).

Experiments indicate that with these examples it sometimes takes zChaff far longer to prove a formula unsatis£able than to £nd 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 unsatis£able instance modeling process executions of length 29. The test cases and the tool translating LTSs to Boolean circuits are available for download at [13].

Even though the test cases contain relatively little 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. With these examples, the results are comparable to those using punroll. However, one can

construct an in£nite family of LTS systems for which the presented encoding grows polynomially w.r.t. the size of the LTS system whereas for the encoding using Petri nets the growth is exponential. Compared to the BDD-based model checking and SPIN, the results reiterate the fact that BMC is at its best in £nding short deadlocks.

The test results suggest the following guidelines. The veri£er should start the task by using an explicit-state model checker like SPIN. If the state space and the memory requirements grow too large, the symbolic methods should be applied. If completeness is not required, BMC procedures can be used to detect short counterexamples. The presented approach is at its best when the system model contains a lot of non-determinism.

5 Conclusions and Related Work

The paper studies bounded model checking of reachability properties of an asynchronous system represented as 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 propositional formulas. In the £rst case, the resulting formula encodes precisely the step executions of the system model and in the second the process executions. The encoding is compact leading to a formula linear in the size of the system description and 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 a component L_j, respectively. The encoding uses Boolean functions outside traditional propositional logic namely cardinality constraints. However, for the presented instances the bound holds were the use of them disallowed. The used functions with i arguments can namely be simulated using $\mathcal{O}(i)$ new $\vee, \wedge$ and $\neg$ operations and variables. The approach is backed by a set of test cases showing that the run 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. Apart from the different modeling formalism the main difference to the current work is that Petri nets do not have a component structure, which could be exploited in BMC encodings. In fact, the on-the-¤y determinization construction presented in this work can not be ef£ciently employed in the Petri net context. Thus, the bene£ts of determinization, including the more ef£cient handling of τ transitions, is a new technique to this work. The paper [8] considers some of the same examples presented here. However, a direct comparison using the Petri nets from [8] is omitted since the nets there do

Table 2. Test Results (BCZChaff)

Problem	St. k	St. s	Pr. k	Pr.s	punroll k	punroll s	SMV k	SMV s	SMV BDD s	SPIN
Dartes	31	1.2	31	0.22	32	0.09	N/A	N/A	N/A	N/A
DP(12)	1	0.008	1	0.02	1	0.004	> 8	830	0.12	N/A
Elev(3)	7	0.40	7	0.13	8	0.15	14	410	0.64	0.03
Elev(4)	9	1.8	9	0.40	10	2.2	17	120	2.7	0.04
Key(2)	35	370	35	230	36	230	> 30	2100	0.10	0.03
Key(3)	36	1100	36	$> 1h$	37	1900	> 21	2700	0.27	0.39
Key(4)	37	90	37	510	38	130	> 19	3200	0.73	5.79
Key(5)	38	$> 1h$	38	10	39	180	> 18	1900	3.2	75
Mmgt(3)	7	0.16	7	0.10	7	0.09	10	14	0.13	0.11
Mmgt(4)	8	1.5	8	0.21	8	0.42	12	73	0.25	1.0
Q(1)	9	0.10	9	0.09	9	0.06	> 11	1500	2.0	36
Hart(75)	150	5.6	150	3.3	150	5.5	151	990	1.9	0.03
Hart(100)	200	11	200	6.4	200	14	201	4800	5.5	0.04
Sentest(75)	83	2.1	83	2.4	83	3.1	88	220	1.5	0.06
Sentest(100)	108	3.6	108	4.1	108	6.0	113	980	4.6	0.07
Dac(15)	2	0.004	2	0.008	3	0.004	3	0.27	0.11	1.0
Speed(1)	4	0.014	4	0.008	4	0.008	7	0.13	0.07	0.20

not correspond to the fsa speci£cations. The cause of this inconsistency could be traced to the fsa to 1-safe Petri net conversion performed in [15]. Instead, the Petri nets used in the presented comparison are created from scratch using the fsa speci£cations.

So far, only the veri£cation of reachability properties has been considered whereas model checking of more complex temporal properties is left for future work. Some initial work in that direction can be found in [9], where *LTL* BMC is presented for Petri nets using a logic programming approach.

Acknowledgement.

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 *LNCS*, 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 *LNCS*, pages 60–71. Springer, November 2000.

3. P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in Alpha microprocessor using satis£ability solvers. In *Proceedings of the 13th International Conference of Computer Aided Veri£cation(CAV'2001)*, volume 2102 of *LNCS*, 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 Veri£cation (CAV'2002)*, volume 2404 of *LNCS*, 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)*, 3(4&5):519–550, 2003.

10. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

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

12. T. A. Junttila and I. Niemelä. Towards an ef£cient tableau method for boolean circuit satis£ability testing. In *Computational Logic - CL 2000; First Internatinal Conference*, volume 1861 of *LNAI*, pages 553–567, London, UK, July 2000. Springer, Berlin.

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

14. T. Latvala. Ef£cient model checking of safety properties. In *Model Checking Software. 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 74–88. Springer, 2003.

15. S. Melzer and S. Römer. Deadlock checking using net unfoldings. In *Proceeding of 9th International Conference on Computer Aided Veri£cation (CAV'97)*, volume 1254 of *LNCS*, pages 352–363. Springer, June 1997.

16. M. Moskewicz, Y. Madigan, L. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an ef£cient SAT solver. In *39th Design Automation Conference*, July 2001.

17. C. Papadimitriou and H. Lewis. *Elements of the Theory of Computation*. Prentice Hall, 1981.