

Emptiness Check of Powerset Büchi Automata using Inclusion Tests

Souheib Baarir Alexandre Duret-Lutz
LIP6-CNRS,
Université Pierre et Marie Curie (Paris 6),
France

E-mail: Souheib.Baarir@lip6.fr, Alexandre.Duret-Lutz@lip6.fr

Abstract

We introduce two emptiness checks for Büchi automata whose states represent sets that may include each other. The first is equivalent to a traditional emptiness check but uses inclusion tests to direct the on-the-fly construction of the automaton. The second is impressively faster but may return false negatives. We illustrate and benchmark the improvement on a symmetry-based reduction.

1. Introduction

The automata-theoretic approach to model-checking [15] uses automata on infinite words to represent a system as well as a property to check on it. Both automata are synchronized, and the resulting *product automaton* is examined by an *emptiness check*.

One drawback of this approach, known as *the state explosion problem*, lies in the large size of the automaton that represents the behavior of the system, and hence in the resulting size of the product automaton that has to be checked for emptiness. Several techniques have been proposed to reduce the size of both automata [14]. Some of these, like unfolding graphs [4], observation graphs [9], or symmetry-based reductions [8], share a common property: instead of an automaton \mathcal{A} , they build an automaton \mathcal{B} whose states are a symbolic representation of a set of states of \mathcal{A} .

To take an example, the method of partial symmetries introduced by Haddad et al. [8] works by partitioning the set of successors s_1, s_2, \dots, s_n of a state of the original automaton, and using these partitions as the states S_1, S_2, \dots, S_m of the reduced automaton \mathcal{B} . However, because this might be done locally in each state, there is no guarantee that a state s of the original automaton may be represented by only one state S in \mathcal{B} . This reduced automaton can then be checked for emptiness (using any mainstream emptiness check algorithm [11, 5, 7]) with the same result as if the original automaton \mathcal{A} had been checked (of

course the idea is that \mathcal{B} is constructed directly from the system and the property, in such a way that we avoid the construction of \mathcal{A}).

Unfortunately, these methods may well construct a “reduced” automaton that has more states than the original automaton! Indeed, if we call \mathcal{A} the original automaton and $Q_{\mathcal{A}}$ its set of states, this technique constructs an automaton \mathcal{B} where $Q_{\mathcal{B}} \subseteq 2^{Q_{\mathcal{A}}}$ and may have at worst $2^{|Q_{\mathcal{A}}|}$ states. However in many practical cases we still have $|Q_{\mathcal{B}}| < |Q_{\mathcal{A}}|$. For instance, Fig. 1 (ignore f and g for the moment) shows two automata \mathcal{A} and \mathcal{B} , where \mathcal{B} ’s states are sets of states of \mathcal{A} . In the sequel, we will always use lowercase letters like s_i to denote states of the original automaton, and uppercase letters like S_i to denote states of the reduced automaton (which are sets of states).

Furthermore, we can see that some states of \mathcal{A} may appear in several states of \mathcal{B} , and most importantly a state of \mathcal{B} may even be a subset of another state of \mathcal{B} (e.g., $S_4 \subseteq S_1$ and $S_5 \subseteq S_2$). These inclusions motivate this paper.

When the state space is constructed on-the-fly during the emptiness check, we will show that we can perform *inclusion tests* to reduce the number of constructed states. For instance, in Fig. 1 we can avoid constructing S_4 and S_5 because the information they represent is already included in S_1 and S_2 .

To rid ourselves of the construction technique used to build \mathcal{B} , we present our emptiness check algorithms in a general framework of *powerset automata*.

Section 2 defines these automata formally, and proposes a set of 5 properties tying \mathcal{A} to \mathcal{B} that are sufficient to ensure that both automata are equivalent with respect to their emptiness. Section 3 presents our *emptiness check* algorithm for such automata. Basically such an algorithm may answer “empty” or “not empty”. Section 4 shows a small modification that leads to a faster algorithm that can answer “empty” or “I don’t know”. The definitions and algorithms presented in the aforementioned sections are abstract in the sense that they do not presume how \mathcal{B} was constructed from \mathcal{A} : as an illustration, Section 5 adapts the

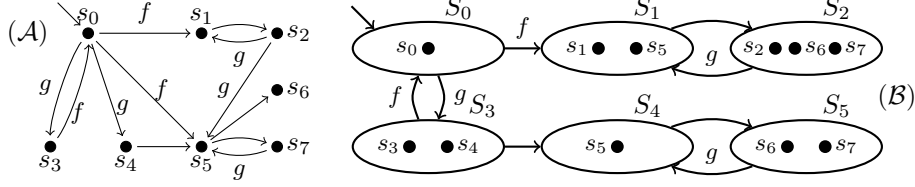


Figure 1. An example of powerset automaton \mathcal{B} for an automaton \mathcal{A} , with $\mathcal{F}_A = \mathcal{F}_B = \{f, g\}$.

technique of Haddad et al. [8] to show how to construct a \mathcal{B} using (partial-)symmetries, and we prove that this construction satisfies the requirements of our algorithm. We also benchmark this construction and show that although theoretically the algorithm may still explore $2^{|\mathcal{Q}_A|}$ states in the worst case, in practice it improves the state space size by a good factor.

2. Definitions

We start with the definition of the automata we manipulate. We use structures that look like Generalized Büchi Automata, but without atomic propositions and with acceptance conditions on transitions. In the automata-theoretic approach, atomic propositions are only used for the computation of the synchronized product and can be ignored after this operation: the emptiness check algorithm does not need them. Putting acceptance conditions on transitions rather than on states is motivated by the fact that it is more generic: state-based acceptance conditions can be converted to transition-based acceptance conditions without adding states or transitions, while the converse is not true.

Definition 1 (UTGBA). An Unlabelled Transition-based Generalized Büchi Automaton (UTGBA) is a Büchi automaton without any atomic propositions, but with generalized acceptance conditions on transitions. It is a tuple $\mathcal{A} = \langle \mathcal{Q}_A, \mathcal{Q}_A^0, \mathcal{F}_A, \Delta_A \rangle$ where

- \mathcal{Q}_A is a finite set of elements called states,
- $\mathcal{Q}_A^0 \subseteq \mathcal{Q}_A$ is a set of initial states,
- \mathcal{F}_A is a finite set of elements called acceptance conditions,
- $\Delta_A \subseteq \mathcal{Q}_A \times 2^{\mathcal{F}_A} \times \mathcal{Q}_A$ is the transition relation, where each transition carries a (possibly empty) set of acceptance conditions of \mathcal{F}_A .

Definition 2 (Reachable states). Let $\mathcal{A} = \langle \mathcal{Q}_A, \mathcal{Q}_A^0, \mathcal{F}_A, \Delta_A \rangle$ be a UTGBA. A state s is reachable if $s \in \mathcal{Q}_A^0$ or if there exists a finite sequence $\langle s_0, F_0, s_1 \rangle \langle s_1, F_1, s_2 \rangle \cdots \langle s_{n-1}, F_{n-1}, s_n \rangle$ of transitions of Δ_A , starting at an initial state $s_0 \in \mathcal{Q}_A^0$, and ending on state $s_n = s$. We denote $\text{Reach}(\mathcal{A})$ the set of all reachable states of \mathcal{A} .

Definition 3 (Run & accepting run). Let \mathcal{A} be a UTGBA as above. A run of \mathcal{A} is an infinite sequence $\langle s_0, F_0, s_1 \rangle \langle s_1, F_1, s_2 \rangle \cdots$ of transitions of Δ_A , starting at an initial state $s_0 \in \mathcal{Q}_A^0$. A run is accepting if $\forall f \in \mathcal{F}_A, \forall i \geq 0, \exists j \geq i$, such that $f \in F_j$, i.e., if its transitions are labelled by each acceptance condition infinitely often.

We write $\text{Run}(\mathcal{A})$ and $\text{Acc}(\mathcal{A})$ the set of all runs and the set of all accepting runs of \mathcal{A} .

For $\sigma = \sigma(0)\sigma(1)\sigma(2)\cdots \in \text{Run}(\mathcal{A})$, we denote $\sigma_{\text{in}}(i)$, $\sigma_{\text{acc}}(i)$, and $\sigma_{\text{out}}(i)$ the source, the acceptance condition, and the destination of the i^{th} transition of σ , in other words $\sigma(i) = \langle \sigma_{\text{in}}(i), \sigma_{\text{acc}}(i), \sigma_{\text{out}}(i) \rangle$. Finally we denote σ^i the suffix of σ starting after the i^{th} transition, that is: $\sigma^i = \sigma(i)\sigma(i+1)\sigma(i+2)\cdots$.

As an example, consider \mathcal{B} in Fig. 1. The acceptance conditions are f, g , and thus \mathcal{B} has only one accepting run: $\text{Acc}(\mathcal{B}) = \{S_0 S_3 S_0 S_3 S_0 S_3 \dots\}$.

An emptiness check tells whether Acc is empty, and here we are interested in an equivalence relation between automata that is solely based on the result of this operation.

Definition 4 (Emptiness-equivalence). Two UTGBAs, \mathcal{A} and \mathcal{B} , are \emptyset -equivalent iff either both automata have an accepting run, or none have.

$$\mathcal{A} \stackrel{\emptyset}{\equiv} \mathcal{B} \quad \text{iff} \quad \text{Acc}(\mathcal{A}) = \emptyset \iff \text{Acc}(\mathcal{B}) = \emptyset$$

Now we propose a set of 5 properties that link two UTGBA \mathcal{A} and \mathcal{B} such that the states of \mathcal{B} are sets of states of \mathcal{A} , and \mathcal{B} is \emptyset -equivalent to \mathcal{A} . The idea is that if we know a method to construct a \mathcal{B} that verifies these sufficient conditions, we can run the emptiness check on \mathcal{B} and avoid constructing \mathcal{A} . These properties hold in Fig. 1.

Definition 5 (\wp -UTGBA). Let $\mathcal{A} = \langle \mathcal{Q}_A, \mathcal{Q}_A^0, \mathcal{F}_A, \Delta_A \rangle$ and $\mathcal{B} = \langle \mathcal{Q}_B, \mathcal{Q}_B^0, \mathcal{F}_B, \Delta_B \rangle$ be two UTGBAs. \mathcal{B} is a \wp -UTGBA (powerset UTGBA) over \mathcal{A} if it satisfies the following properties:

$$\mathcal{Q}_B \subseteq 2^{\mathcal{Q}_A} \setminus \{\emptyset\} \quad (1)$$

$$\mathcal{F}_B = \mathcal{F}_A \quad (2)$$

$$\bigcup_{S \in \mathcal{Q}_B^0} S = \mathcal{Q}_A^0 \quad (3)$$

$$\begin{aligned} \forall \langle s, F, s' \rangle \in \Delta_A, \forall S \in \text{Reach}(\mathcal{B}), \\ s \in S \implies \exists s' \in \mathcal{Q}_B \text{ s.t. } s' \in S', \text{ and } \langle S, F, S' \rangle \in \Delta_B \end{aligned} \quad (4)$$

$$\forall \langle S, F, S' \rangle \in \Delta_B, \forall s' \in S', \exists s \in S, \text{ s.t. } \langle s, F, s' \rangle \in \Delta_A \quad (5)$$

Proposition 1. Let $\mathcal{A} = \langle \mathcal{Q}_A, \mathcal{Q}_A^0, \mathcal{F}_A, \Delta_A \rangle$ and $\mathcal{B} = \langle \mathcal{Q}_B, \mathcal{Q}_B^0, \mathcal{F}_B, \Delta_B \rangle$ be two UTGBAs such that \mathcal{B} is a \wp -UTGBA over \mathcal{A} . Then $\mathcal{A} \stackrel{\emptyset}{=} \mathcal{B}$.

Proof. We want to show that $\exists \sigma \in \text{Acc}(\mathcal{A}) \iff \exists \sigma' \in \text{Acc}(\mathcal{B})$.

(\implies) Let $\sigma = \langle s_0, F_0, s_1 \rangle \langle s_1, F_1, s_2 \rangle \cdots \in \text{Acc}(\mathcal{A})$. Since $s_0 \in \mathcal{Q}_A^0$ we can use (3) and find an $S_0 \in \mathcal{Q}_B^0$ such that $s_0 \in S_0$. Since S_0 is reachable in \mathcal{B} and contains s_0 , we can use (4) to find an $S_1 \in \mathcal{Q}_B$ such that $s_1 \in S_1$ and $\langle S_0, F_0, S_1 \rangle \in \Delta_B$. Likewise, because S_1 is reachable in \mathcal{B} and contains s_1 by construction, we can use (4) again to find an $S_2 \in \mathcal{Q}_B$ such that $s_2 \in S_2$ and $\langle S_1, F_1, S_2 \rangle \in \Delta_B$. Iterating (4) we can construct a sequence $\sigma' = \langle S_0, F_0, S_1 \rangle \langle S_1, F_1, S_2 \rangle \cdots \in \text{Run}(\mathcal{B})$ such that $s_i \in S_i$ for all i . Since $\mathcal{F}_B = \mathcal{F}_A$ (2) and σ' visits each acceptance condition as often as σ , $\sigma' \in \text{Acc}(\mathcal{B})$.

(\impliedby) Let $\sigma' = \langle S_0, F_0, S_1 \rangle \langle S_1, F_1, S_2 \rangle \cdots \in \text{Acc}(\mathcal{B})$. Let's build a tree whose nodes (except the root) are states of \mathcal{A} . Let's call \perp the root of the tree at depth 0. The nodes of depth $n > 0$ are exactly the states in S_{n-1} . The father s of any node s' at depth $n > 1$ is chosen among the nodes of depth $n - 1$ such that $\langle s, F_{n-1}, s' \rangle \in \Delta_A$; (5) guarantees that such a node s exists. The father of any node at depth 1 is \perp . All edges of this tree, except those leaving the root node, correspond to transitions of Δ_A .

The set of nodes at depth $n > 0$ is a subset of \mathcal{Q}_A , which is finite, so although this tree is infinite it has a finite degree. By König's lemma it contains an infinite branch. The sequence constructed by following the edges of this infinite branch and ignoring the first edge (leaving \perp) $\langle s_0, F_0, s_1 \rangle \langle s_1, F_1, s_2 \rangle \cdots$ is an accepting run of \mathcal{A} . Indeed it is a run of \mathcal{A} ($s_0 \in \mathcal{Q}_A^0$) that visits each acceptance condition as often as σ' . \square

We now develop two propositions that introduce the emptiness check algorithm. (For brevity, we omit the proofs of these propositions and the following, and refer the interested reader to our technical report [1].) Both propositions use the following notation.

Definition 6 (Substitution of initial states). Let $\mathcal{A} = \langle \mathcal{Q}, \mathcal{Q}^0, \mathcal{F}, \Delta \rangle$ be a UTGBA, and $T \subseteq \mathcal{Q}$ a set of states of \mathcal{A} . We denote $\mathcal{A}[T]$ the automaton sharing the same structure as \mathcal{A} but using the set T as initial states. In other words $\mathcal{A}[T] = \langle \mathcal{Q}, T, \mathcal{F}, \Delta \rangle$.

The next proposition can be observed on Fig. 1: since no run that traverses state S_1 is accepting, then neither are the runs that traverse state S_4 because $S_4 \subseteq S_1$.

Proposition 2. Let $\mathcal{B} = \langle \mathcal{Q}_B, \mathcal{Q}_B^0, \mathcal{F}, \Delta_B \rangle$ be a \wp -UTGBA over $\mathcal{A} = \langle \mathcal{Q}_A, \mathcal{Q}_A^0, \mathcal{F}, \Delta_A \rangle$ and consider two states T and D of \mathcal{Q}_B such that $D \subseteq T$. We have

$$\text{Acc}(\mathcal{B}[\{T\}]) = \emptyset \implies \text{Acc}(\mathcal{B}[\{D\}]) = \emptyset$$

The following proposition allows us to split a transition $\langle R, F, T \rangle$ into a set of transitions $\langle R, F, T_1 \rangle, \dots, \langle R, F, T_n \rangle$ with $T_1 \cup \dots \cup T_n = T$, while preserving \emptyset -equivalence. Doing so might require adding new states and transitions to the automaton. Basically we want to substitute T by an automaton \mathcal{C} that has T_1, \dots, T_n as initial states, and that is \emptyset -equivalent to $\mathcal{A}[T]$. Fig. 2 illustrates this proposition. It will prove useful to apply such a decomposition if some T_i states have already been visited.

Proposition 3 (Decomposition of a transition in a \wp -UTGBA). Let $\mathcal{B} = \langle \mathcal{Q}_B, \mathcal{Q}_B^0, \mathcal{F}, \Delta_B \rangle$ be a \wp -UTGBA over $\mathcal{A} = \langle \mathcal{Q}_A, \mathcal{Q}_A^0, \mathcal{F}, \Delta_A \rangle$. Consider a transition $\langle R, F, T \rangle \in \Delta_B$ and let $\mathcal{C} = \langle \mathcal{Q}_C, \mathcal{Q}_C^0, \mathcal{F}, \Delta_C \rangle$ be a \wp -UTGBA over $\mathcal{A}[T]$. The automaton $\mathcal{B}' = \langle \mathcal{Q}_B \cup \mathcal{Q}_C, \mathcal{Q}_B^0, \mathcal{F}, \Delta_{B'} \rangle$ where

$$\mathcal{Q}_{B'}^0 = \begin{cases} (\mathcal{Q}_B^0 \setminus \{T\}) \cup \mathcal{Q}_C^0 & \text{if } T \in \mathcal{Q}_B^0 \\ \mathcal{Q}_B^0 & \text{otherwise} \end{cases}$$

$$\Delta_{B'} = (\Delta_B \setminus \{\langle R, F, T \rangle\}) \cup \{\langle R, F, T' \rangle \mid T' \in \mathcal{Q}_C^0\} \cup \Delta_C$$

is a \wp -UTGBA over \mathcal{A} .

3. Emptiness Check of \wp -UTGBA

A generalized Büchi automaton accepts a run (i.e., is nonempty) if it contains a reachable cycle in which all acceptance conditions appear. The emptiness check algorithms that have been devised to check this condition can be distinguished into two classes [5]:

- Nested Depth-First Searches (NDFS) algorithms will use a first DFS to find a state from which several DFS will be nested to find cycles around this state that visit each acceptance condition [12].
- Strongly-Connected-Component-based (SCC) algorithms will decompose the automaton into SCC and look for a reachable SCC in which all acceptance conditions appear.

NDFS algorithms have long been favored to SCC-based algorithms because of their better memory footprint (a couple of extra bits per state versus an integer). However it has

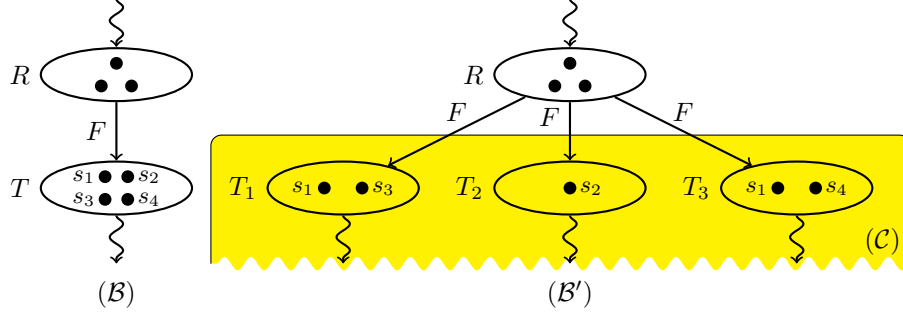


Figure 2. Example of decomposition of a transition $\langle R, F, T \rangle$ using proposition 3.

been pointed out that these extra data are negligible compared to what it takes to represent a state of a system (often hundreds of bytes) [11]. Furthermore, SCC-based algorithms have several advantages over NDFS: they need to explore less states to detect non-empty automata [7, 5], and they will deal with generalized acceptance conditions for free [5]. The algorithm we present here is derived from that of Couvreur [3], which is SCC-based.

3.1. Original algorithm

The idea is to enumerate all the maximal strongly connected components (MSCC) of an automaton. Any graph contains at least one MSCC without an outgoing arc, so to list all MSCCs, we should find such a terminal MSCC, remove it from the graph, and list all MSCCs of the resulting graph. To do so, the algorithm performs a depth-first search (DFS) of the automaton. While doing so, it maintains a stack of SCCs traversed by the DFS stack. As new transitions are visited, the SCC stack may be augmented or compacted. When an SCC is popped off the stack, meaning it is terminal, we check whether it is accepting: if that is the case, the algorithm terminates, otherwise all the states of this component are marked as “removed” so that whenever the DFS hits one of them again it can ignore it.

3.2. Adaptation to \wp -UTGBA

The new algorithm differs from the original in two points. First, the check for removed states is generalized: any removed state D can indeed be ignored by the DFS, but so can any state $T \subseteq D$! This is thanks to proposition 2.

Fig. 3 illustrates the second difference. Consider automaton \mathcal{B}_1 where the DFS is examining the transition $\langle R, F, T \rangle$ going to a new state T . Notice that there exists a state D in the search stack (or more generally in any SCC on the search stack) such that $D \subseteq T$. From the point of view of the underlying automaton it means some states in R can reach those in D and vice-versa, so they all belong to

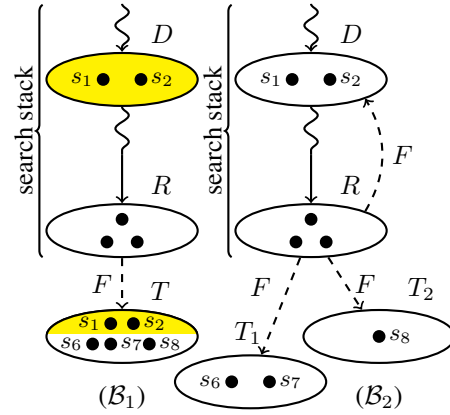


Figure 3. Inclusion checks in the search stack. We rewrite \mathcal{B}_1 to \mathcal{B}_2 .

the same SCC. For the emptiness check it would be beneficial to split the transition $\langle R, F, T \rangle$ as in \mathcal{B}_2 : it explicits the loop on the SCC and reuses previously seen states. Such a decomposition is correct thanks to proposition 3, but it has additional constraints we now formalize.

Let $\mathcal{B} = \langle Q_{\mathcal{B}}, Q_{\mathcal{B}}^0, \mathcal{F}, \Delta_{\mathcal{B}} \rangle$ be a \wp -UTGBA over some \mathcal{A} . $\text{Decomp}(\mathcal{B}, \langle S, F, T \rangle, D)$ is an operation that should perform a decomposition like in proposition 3. Besides \mathcal{B} and $\langle S, F, T \rangle$, which have the same purpose as in the definition, the argument D is a state of \mathcal{B} that is also a subset of T . Decomp should build the required automaton \mathcal{C} with two additional constraints:

- we want $D \in Q_{\mathcal{C}}^0$ (and the other states of $Q_{\mathcal{C}}^0$ will by definition complete T),
- and $\Delta_{\mathcal{C}}$ should not add transitions to the states of \mathcal{B} , i.e., $\{\langle S, F, S' \rangle \in \Delta_{\mathcal{C}} \mid S \in \text{Reach}(\mathcal{B})\} \subseteq \Delta_{\mathcal{B}}$.

Decomp returns a pair $\mathcal{B}', Q_{\mathcal{C}}^0$: the new automaton, and the initial states of \mathcal{C} . (Since in practice we build \mathcal{B} on the fly, as needed by the emptiness check, what really matters is that Decomp doesn't add any transition to the part of \mathcal{B}

already seen by the emptiness check, so it can continue from the result of Decomp as if it had started from it.)

The complete algorithm presented in Fig. 4 requires three operations on the structure of the states: tests for equality and inclusion, and the decomposition. We denote them respectively, =, \subseteq and Decomp. These operations are the only steps of the algorithm that have to be tailored to the encoding of the states.

3.3. Correctness of this algorithm

We only sketch enough of the actual proof [1] to be able to understand the algorithm. At any point of the execution we denote the contents of the *todo* and *SCC* stacks by:

$$\begin{aligned} todo &= \langle state_0, succ_0 \rangle \langle state_1, succ_1 \rangle \cdots \langle state_m, succ_m \rangle \\ SCC &= \langle root_0, la_0, acc_0, rem_0 \rangle \cdots \langle root_n, la_n, acc_n, rem_n \rangle \end{aligned}$$

todo is a DFS stack of pairs $\langle state, succ \rangle$ where *succ* is the set of outgoing transitions of *state* that haven't been considered yet. We call $state_0 \dots state_m$ the search path.

Each tuple in *SCC* represents a strongly connected component traversed by the search path. $root_i$ is the number of the first state of the component visited by the algorithm, and together with *H* (a map that numbers each visited state) it allows to define the set \mathfrak{S}_i of states belonging to the i^{th} SCC as follows:

$$\begin{aligned} \mathfrak{S}_i &= \{s \in Q_B \mid root_i \leq H[s] < root_{i+1}\} \text{ for } 0 \leq i < n \\ \mathfrak{S}_n &= \{s \in Q_B \mid root_n \leq H[s]\} \end{aligned}$$

acc_i is the set of acceptance conditions traversed by transitions between states of \mathfrak{S}_i . la_i are the acceptance conditions on the transition between the $(i-1)^{\text{th}}$ and the i^{th} components. The resulting chain of SCC is depicted by Fig. 5. Finally rem_i is a set of states to be removed when the component is popped, as we will see later.

The states of the automaton \mathcal{B} being checked are partitioned into three sets:

- The *active* states are those which are keys of *H* and have a nonzero value,
- the *removed* states are those which are keys of *H* and have a value of 0,
- the *unexplored* states are those that are not keys of *H*.

Initially, all states are *unexplored*. The function “DFSpush” is the only place a state can switch from *unexplored* to *active*, and the function “DFSpop” is the only place where it can switch from *active* to *removed*.

The following invariants are preserved at every line of “main” [1]:

Proposition 4. $m \geq n$ (in the above notation for *todo* and *SCC*) and there exists a strictly increasing function f such that $\forall i \leq n, root_i = H[state_{f(i)}]$. In

```

1 // Let  $\mathcal{B} = \langle Q_B, Q_B^0, \mathcal{F}, \Delta_B \rangle$  be the input automaton to check.
2 todo: stack of  $\langle state \in Q_B, succ \subseteq \Delta_B \rangle$ 
3 SCC: stack of  $\langle root \in \mathbb{N}, la \subseteq \mathcal{F}, acc \subseteq \mathcal{F}, rem \subseteq Q_B \rangle$ 
4 H: map of  $Q_B \mapsto \mathbb{N}$ 
5 max  $\leftarrow 0$ 
6
7 main():
8   forall  $S^0 \in Q_B^0$ 
9     DFSpush( $\emptyset, S^0$ )
10    while  $\neg todo.empty()$ 
11      if  $todo.top().succ = \emptyset$ 
12        DFSpop()
13      else
14        pick one  $\langle R, F, T \rangle$  off  $todo.top().succ$ 
15        if  $\exists D \in H.keys()$  such that  $(T \subseteq D) \wedge H[D] = 0$ 
16          continue
17        elseif  $T \not\subseteq H$ 
18          if  $\exists D \in H.keys()$  such that  $D \subseteq T \wedge H[D] > 0$ 
19             $\mathcal{B}, Q_C^0 \leftarrow Decomp(\mathcal{B}, \langle R, F, T \rangle, D)$ 
20             $todo.top().succ \leftarrow todo.top().succ \cup$ 
21               $\{\langle R, F, D \rangle\} \cup \{\langle R, F, T' \rangle \mid T' \in Q_C^0\}$ 
22          else
23            DFSpush( $F, T$ )
24          elseif  $H[T] > 0$ 
25            if  $merge(F, H[T]) = \mathcal{F}$ 
26              return  $\perp$ 
27          return  $\top$ 
28 DFSpush( $F \subseteq \mathcal{F}, S \in \mathcal{Q}$ ):
29   max  $\leftarrow max + 1$ 
30    $H[S] \leftarrow max$ 
31   SCC.push( $\langle max, F, \emptyset, \emptyset \rangle$ )
32   todo.push( $\langle S, \{\langle R, F, T \rangle \in \Delta_B \mid R = S\} \rangle$ )
33
34 DFSpop():
35    $\langle S, \_ \rangle \leftarrow todo.pop()$ 
36   SCC.top().rem.insert( $S$ )
37   if  $H[S] = SCC.top().root$ 
38     forall  $R \in SCC.top().rem$ 
39        $H[R] \leftarrow 0$ 
40     SCC.pop()
41
42 merge( $F \subseteq \mathcal{F}, n \in \mathbb{N}$ ):
43    $r \leftarrow \emptyset$ 
44   while  $(n < SCC.top().root)$ 
45      $F \leftarrow (F \cup SCC.top().acc \cup SCC.top().la)$ 
46      $r \leftarrow r \cup SCC.top().rem$ 
47     SCC.pop()
48   SCC.top().acc  $\leftarrow SCC.top().acc \cup F$ 
49   SCC.top().rem  $\leftarrow SCC.top().rem \cup r$ 
50   return SCC.top().acc

```

Figure 4. Emptiness check of a \wp -UTGBA.

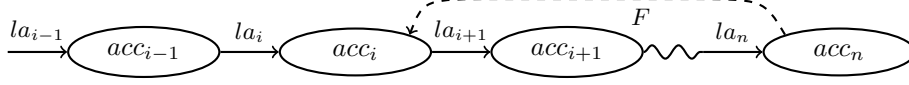


Figure 5. The meaning of la and acc in SCC.

other words, $root_0, root_1, \dots, root_n$ is a subsequence of $H[state_0], H[state_1], \dots, H[state_m]$. (I.e., the roots of the strongly connected components are on the search path of the depth-first search, and in the same order.)

Proposition 5. For any $i \leq n$ the subgraph induced by the states of \mathfrak{S}_i is a SCC. Furthermore there exists a cycle in this SCC that visits all the acceptance conditions of acc_i . Finally $\mathfrak{S}_0, \mathfrak{S}_1, \dots, \mathfrak{S}_n$ is a partition of the set of active states.

Proposition 6. $\forall i < n, \exists s \in \mathfrak{S}_i, \langle s, la_{i+1}, state_{f(i+1)} \rangle \in \Delta_B$.

Proposition 7. For any $i < n$, rem_i holds all states of \mathfrak{S}_i not on the search path.

Proposition 8. For any removed state q , $Acc(\mathcal{B}[\{q\}]) = \emptyset$.

The first two propositions guarantee that if the algorithm finds an i such that $acc_i = \mathcal{F}$, it corresponds to a reachable (prop. 4) accepting component (prop. 5). The last proposition justifies that no accepting run exists if the algorithm has removed all the states.

3.4. Counterexamples

When verifying a model by the automata theoretic approach, the presence of an accepting run means that there exists an execution of the modeled system that invalidates the property being checked, i.e., a counterexample. Therefore whenever the emptiness check exits with \perp , meaning the automaton has an accepting run, the user usually wants to see such a run to debug the model (or the property).

To produce a genuine accepting run from the stack of SCCs, these SCCs have to be searched again. The method described by Couvreur et al. [5] could be used only with the guarantee it would visit the same states that the emptiness check visited.

In our case, doing so seems hard because the computation of the successors of a state using inclusion and decomposition depend on the value of $H.keys()$ which has evolved. Besides, it would only give a counterexample using states of \mathcal{B} while the user will prefer a counterexample using states of \mathcal{A} .

Our suggestion is that once the emptiness check of \mathcal{B} has failed, we search a counterexample in \mathcal{A} , but using the data structures computed by the emptiness check of \mathcal{B} to narrow the search. Computing the set of states of each SCC is easily

done by unwinding the *todo* and *SCC* search stacks: the set of states that belong to the top-most SCC is $\mathfrak{S}_n = rem_n \cup \{state_{f(root_n)}, state_{f(root_n)+1}, \dots, state_m\}$. The set of states from the previous SCC are then $\mathfrak{S}_{n-1} = rem_{n-1} \cup \{state_{f(root_{n-1})}, state_{f(root_{n-1})+1}, \dots, state_{f(root_n)-1}\}$, etc. Checking whether a state $s \in \mathcal{Q}_{\mathcal{A}}$ belongs to the i^{th} SCC then amounts to testing whether $\exists \mathfrak{S} \in \mathfrak{S}_i$ such that $s \in \mathfrak{S}$. It is thus possible to constraint the search in \mathcal{A} to remain inside these SCCs.

4. Approximative Emptiness Check

Consider Fig. 3 again. In the previous section we have seen that at lines 18–21 the algorithm of Fig. 4 takes the situation depicted by automaton \mathcal{B}_1 , where the emptiness check reaches a state $T \supseteq D$ such that D belongs to the search stack, and translates that into \mathcal{B}_2 to reuse existing states and build SCCs as soon as possible. We proved that this transformation preserves the result of the emptiness check ($\mathcal{B}_1 \emptyset$ -equivalent to \mathcal{B}_2).

We now turn to the situation \mathcal{B}_3 on Fig. 6, where the emptiness check examines a transition $\langle R, F, T \rangle$ such that $T \subseteq D$ and D is in the search stack. We can rewrite this transition as $\langle R, F, D \rangle$, as depicted by \mathcal{B}_4 , by replacing lines 18–20 of Fig. 4 by:

```

18   if  $\exists D \in H.keys()$  such that  $T \subseteq D \wedge H[D] > 0$ 
19     // Note the order of  $T$  and  $D$  above.
20      $todo.top().succ \leftarrow todo.top().succ \cup \{\langle R, F, D \rangle\}$ 

```

Assume that \mathcal{B}_3 is a \wp -UTGBA over some \mathcal{A} . Note that the above transformation breaks property (5) of definition 5, because s_3 and s_4 have no predecessor in R ; so \mathcal{B}_4 is *not* a \wp -UTGBA over \mathcal{A} . However by adding some transitions to \mathcal{A} to please property (5) it is possible to derive an \mathcal{A}' such that \mathcal{B}_4 is a \wp -UTGBA over \mathcal{A}' .

Therefore if the emptiness check algorithm finds an accepting component in \mathcal{B}_4 , there is an accepting run in \mathcal{A}' but not necessarily in \mathcal{A} . However since runs of \mathcal{A} are also runs of \mathcal{A}' , if the algorithm does not find any accepting component in \mathcal{B}_4 , no accepting run exists in \mathcal{A} and \mathcal{A}' .

In other words, this modified algorithm returns “empty” or “I don’t know”. As we will show in Section 5.4 this transformation is a lot faster than the other (“correct”) one presented in Section 3. Since model-checking is mainly interested in ensuring that some automaton is empty, it makes sense to try this semi-decision procedure first and fall back to the “correct” procedure if the answer isn’t “empty”.

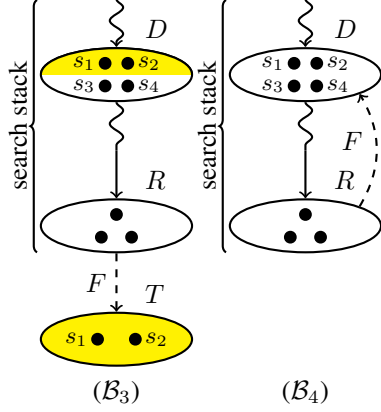


Figure 6. Inclusion checks in the search stack. We rewrite \mathcal{B}_3 to \mathcal{B}_4 .

Note by the way that if the modified algorithms return \perp but no inclusion has been done in the stack (one could count the number of times line 20 has been executed), then the automaton actually is nonempty.

5. Application to Symmetries

In this section we show how to exploit symmetries to construct a UTGBA \mathcal{B} that is a \wp -UTGBA over some automaton \mathcal{A} . We first define \mathcal{A} , the synchronized product of a *transition system* \mathcal{T} , representing the behavior of a system, and a *Transition-based Generalized Büchi Automaton* \mathcal{P} , representing the property to check.

Definition 7 (Labelled transition system). A labelled transition system is a tuple $\mathcal{T} = \langle \mathcal{Q}_{\mathcal{T}}, \mathcal{Q}_{\mathcal{T}}^0, \Sigma, \Delta_{\mathcal{T}} \rangle$, where

- $\mathcal{Q}_{\mathcal{T}}$ is a finite set of states,
- $\mathcal{Q}_{\mathcal{T}}^0 \subseteq \mathcal{Q}_{\mathcal{T}}$ is the set of initial states,
- $\Sigma = 2^{AP}$ is an alphabet, where AP is the set of atomic propositions,
- $\Delta_{\mathcal{T}} \subseteq \mathcal{Q}_{\mathcal{T}} \times \Sigma \times \mathcal{Q}_{\mathcal{T}}$ is a transition relation such that $\forall \langle s_1, p_1, d_1 \rangle, \langle s_2, p_2, d_2 \rangle \in \Delta_{\mathcal{T}}, p_1 = p_2 \iff (s_1, d_1) = (s_2, d_2)$.

The set of reachable states $\text{Reach}(\mathcal{T})$ is defined as usual. The latter condition on $\Delta_{\mathcal{T}}$ means that each transition is uniquely defined by its label (it is always possible to add more atomic propositions to the system to satisfy this constraint).

Definition 8 (Transition-based Generalized Büchi Automaton). A TGBA is a Büchi automaton with labels and generalized acceptance conditions on transitions. It is defined as a tuple $\mathcal{P} = \langle \mathcal{Q}_{\mathcal{P}}, \mathcal{Q}_{\mathcal{P}}^0, \Sigma, \mathcal{F}, \Delta_{\mathcal{P}} \rangle$, where

- $\mathcal{Q}_{\mathcal{P}}$ is a finite set of states,
- $\mathcal{Q}_{\mathcal{P}}^0 \subseteq \mathcal{Q}_{\mathcal{P}}$ is a set of initial states,
- $\Sigma = 2^{AP}$ is an alphabet,
- \mathcal{F} is a finite set of acceptance conditions,
- $\Delta_{\mathcal{P}} \subseteq \mathcal{Q}_{\mathcal{P}} \times \Sigma \times 2^{\mathcal{F}} \times \mathcal{Q}_{\mathcal{P}}$ is the transition relation, where each transition is labelled by a letter of Σ and a set of acceptance conditions of \mathcal{F} .

Definition 9 (Synchronized product). The synchronized product between $\mathcal{T} = \langle \mathcal{Q}_{\mathcal{T}}, \mathcal{Q}_{\mathcal{T}}^0, \Sigma, \Delta_{\mathcal{T}} \rangle$ and $\mathcal{P} = \langle \mathcal{Q}_{\mathcal{P}}, \mathcal{Q}_{\mathcal{P}}^0, \Sigma, \mathcal{F}, \Delta_{\mathcal{P}} \rangle$ is the UTGBA $\mathcal{A} = \mathcal{T} \otimes \mathcal{P}$ defined by $\mathcal{A} = \langle \mathcal{Q}_{\mathcal{A}}, \mathcal{Q}_{\mathcal{A}}^0, \Sigma, \mathcal{F}, \Delta_{\mathcal{A}} \rangle$, where

- $\mathcal{Q}_{\mathcal{A}} = \mathcal{Q}_{\mathcal{T}} \times \mathcal{Q}_{\mathcal{P}}$ is the set of states,
- $\mathcal{Q}_{\mathcal{A}}^0 = \mathcal{Q}_{\mathcal{T}}^0 \times \mathcal{Q}_{\mathcal{P}}^0$ is the set of initial states,
- $\Delta_{\mathcal{A}} \subseteq \mathcal{Q}_{\mathcal{A}} \times 2^{\mathcal{F}} \times \mathcal{Q}_{\mathcal{A}}$ is the transition relation between states such that $\exists \langle \langle s, q \rangle, F, \langle s', q' \rangle \rangle \in \Delta_{\mathcal{A}}$ iff $\exists \langle s, p, s' \rangle \in \Delta_{\mathcal{T}}, \exists \langle q, p', F, q' \rangle \in \Delta_{\mathcal{P}}$ and $p = p'$.

5.1. Symmetry-based Construction of a \wp -UTGBA

Now we construct \mathcal{B} , a \wp -UTGBA over $\mathcal{A} = \mathcal{T} \otimes \mathcal{P}$, using a technique introduced by Haddad et al. [8] but adapted to transition-based automata. The idea is to exploit the symmetries of \mathcal{T} in addition to those of the arcs of \mathcal{P} , to gather sets of nodes of \mathcal{A} .

Since these symmetries sit on group theory, we recall some elementary notions.

Definition 10. Let (G, \circ) be a group with a neutral element id , and let E be a set.

- An action of G over E is a mapping $G \times E \mapsto E$ such that the image (g, e) denoted by $g.e$ fulfills $\forall e \in E : id.e = e$ and $\forall g, g' \in G, (g \circ g').e = g.(g'.e)$.
- The isotropy subgroup $G_{E'}$ of a subset $E' \subseteq E$ is defined by $G_{E'} = \{g \in G \mid \forall e \in E', g.e \in E'\}$.
- For a subgroup H of G (denoted $H < G$), the orbit $H.e$ of $e \in E$ under H is $H.e = \{g.e \mid g \in H\}$.
- An action g of G can be straightforwardly extended to the powerset of E . For any $E' \subseteq E$, $g.E' = \{g.e \mid e \in E'\}$.

We can now characterize a symmetric transition system with respect to a group.

Definition 11 (Symmetric transition system with respect to a group). Let $\mathcal{T} = \langle \mathcal{Q}_{\mathcal{T}}, \mathcal{Q}_{\mathcal{T}}^0, \Sigma, \Delta_{\mathcal{T}} \rangle$ be a transition system and G be a group acting on AP . \mathcal{T} is said to be symmetric with respect to G iff every transition of \mathcal{T} has a “symmetric” transition with respect to any element of G and the action of G is congruent with respect to the transition relation: $\forall g \in G, \forall \langle s_1, p_1, d_1 \rangle \in \Delta_{\mathcal{T}}, \exists \langle s_2, p_2, d_2 \rangle \in \Delta_{\mathcal{T}}$ such that

$$\left\{ \begin{array}{l} s_1 \in \mathcal{Q}_T^0 \iff s_2 \in \mathcal{Q}_T^0, \\ p_2 = g.p_1 \text{ and} \\ \forall \langle s'_1, p'_1, d'_1 \rangle \in \Delta_T, s'_1 = d_1, \exists \langle s'_2, p'_2, d'_2 \rangle \in \Delta_T, s.t. \\ \quad s'_2 = d_2 \text{ and } p'_2 = g.p_2 \end{array} \right.$$

The action of the group on AP is extended to $\text{Reach}(T)$ by denoting $g.s$ the unique s_2 such that $\forall g \in G, \forall \langle s_1, p_1, d_1 \rangle \in \Delta_T, s_1 = s, \exists d_2 \in \mathcal{Q}_T, \langle s_2, g.p_1, d_2 \rangle \in \Delta_T$. (The uniqueness is due to the constraint on Δ_T in definition 7.)

Because G is group, a consequence of this definition is that $G.\mathcal{Q}_T^0 = \mathcal{Q}_T^0$.

These definitions allow us to give a possible construction of a \mathcal{B} .

Definition 12 (Symbolic Synchronized Product). Let $\mathcal{T} = \langle \mathcal{Q}_T, \mathcal{Q}_T^0, \Sigma, \Delta_T \rangle$ be a transition system symmetric w.r.t. a group G , $\mathcal{P} = \langle \mathcal{Q}_P, \mathcal{Q}_P^0, \Sigma, \mathcal{F}, \Delta_P \rangle$. The Symbolic Synchronized Product of \mathcal{T} and \mathcal{P} is a UTGBA $\mathcal{B} = \langle \mathcal{Q}_B, \mathcal{Q}_B^0, \mathcal{F}, \Delta_B \rangle$ where:

- $\mathcal{Q}_B^0 = \{ \langle G, G.s, q \rangle \mid s \in \mathcal{Q}_T^0, q \in \mathcal{Q}_P^0 \}$
- $\mathcal{Q}_B = \mathcal{Q}_B^0 \cup \mathcal{V}$ where \mathcal{V} is the set of tuples of the form $\langle H, O, q \rangle$ such that $H < G$, $O \subseteq \text{Reach}(T)$, $q \in \mathcal{Q}_P$, and $H.O = O$.
- Δ_B is defined by construction as follows: $\langle \langle H, O, q \rangle, F, \langle H', O', q' \rangle \rangle \in \Delta_B$ iff $\exists (s, s', p, p', F) \in O \times O' \times \Sigma \times \Sigma \times 2^{\mathcal{F}}$ such that $\langle s, p, s' \rangle \in \Delta_T, \langle q, p', F, q' \rangle \in \Delta_P$, and $p = p'$. Then $O' = (H \cap G.p').s'$ and $H' \subseteq G.O'$.

If $\mathcal{A} = \mathcal{T} \otimes \mathcal{P} = \langle \mathcal{Q}_A, \mathcal{Q}_A^0, \mathcal{F}, \Delta_A \rangle$, any state $\langle H, O, q \rangle \in \mathcal{Q}_B$ of \mathcal{B} represents the set $\{ \langle x, q' \rangle \in \mathcal{Q}_A \mid x \in O \wedge q' = q \}$ of states of \mathcal{A} . Hence we can write $\langle x, q' \rangle \in \langle H, O, q \rangle$, and this allows us to prove that \mathcal{B} is a \wp -UTGBA over \mathcal{A} [1].

5.2. Asymmetric Transition Systems

The method, as described here, is heavily dependent on the global symmetries of the transition system \mathcal{T} (i.e., on the group G). The bigger G is, the better the achieved reduction is. On a transition system mostly asymmetric, G will be very small, maybe the identity (i.e., no symmetries at all), and consequently the subgroups $H < G$ computed for each node will allow even less reductions.

There is one way to handle an asymmetric transitions system \mathcal{T} with this method: it is to rewrite it as a composition $\mathcal{T}_S \otimes \mathcal{C}$ where \mathcal{T}_S is globally symmetric w.r.t. a large G and \mathcal{C} is a constraint automaton such that $\mathcal{T}_S \otimes \mathcal{C} = \mathcal{T}$. Now, instead of constructing the symbolic synchronized product method of $\mathcal{T} \otimes \mathcal{P}$, we can construct it for $\mathcal{T}_S \otimes \mathcal{P}_C$ where

$\mathcal{P}_C = \mathcal{C} \otimes \mathcal{P}$. In other words, we shifted all asymmetries from the system automaton to the property automaton. This works because the method does not require a symmetric property automaton.

Haddad et al. [8] show one way to construct \mathcal{T}_S and \mathcal{C} from \mathcal{T} , while we are using a more optimal transition-based construction which is yet unpublished.

5.3. Operations for the emptiness check

For this construction to make sense (memory-wise), the set O of a state $\langle H, O, q \rangle$ must never be stored explicitly. In our implementation this is achieved by using a modified version of the *symbolic representation* of Well Formed Petri-Nets [2]. This data structure supports the following operations.

Let $T = \langle H_1, O_1, q_1 \rangle$ and $D = \langle H_2, O_2, q_1 \rangle$ be two states of \mathcal{B} . Since the sets O_i are not stored explicitly we cannot compare states with different H_i unless they are expanded into the set of states of \mathcal{A} they represent. To avoid this explicit expansion we introduce the following operation that we can use to unify the H_i .

The refinement of $\langle H_1, O_1, q_1 \rangle$ w.r.t. H_2 is the finite set $\text{Ref}(\langle H_1, O_1, q_1 \rangle, H_2) = \{ \langle H_1 \cap H_2, O_i, q_1 \rangle \mid i \in \mathbb{N} \}$ such that $\forall i, (H_1 \cap H_2).O_i = O_i$, and $\bigcup_i O_i = O_1$.

This allows us to check the inclusion of two states with: $\langle H_1, O_1, q_1 \rangle \subseteq \langle H_2, O_2, q_1 \rangle$ iff $\text{Ref}(\langle H_1, O_1, q_1 \rangle, H_2) \subseteq \text{Ref}(\langle H_2, O_2, q_1 \rangle, H_1)$.

Seeking visited states that include others (as on lines 15 and 18 of Fig. 4) can be sped up using a two level hash-table. Let G be the group acting on AP such that $\forall p \in AP, G.p = AP$. For a state $\langle H_1, O_1, q_1 \rangle$, pick an $s \in O_1$: $G.s$ is the coarsest equivalence class in which s can belong. We use $G.s$ as a key for our first-level hash table and q_1 as a key for the second level. Therefore when looking for states that include $\langle H_1, O_1, q_1 \rangle$, we only need to look through the states that share the same $G.s$ and q_1 .

$\text{Decomp}(\mathcal{B}, \langle R, F, T \rangle, D)$ is achieved using the refinement above. T is refined with respect to D 's H_2 , and D is refined with respect to H_1 , so we can compute the difference: $\{T_i\}_i = \text{Ref}(\langle H_1, O_1, q_1 \rangle, H_2) \setminus \text{Ref}(\langle H_2, O_2, q_1 \rangle, H_1)$. (The algorithm is improved by grouping some of these T_i s.)

5.4. Performance

The symbolic synchronized product of definition 12 has been implemented using the core of GreatSPN¹ [2], and the emptiness checks we presented are implemented in Spot² [6]. Connecting the two tools allowed us to compare different techniques.

¹<http://www.di.unito.it/~greatspn/>

²<http://spot.lip6.fr/>

model	n	SP+TEC			SSP+TEC			SSP+NSIEC			SSP+IEC			SSP+AEC		
		st.	tr.	T	st.	tr.	T	st.	tr.	T	st.	tr.	T	st.	tr.	T
WCS3	28	28	80	0.05	25	58	0.06	26	51	0.06	24	45	0.05	21	39	0.05
WCS4	28	78	250	0.06	77	202	0.14	66	176	0.17	43	96	0.10	29	57	0.06
WCS5	28	290	979	0.13	416	1309	2.76	294	1118	6.44	106	287	0.95	39	82	0.07
PO22	18	252	431	0.13	690	1307	0.95	738	1505	1.10	738	1505	1.10	738	1505	1.10
PO23	18	292	511	0.17	770	1441	1.48	750	1550	1.69	750	1550	1.69	750	1550	1.69
PO32	22	1173	2235	0.65	2184	4730	7.40	1400	3031	3.75	1400	3031	3.75	1392	2982	3.71
WCS3	22	99	279	0.06	94	255	0.13	91	250	0.14	73	194	0.13	30	70	0.07
WCS4	22	434	1485	0.13	602	2063	1.60	568	1980	2.17	297	940	1.07	64	177	0.15
WCS5	22	1889	7430	0.60	4224	17744	144	3905	16719	107	1370	4815	23.7	136	428	0.46
PO22	32	2484	5482	0.91	866	1817	2.16	864	1814	2.14	865	1814	2.14	864	1813	2.14
PO23	32	3253	7200	1.56	952	2030	3.68	868	1830	3.08	868	1831	3.09	868	1830	3.08
PO32	28	4617	10651	2.48	1334	2848	4.97	1294	2784	4.78	1294	2784	4.78	1294	2783	4.79

Table 1. States (st.) and transitions (tr.) explored by each algorithm on different models, and seconds (T) taken. All averaged on n properties.

Table 1 presents some measurements on two parametrized models, WCS [2] and PO [10], that were chosen for their symmetric or asymmetric properties: PO is a completely symmetric model (objects of similar nature behave identically), while WCS is an asymmetric model (objects behave differently). In both cases increasing the parameter increases the number of states of the system. Each of these models was synchronized against 50 randomly generated property automata: the table has been split to average the cases where the resulting product is empty separately from the cases where it is not. The reason is that the emptiness check has to check all states of an empty product, but can abort early if it is nonempty. The column n shows how many of the 50 cases were empty or not.

The abbreviations in the headers refer to how the product was constructed and checked for emptiness. SP is the synchronized product of definition 9 while SSP designates the symbolic synchronized product of definition 12. SP’s states are not sets, so it is checked with a traditional emptiness check (TEC) similar to the one of Fig. 4 but without any inclusion check or decomposition. IEC designates the emptiness check of Fig. 4. NSIEC is the same algorithm without lines 18–21 (i.e., No Stack Inclusion). Finally, AEC designates the approximative emptiness check of Section 4.

We observe that although SP is a lot faster than SSP, it visits many more states and hence requires a lot more memory. The different versions of our emptiness check algorithm can be compared in the four SSP columns: on the WCS model, adding inclusion checks in the removed states (NSIEC) reduces the size of the explored automaton (compared to TEC), and adding inclusion checks in the search stack (IEC) reduces the automaton further. Therefore, although the decomposition operation is costly (time-wise) it really helps reduce the memory footprint of the model-checking. The last column shows that approximation is indeed faster and constructs less states than all other methods (and yielded no false negatives in these experiments). On the PO model, the new emptiness check algorithms are not significantly better because the symmetric nature of the model offers little occasion for inclusion; still it can be seen that they do not incur any overhead.

Finally Fig. 7 shows the memory consumption of our implementation of the latest emptiness-check algorithm w.r.t. the traditional approach.

6. Conclusion

In this paper, we presented two novel emptiness check algorithms dealing with automata whose states are sets, and exploiting inclusions between these sets.

The results we obtained on the symmetry-based construction indicate that using inclusion and decomposition reduces the number of states by a great factor to the detri-

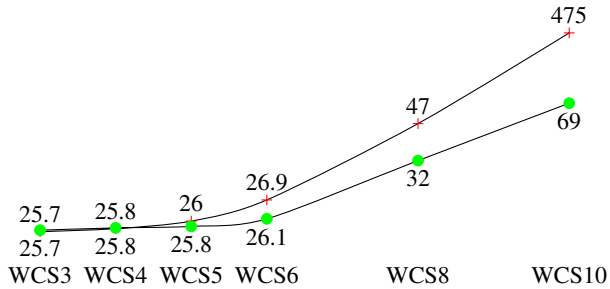


Figure 7. Average Memory used (in MB, on a log scale) by our implementations of SP+TEC (top) and SSP+AEC (bottom) to check empty products of increasing WCS models.

ment of the time. Actually, the loss of time is due to the way the states are encoded. Because this technique calls for a representation of sets that allow fast inclusion checks and decomposition, we expect that encoding sets with BDDs or DDDs [13] would improve the situation.

However this application should not occlude the fact that our algorithms have been developed in a more generic framework: any technique that can construct an automaton satisfying the five constraints of definition 5 could be used with these emptiness checks.

We believe it might be even possible to relax these constraints in order to support more methods, like the previously mentioned unfolding graphs [4] or observation graphs [9], that construct some “set automata” that do not strictly satisfy our conditions.

Another area that could be explored is the use of inclusion checks in NDFS emptiness checks. This doesn’t seem easy because NDFS algorithms will visit a state several times and require that the set of successors (re-)computed for a state remain the same: this is not guaranteed if the inclusion checks use a $H.keys()$ that changes. This is the same problem faced with counterexample generation in section 3.4, so ideas for either area could benefit the other.

References

- [1] S. Baarir and A. Duret-Lutz. Emptiness check of powerset Büchi automata. Technical report 2006/003, Université Pierre et Marie Curie, LIP6-CNRS, Paris, France, Oct. 2006.
- [2] S. Baarir, S. Haddad, and J.-M. Ilié. Exploiting partial symmetries in well-formed nets for the reachability and the linear time model checking problems. In *Proc. of WODES’04*, pages 223–228, Sept. 2004.
- [3] J.-M. Couvreur. On-the-fly verification of temporal logic. In *Proc. FM’99*, volume 1708 of *LNCS*, pages 253–271. Springer-Verlag, Sept. 1999.
- [4] J.-M. Couvreur, S. Grivet, and D. Poitrenaud. Designing a LTL model-checker based on unfolding graphs. In *Proc. of the ICATPN’00*, volume 1825 of *LNCS*. Springer-Verlag, June 2000.
- [5] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *Proc. of SPIN’05*, volume 3639 of *LNCS*, pages 143–158. Springer-Verlag, Aug. 2005.
- [6] A. Duret-Lutz and D. Poitrenaud. SPOT: an extensible model checking library using transition-based generalized Büchi automata. In *Proc. MASCOTS’04*, pages 76–83. IEEE Computer Society Press, Oct. 2004.
- [7] J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan’s algorithm. *Theoretical Computer Science*, 345(1):60–82, Nov. 2005.
- [8] S. Haddad, J.-M. Ilié, and K. Ajami. A model checking method for partially symmetric systems. In *Proc. of FORTE/PSTV’00*, volume 183 of *IFIP Conference Proceedings*. Kluwer, Oct. 2000.
- [9] S. Haddad, J.-M. Ilié, and K. Klai. Design and evaluation of a symbolic and abstraction-based model checker. In *Proc. of ATVA’04*, volume 3299 of *LNCS*, pages 198–210. Springer-Verlag, Oct. 2004.
- [10] J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Barrir, and T. Vergnaud. On the formal verification of middleware behavioral properties. In *Proc. of FMICS’04*, volume 133 of *ENTCS*, pages 139–157. Elsevier, Sept. 2004.
- [11] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *Proc. of TACAS’05*, volume 3440 of *LNCS*. Springer-Verlag, Apr. 2005.
- [12] H. Tauriainen. A note on the worst-case memory requirements of generalized nested depth-first search. Research Report A96, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, Sept. 2005.
- [13] Y. Thierry-Mieg, J.-M. Ilié, and D. Poitrenaud. A symbolic symbolic state space representation. In *Proc. of FORTE’04*, volume 3235 of *LNCS*, Sept. 2004.
- [14] A. Valmari. The state explosion problem. In *Lectures on Petri Nets 1: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1998.
- [15] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of Banff’94*, volume 1043 of *LNCS*, pages 238–266. Springer-Verlag, 1996.