Emptiness Check of Powerset Büchi Automata using Inclusion Tests

Souheib Baarir, Alexandre Duret-Lutz

Emptiness Check of Powerset Büchi Automata using Inclusion Tests

Souheib Baarir and Alexandre Duret-Lutz

Technical report 2006/003
Université Pierre et Marie Curie, LIP6-CNRS, Paris 6, France
souheib.baarir@lip6.fr, alexandre.duret-lutz@lip6.fr

Abstract. A possible attack on the state explosion of the automata-theoretic approach to model-checking is to build an automaton $\mathcal{B}$ whose states represent sets of states of the original automaton $\mathcal{A}$ to check for emptiness. This paper introduces two emptiness checks for Büchi automata whose states represent sets that may include each other. The first check on $\mathcal{B}$ is equivalent to a traditional emptiness check on $\mathcal{A}$ but uses inclusion tests to direct and further reduce the on-the-fly construction of $\mathcal{B}$. The second check is impressively faster but may return false negatives. We illustrate and benchmark both using a symmetry-based reduction.

1 Introduction

The automata-theoretic approach to model-checking [12] 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 used to represent the behavior of the system, and hence in the resulting size of the product automaton that has to be explored by the emptiness check. Several techniques have been proposed to reduce the size of both automata [11].

If the state space has a global symmetry (for example a client/server system where all clients behave identically), it is easy to "fold" the automaton by factorizing the representations of its symmetric states. However such globally symmetric automata are uncommon in practice.

The method of *partial symmetries*, introduced by Haddad et al. [6], tackles the reduction of non-globally symmetric automata, by exploiting symmetries locally (for example if two clients have different access priorities to a server, they are asymmetric when they access the server, but they can still be symmetric the rest of the time). Roughly speaking, this method works by partitioning the set of successors $s_1, s_2, \ldots, s_n$ of a state of the original automaton, and using these partitions as the states $S_1, S_2, \ldots, S_m$ of the reduced automaton. However, because this is done locally in each state, there is no guarantee that a state s of the original automaton may be represented by only once state S in the reduced automaton. This reduced automaton $\mathcal{B}$ can then be checked for emptiness (using any mainstream emptiness check algorithm [9, 4, 5]) 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 $\mathcal{A}$).

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

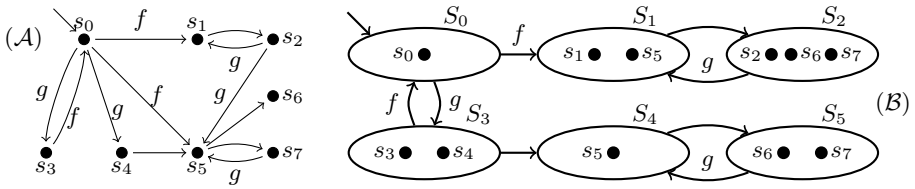

Fig. 1. An example of powerset automaton $\mathcal{B}$ for an automaton $\mathcal{A}$, with $\mathcal{F}_\mathcal{A} = \mathcal{F}_\mathcal{B} = \{f, g\}$.

automaton (which are sets of states). 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., $S_4 \subseteq S_1$).

As such, this method may well construct a "reduced" automaton that has more states that the original automaton! Indeed, if we call $\mathcal{A}$ the original automaton and $\mathcal{Q}_\mathcal{A}$ its set of states, this technique constructs an automaton $\mathcal{B}$ where $\mathcal{Q}_\mathcal{B} \subseteq 2^{\mathcal{Q}_\mathcal{A}}$ and may have at worst $2^{|\mathcal{Q}_\mathcal{A}|}$ states. However in many practical cases we still have $|\mathcal{Q}_\mathcal{B}| < |\mathcal{Q}_\mathcal{A}|$.

Usually the state space is constructed on-the-fly during the emptiness check, and we show that during this emptiness check it is possible to perform *inclusion tests* to limit the number of constructed states, further reducing the complexity of the verification. Because this idea can be useful in general (i.e., not only to symmetries), we present this new emptiness check 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 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*", and Section 5 briefly discusses how to generate counterexamples. 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 $\mathcal{A}$: in Section 6 we adapt the technique of Haddad et al. [6] to show how to construct a $\mathcal{B}$ using symmetries, and we prove that this construction satisfies the requirements of our algorithm. We benchmark this construction in Section 7 and show that although theoretically the algorithm may still explore $2^{|\mathcal{Q}_\mathcal{A}|}$ 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* $\mathcal{A} = \langle \mathcal{Q}_\mathcal{A}, \mathcal{Q}^0_\mathcal{A}, \mathcal{F}_\mathcal{A}, \Delta_\mathcal{A} \rangle$ *where*

– $\mathcal{Q}_\mathcal{A}$ *is a finite set of elements called* states*,*

– $\mathcal{Q}^0_\mathcal{A} \subseteq \mathcal{Q}_\mathcal{A}$ *is a set of initial states,*

– $\mathcal{F}_\mathcal{A}$ *is a finite set of elements called* acceptance conditions*,*

– $\Delta_\mathcal{A} \subseteq \mathcal{Q}_\mathcal{A} \times 2^{\mathcal{F}_\mathcal{A}} \times \mathcal{Q}_\mathcal{A}$ *is the transition relation, where each transition carries a (possibly empty) set of acceptance conditions of* $\mathcal{F}_\mathcal{A}$*.*

Definition 2 (Reachable states)**.** *Let* $\mathcal{A} = \langle \mathcal{Q}_\mathcal{A}, \mathcal{Q}^0_\mathcal{A}, \mathcal{F}_\mathcal{A}, \Delta_\mathcal{A} \rangle$ *be a UTGBA. A state* s *is* reachable *if* $s \in \mathcal{Q}^0_\mathcal{A}$ *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* $\Delta_\mathcal{A}$*, starting at an initial state* $s_0 \in \mathcal{Q}^0_\mathcal{A}$*, and ending on state* $s_n = s$*. We denote* $\mathrm{Reach}(\mathcal{A})$ *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* $\Delta_\mathcal{A}$*, starting at an initial state* $s_0 \in \mathcal{Q}^0_\mathcal{A}$*. A run is* accepting *if* $\forall f \in \mathcal{F}_\mathcal{A}$, $\forall i \geqslant 0$, $\exists j \geqslant i$*, such that* $f \in F_j$*, i.e., if its transitions are labelled by each acceptance condition infinitely often.*

We denote $\mathrm{Run}(\mathcal{A})$ *and* $\mathrm{Acc}(\mathcal{A})$ *the set of all runs and the set of all accepting runs of* $\mathcal{A}$*.*

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

For instance on Fig. 1, $\mathcal{B}$ accepts only one run: $\mathrm{Acc}(\mathcal{B}) = \{S_0 S_3 S_0 S_3 S_0 S_3 \ldots\}$.

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* ξ*-equivalent iff either both automata have an accepting run, or none have.*

$$\mathcal{A} \overset{\xi}{\equiv} \mathcal{B} \quad \textit{iff} \quad \mathrm{Acc}(\mathcal{A}) = \emptyset \iff \mathrm{Acc}(\mathcal{B}) = \emptyset$$

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 ξ-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 the example of Fig. 1.

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

$$\mathcal{Q}_\mathcal{B} \subseteq 2^{\mathcal{Q}_\mathcal{A}} \setminus \{\emptyset\} \tag{1}$$

$$\mathcal{F}_\mathcal{B} = \mathcal{F}_\mathcal{A} \tag{2}$$

$$\bigcup_{S \in \mathcal{Q}^0_\mathcal{B}} S = \mathcal{Q}^0_\mathcal{A} \tag{3}$$

$$\begin{aligned} &\forall \langle s, F, s' \rangle \in \Delta_\mathcal{A},\ \forall S \in \mathrm{Reach}(\mathcal{B}), \\ &\qquad s \in S \implies \exists S' \in \mathcal{Q}_\mathcal{B} \text{ such that } s' \in S', \text{ and } \langle S, F, S' \rangle \in \Delta_\mathcal{B} \end{aligned} \tag{4}$$

$$\forall \langle S, F, S' \rangle \in \Delta_\mathcal{B},\ \forall s' \in S',\ \exists s \in S,\ \text{such that } \langle s, F, s' \rangle \in \Delta_\mathcal{A} \tag{5}$$

Proposition 1. *Let* $\mathcal{A} = \langle \mathcal{Q}_\mathcal{A}, \mathcal{Q}^0_\mathcal{A}, \mathcal{F}_\mathcal{A}, \Delta_\mathcal{A} \rangle$ *and* $\mathcal{B} = \langle \mathcal{Q}_\mathcal{B}, \mathcal{Q}^0_\mathcal{B}, \mathcal{F}_\mathcal{B}, \Delta_\mathcal{B} \rangle$ *be two UTGBAs such that* $\mathcal{B}$ *is a* $\wp$*-UTGBA over* $\mathcal{A}$*. Then* $\mathcal{A} \overset{\xi}{\equiv} \mathcal{B}$*.*

Proof. We want to show that $\exists\sigma \in \mathrm{Acc}(\mathcal{A}) \iff \exists\sigma' \in \mathrm{Acc}(\mathcal{B})$.
($\Longrightarrow$) Let $\sigma = \langle s_0, F_0, s_1\rangle\langle s_1, F_1, s_2\rangle \cdots \in \mathrm{Acc}(\mathcal{A})$. Since $s_0 \in \mathcal{Q}^0_\mathcal{A}$ we can use (3) and find an $S_0 \in \mathcal{Q}^0_\mathcal{B}$ 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 $S_1 \in \mathcal{Q}_\mathcal{B}$ such that $s_1 \in S_1$ and $\langle S_0, F_0, S_1\rangle \in \Delta_\mathcal{B}$. Likewise, because S_1 is reachable in $\mathcal{B}$ and contains s_1 by construction, we can use (4) again to find an $S_2 \in \mathcal{Q}_\mathcal{B}$ such that $s_2 \in S_1$ and $\langle S_1, F_1, S_2\rangle \in \Delta_\mathcal{B}$. Iterating (4) we can construct a sequence $\sigma' = \langle S_0, F_0, S_1\rangle\langle S_1, F_1, S_2\rangle \cdots \in \mathrm{Run}(\mathcal{B})$ such that $s_i \in S_i$ for all i. Since $\mathcal{F}_\mathcal{B} = \mathcal{F}_\mathcal{A}$ (2) and σ' visits each acceptance condition as often as σ, $\sigma' \in \mathrm{Acc}(\mathcal{B})$.
($\Longleftarrow$) Let $\sigma' = \langle S_0, F_0, S_1\rangle\langle S_1, F_1, S_2\rangle \cdots \in \mathrm{Acc}(\mathcal{B})$. Let's build a tree whose nodes (except the root) are states of $\mathcal{A}$. Let's call $\bot$ 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 $\langle s, F_{n-1}, s'\rangle \in \Delta_\mathcal{A}$; (5) guarantees that such a node s exists. The father of any node at depth 1 is $\bot$. All edges of this tree, expect those leaving the root node, correspond to transitions of $\Delta_\mathcal{A}$.
The set of nodes at depth $n > 0$ is a subset of $\mathcal{Q}_\mathcal{A}$, 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 $\bot$) $\langle s_0, F_0, s_1\rangle\langle s_1, F_1, s_2\rangle \cdots$ is an accepting run of $\mathcal{A}$. Indeed it is a run of $\mathcal{A}$ ($s_0 \in \mathcal{Q}^0_\mathcal{A}$) that visits each acceptance conditions as often as σ'. □

We now develop two propositions that introduce the emptiness check algorithm. 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 $\mathcal{B} = \langle \mathcal{Q}_\mathcal{B}, \mathcal{Q}^0_\mathcal{B}, \mathcal{F}, \Delta_\mathcal{B}\rangle$ be a $\wp$-UTGBA over $\mathcal{A} = \langle \mathcal{Q}_\mathcal{A}, \mathcal{Q}^0_\mathcal{A}, \mathcal{F}, \Delta_\mathcal{A}, \rangle$ and consider two states T and D of $\mathcal{Q}_\mathcal{B}$ such that $D \subseteq T$. We have*

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

Proof. We prove the contraposition: $\mathrm{Acc}(\mathcal{B}[\{D\}]) \neq \emptyset \implies \mathrm{Acc}(\mathcal{B}[\{T\}]) \neq \emptyset$. Consider $\sigma = \langle D_0, F_0, D_1\rangle\langle D_1, F_1, D_2\rangle \cdots \in \mathrm{Acc}(\mathcal{B}[\{D\}])$. Using (5) in the same way as we did in the proof of the ($\Longleftarrow$) part of proposition 1, we can find a sequence $\sigma' = \langle d_0, F_0, d_1\rangle\langle d_1, F_1, d_2\rangle \cdots$ such that $\forall i \geqslant 0, d_i \in D_i$. Consider the first transition of σ': $\langle d_0, F_0, d_1\rangle$. Since $D \subseteq T$, we have $d_0 \in T$, therefore we can apply proposition (4) to find a set T_1 such that $d_1 \in T_1$ and $\langle T, F, T_1\rangle \in \Delta_\mathcal{B}$. Then because $d_1 \in T_1$ we can apply proposition (4) again to find $\langle T_1, F, T_2\rangle \in \Delta_\mathcal{B}$. And iterating this operation we construct an accepting run $\sigma'' = \langle T, F_0, T_1\rangle\langle T_1, F_1, T_2\rangle \cdots$ of $\mathcal{B}[\{T\}]$. □

The following proposition allows us to split a transition $\langle R, F, T\rangle$ into a set of transitions $\langle R, F, T_1\rangle, \ldots, \langle R, F, T_n\rangle$ with $T_1 \cup \cdots \cup T_n = T$, while preserving ξ-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 $T_1, \ldots, T_n$ as initial states, and that is ξ-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.

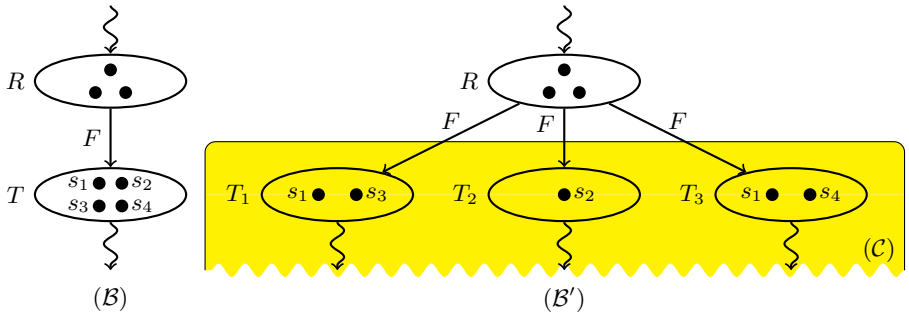

Fig. 2. Example of decomposition of a transition $\langle R, F, T\rangle$ using proposition 3.

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

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

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

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

Proof. By definition $\mathcal{B}'$ satisfies properties (1) and (2) of definition 5. We have to prove that $\mathcal{B}'$ also satisfies properties (3) to (5).

(3) We should show that $\bigcup_{S\in\mathcal{Q}^0_{\mathcal{B}'}} S = \mathcal{Q}^0_\mathcal{A}$. Since $\mathcal{B}$ is a $\wp$-UTGBA over $\mathcal{A}$, we have $\bigcup_{S\in\mathcal{Q}^0_\mathcal{B}} S = \mathcal{Q}^0_\mathcal{A}$. If $T \notin \mathcal{Q}^0_\mathcal{B}$ we have $\mathcal{Q}^0_{\mathcal{B}'} = \mathcal{Q}^0_\mathcal{B}$ and prop. (3) holds. Otherwise, because $\mathcal{C}$ is a $\wp$-UTGBA over $\mathcal{A}[T]$ we have $\bigcup_{S\in\mathcal{Q}^0_\mathcal{C}} S = T$, so (3) holds too.

(4) Let $\langle s, F, s'\rangle \in \Delta_\mathcal{A}$ be a transition of $\mathcal{A}$, and $S \in \text{Reach}(\mathcal{B}')$ such that $s \in S$. To prove (4) we must exhibit a transition $\langle S, F, S'\rangle \in \Delta_{\mathcal{B}'}$ such that $s' \in S'$. We distinguish three cases that cover all possible states of $\text{Reach}(\mathcal{B}')$:

- If $S \in (\text{Reach}(\mathcal{B}) \setminus \{R\}) \vee (S = R \wedge s' \notin T)$ then because $\mathcal{B}$ is a $\wp$-UTGBA over $\mathcal{A}$ its property (4) guarantees the existence of such a transition.
- If $S = R \wedge s' \in T$, because $\mathcal{C}$ is a $\wp$-UTGBA over $\mathcal{A}[T]$ its property (3) ensures that $\exists T' \in \mathcal{Q}^0_\mathcal{C}$ such that $s' \in T'$, therefore $\langle S, F, T'\rangle \in \Delta_{\mathcal{B}'}$ by definition of $\mathcal{B}'$.
- Finally, if $S \in \text{Reach}(\mathcal{C})$ then because $\mathcal{C}$ is a $\wp$-UTGBA over $\mathcal{A}[T]$ its property (4) guarantees the existence of such a transition.

(5) To prove (5) we must check all tuples $\langle S, F, S'\rangle$ of $\Delta_{\mathcal{B}'}$ and ensure that $\forall s' \in S', \exists s \in S, \langle s, F, s'\rangle \in \Delta_\mathcal{A}$. Since $\mathcal{B}$ is a $\wp$-UTGBA over $\mathcal{A}$ this property holds for any transition $\langle S, F, S'\rangle$ in $\Delta_\mathcal{B}$. Similarly because $\mathcal{C}$ is a $\wp$-UTGBA over $\mathcal{A}[T]$, (5) is true for any tuple in $\Delta_\mathcal{C}$. Only the following set of transition remains to be checked: $\{\langle R, F, T'\rangle \mid T' \in \mathcal{Q}^0_\mathcal{C}\}$. However because (5) was true for $\langle R, F, T\rangle$ it also holds for all these transitions. □

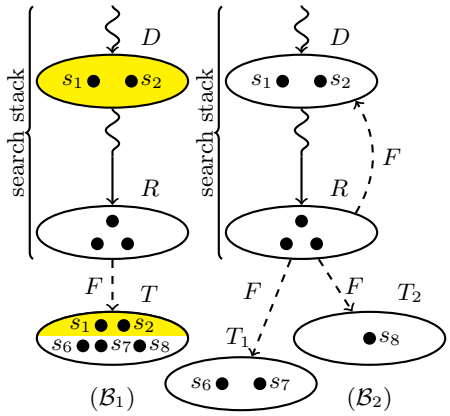

Fig. 3. Inclusions checks in the search stack. We rewrite $\mathcal{B}_1$ to $\mathcal{B}_2$.

3 Emptiness Check of $\wp$-UTGBA

A generalized Büchi automaton can accept a run (i.e., is nonempty) if it contains a strongly connected component (SCC) reachable from some initial state, and in which all acceptance conditions appear. Checking the emptiness of an automaton amounts to checking for the existence of such an SCC. Several algorithms have been designed with this aim (we refer the reader to Couvreur et al. [4] for an extensive bibliography on this subject) and the one we present here is derived from that of Couvreur [2].

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 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 in the above sense, 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.

Adaptation to $\wp$-UTGBA. The new algorithm differs from the original in two points. First, the check of removed states above 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 the same SCC. For the emptiness check it would be beneficial to split the transition $\langle R, F, T\rangle$ as on $\mathcal{B}_2$: 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 $\mathcal{B} = \langle \mathcal{Q}_\mathcal{B}, \mathcal{Q}^0_\mathcal{B}, \mathcal{F}, \Delta_\mathcal{B}\rangle$ be a $\wp$-UTGBA over some $\mathcal{A}$. 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 $D \in \mathcal{Q}^0_\mathcal{C}$ (and the other states of $\mathcal{Q}^0_\mathcal{C}$ will by definition complete T),
- and $\Delta_\mathcal{C}$ should not add transitions to the states of $\mathcal{B}$, in other words $\{\langle S, F, S'\rangle \in \Delta_\mathcal{C} \mid S \in \text{Reach}(\mathcal{B})\} \subseteq \Delta_\mathcal{B}$.

Decomp returns a pair $\mathcal{B}', \mathcal{Q}^0_\mathcal{C}$: 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 on 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.

Correctness of this algorithm. We use the following notations in the proof. At any point of the execution we denote the contents of the *todo* and *SCC* stacks as follows:

$$\begin{aligned} \mathit{todo} =& \langle \mathit{state}_0, \mathit{succ}_0\rangle\langle \mathit{state}_1, \mathit{succ}_1\rangle \cdots \langle \mathit{state}_m, \mathit{succ}_m\rangle \\ \mathit{SCC} =& \langle \mathit{root}_0, \mathit{la}_0, \mathit{acc}_0, \mathit{rem}_0\rangle\langle \mathit{root}_1, \mathit{la}_1, \mathit{acc}_1, \mathit{rem}_1\rangle \cdots \langle \mathit{root}_n, \mathit{la}_n, \mathit{acc}_n, \mathit{rem}_n\rangle \end{aligned}$$

todo is a DFS stack of pairs $\langle \mathit{state}, \mathit{succc}\rangle$ where *succ* is the set of outgoing transitions of *state* that haven't been considered yet. We call $\mathit{state}_0 \ldots \mathit{states}_m$ 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 S_i of states belonging to the i^{th} SCC as follows:

$$\begin{aligned} S_i =& \{s \in \mathcal{Q}_\mathcal{B} \mid \mathit{root}_i \leqslant H[s] < \mathit{root}_{i+1}\} \qquad \text{for } 0 \leqslant i < n \\ S_n =& \{s \in \mathcal{Q}_\mathcal{B} \mid \mathit{root}_n \leqslant H[s]\} \end{aligned}$$

acc_i is the set of acceptance conditions traversed by transitions between states of 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,
- finally 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 move from the *unexplored* set to the *active* set, and the function "DFSpop" is the only place where it can move from the *active* set to the *removed* set.

To prove the correctness of the algorithm we show that the following invariants are preserved at every line of "main":

```
1  // Let B = ⟨Q_B, Q_B^0, F, Δ_B⟩ be the input automaton to check.
2  todo: stack of ⟨state ∈ Q_B, succ ⊆ Δ_B⟩
3  SCC: stack of ⟨root ∈ ℕ, la ⊆ F, acc ⊆ F, rem ⊆ Q_B⟩
4  H: map of Q_B ↦ ℕ
5  max ← 0
6
7  main():
8      forall S^0 ∈ Q_B^0
9          DFSpush(∅, S^0)
10         while ¬todo.empty()
11             if todo.top().succ = ∅
12                 DFSpop()
13             else
14                 pick one ⟨R, F, T⟩ off todo.top().succ
15                 if ∃D ∈ H.keys() such that (T ⊆ D) ∧ H[D] = 0
16                     continue
17                 elsif T ∉ H
18                   if ∃D ∈ H.keys() such that D ⊆ T ∧ H[D] > 0
19                       B, Q_C^0 ← Decomp(B, ⟨R, F, T⟩, D)
20                       todo.top().succ ← todo.top().succ ∪ {⟨R, F, D⟩} ∪ {(R, F, T') | T' ∈ Q_C^0}
21                   else
22                       DFSpush(F, T)
23                 elsif H[T] > 0
24                    if merge(F, H[T]) = F
25                       return ⊥
26     return ⊤
27
28 DFSpush(F ⊆ F, S ∈ Q):
29     max ← max + 1
30     H[S] ← max
31     SCC.push(⟨max, F, ∅, ∅⟩)
32     todo.push(⟨S, {⟨R, F, T⟩ ∈ Δ_B | R = S}⟩)
33
34 DFSpop():
35     ⟨S, _⟩ ← todo.pop()
36     SCC.top().rem.insert(s)
37     if H[S] = SCC.top().root
38         forall R ∈ SCC.top().rem
39             H[R] ← 0
40         SCC.pop()
41
42 merge(F ⊆ F, n ∈ ℕ):
43     r ← ∅
44     while (n < SCC.top().root)
45         F ← (F ∪ SCC.top().acc ∪ SCC.top().la)
46         r ← r ∪ SCC.top().rem
47         SCC.pop()
48     SCC.top().acc ← SCC.top().acc ∪ F
49     SCC.top().rem ← SCC.top().rem ∪ r
50     return SCC.top().acc
```

Fig. 4. Emptiness check of a $\wp$-UTGBA.

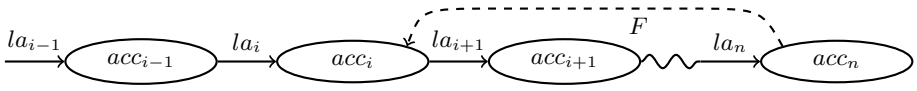

Fig. 5. The meaning of *la* and *acc* in *SCC*.

Proposition 4. *$m \geqslant n$ (in the above notation for todo and SCC) and there exists a strictly increasing function f such that $\forall i \leqslant n, root_i = H[state_{f(i)}]$. In other words, $root_0, root_1, \ldots, root_n$ is a subsequence of $H[state_0], H[state_1], \ldots, H[state_m]$. (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 $i \leqslant n$ the subgraph induced by the states of S_i is a SCC. Furthermore there exists a cycle in this SCC that visits all the acceptance conditions of acc_i. Finally $S_0, S_1, \cdots, S_n$ is a partition of the set of active states.*
Proposition 6. *$\forall i < n, \exists s \in S_i, \langle s, la_{i+1}, state_{f(i+1)}\rangle \in \Delta$.*
Proposition 7. *For any $i < n$, rem_i holds all states of S_i not on the search path.*
Proposition 8. *For any* removed *states q,* $\mathrm{Acc}(\mathcal{B}[\{q\}]) = \emptyset$.

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.

Proof. All propositions 4-8 hold when line 10 is first reached. There is only one element on *todo* and *SCC* and the way it has been pushed by "DFSpush" guarantees prop. 4. S_0 contains only one states, so prop. 5 holds. Since $n = m = 0$, prop. 6 and 7 are trivially true. No state have been removed yet so prop. 8 is true too.

When a transition $\langle S, F, T\rangle$ is picked off $succ_m$, we fall in one of the following cases:

- $H[T] = 0$ means that T is a removed state (lines 15–16). Because of prop. 8 neither T nor any of its descendants can be part of an accepting run: T can be safely ignored from the search. Since no data structure is altered, prop. 4-8 are preserved.
- $\exists D, H[D] = 0 \wedge T \subseteq D$. There exists a removed state D that contains T (lines 15–16). Using prop. 2 we can ignore T as above, and prop. 4-8 are preserved.
- $T \notin H$ and $\exists D, H[D] > 0 \wedge D \subseteq T$. There exists an active state D that is included in the current destination T, and T hasn't been visited yet (lines 17–20).

  Since we have already seen D we would like to avoid that part of T. We therefore apply Decomp to split T into D plus several other (possibly new) states that completes T. Since this operation does not add any transitions to states that exist in $\mathcal{B}$ (this is a constraint we stated on Decomp) and does not remove any visited state or transition, we can overwrite $\mathcal{B}$ with the automaton produced by Decomp without affecting the DFS. We simply replace $\langle S, F, T\rangle$ that have been removed from $succ_m$ by the list of transitions to the D and $\mathcal{Q}^0_{\mathcal{C}}$.

  This case is a no-op for the whole algorithm, as if it had worked with the new automaton since the beginning. Prop. 4-8 are unaffected.
- $T \notin H$ and we didn't find an active D included in T (lines 21–22). T is a trivial SCC, we number it in H, stack it onto *SCC* as a new root, and push it onto the DFS stack *todo*. Doing so guaranties prop. 4-6 and does not affect prop. 7-8.

– $H[T] > 0$, i.e., T is an active state (lines 23–25). Let $root_i$ be the greatest root such that $root_i < H[T]$, we denote $r_i = state_{f(i)}$ the associated state. Because of prop. 5, r_i and T are in the same SCC. Moreover, because of r_i and R are on the depth-first search path, r_i can reach R. Since we are considering the transition $\langle R, F, T\rangle$, we conclude that r_i, R, and T belong to the same SCC.

  We therefore merge all SCCs above $root_i$. The new SCC is the union of $S_i, \ldots, S_n$, and inside this SCC these exists a cycle that traverses the acceptance conditions $acc_i, \ldots, acc_n$ of any former SCC, as well as the those $la_{i+1}, \ldots, la_n$ of the intervening transitions. Fig. 5 depicts the situation before the merge, showing acceptance conditions in and between the SCCs.

  The function "merge" takes care of merging these acceptance conditions to preserve prop. 5 and also merges all the removed states to preserve prop. 7. The other three propositions are not affected by this operation.

  The "merge" operation returns the acceptance conditions of the merged SCC. If this is $\mathcal{F}$ then the algorithm can answer negatively immediately (line 25).

We now consider the case where $succ_m = \emptyset$ (lines 11-12). The properties of the DFS imply the algorithm has explored all the successors of $state_m$ and their descendants.

$state_m$ is popped off the search path on line 35, and to preserve prop. 7 it is added to rem_n on line 36. Popping $state_m$ will not affect prop. 4 as long as this state is not the root of a SCC. If it is, the SCC has to be popped too. Doing so does not affect prop. 6 but to preserve prop. 5 we should also remove the states of S_i from the *active* set. At this point we know that the top-most SCC is maximal:

– no unexplored state can be part of the SCC because we have explored all the descendants of this state,

– no active states from a lower SCC can be part of this one, because we would have merged them if a descendant of $state_m$, could reach it,

– no removed state can belong to this SCC thanks to prop. 8.

We also know that this maximal SCC doesn't contain any accepting cycle, otherwise the algorithm would have stopped on line 24 when adding this cycle. Therefore $\forall q \in S_i, \mathrm{Acc}(\mathcal{B}[q]) = \emptyset$, and we can mark all these states as *removed* without invalidating prop. 8. Because of prop. 7 at line 38 rem_n contains all the states of S_i, so this loop actually removes all the states of the SCC as required by prop. 5.

If the algorithm ends line 26, the *todo* stack is empty. By prop. 4 this means that SCC is empty too, and by prop. 5 it means that the *active* set is empty. Since the DFS has explored all the reachable states of the automaton, we conclude that all reachable states have been *removed*, which, because of prop. 8 implies that $\mathrm{Acc}(\mathcal{B}) = \emptyset$.

We have shown that if the algorithm terminates on line 25 there exists an accepting run, and if the algorithm terminates on line 26 there does not. The algorithm is guaranteed to terminate because it performs a DFS on a finite automaton. Because of its use of Decomp the algorithm can explore more than $\mathcal{Q}_\mathcal{B}$ states, but it cannot explore more than $2^{\mathcal{Q}_\mathcal{A}}$ states if we denote $\mathcal{Q}_\mathcal{A}$ the number of states of $\mathcal{A}$ (recall the automaton being checked is a $\wp$-UTGBA over $\mathcal{A}$).

We conclude that the algorithm returns $\bot$ iff $\mathcal{B}$ contains an accepting run. □

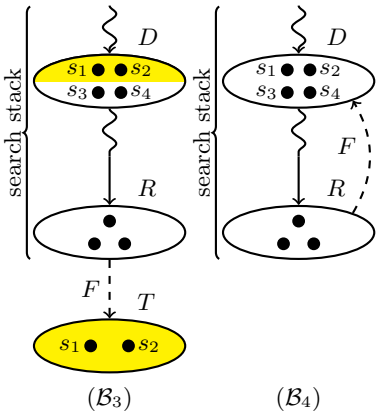

Fig. 6. Inclusions checks in the search stack. We rewrite $\mathcal{B}_3$ to $\mathcal{B}_4$.

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 ($\mathcal{B}_1$ ξ-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 ∃D ∈ H.keys() such that T ⊆ D ∧ H[D] > 0
19     // Note the order of T and D above.
20     todo.top().succ ← todo.top().succ ∪ {⟨R, F, D⟩}
```

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 7 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" if the answer isn't "empty".

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

5 Counterexamples

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

Unlike mainstream emptiness checks working with nested depth-first search on non-generalized Büchi automata [9] whose search stacks directly provide an accepting run on exit, SCC-based emptiness checks can only describe the counterexample in term of SCCs. To produce a genuine accepting run from the stack of SCCs, these SCCs have to be searched again [4]. First, one should compute an accepting cycle inside the top-most SCC, and then try to reach it from the initial state. All these searches are localized, because the states we explore necessarily belong to the the SCCs in the stack.

Our algorithm can produce such a stack of SCCs on failure (this is the *SCC* stack), but our case is worsened by the use of inclusions and decompositions. The algorithm described by Couvreur et al. [4] could be used only with guarantee that during its new exploration of these SCC it would visit the same states that the emptiness check visited. 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 $S_n = rem_n \cup \{state_{f(root_n)}, state_{f(root_n)+1}, \ldots, state_m\}$. The set of states from the previous SCC are then $S_{n-1} = rem_{n-1} \cup \{state_{f(root_{n-1})}, state_{f(root_{n-1})+1}, \ldots, state_{f(root_n)-1}\}$, etc. Checking whether a state $s \in \mathcal{Q}_\mathcal{A}$ belongs to the i^{th} SCC then amounts to testing whether $\exists S \in S_i$ such that $s \in S$. It is thus possible to constraint the search in $\mathcal{A}$ to remain inside these SCCs.

6 Symmetries and $\wp$-UTGBA

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* $\mathcal{T} = \langle \mathcal{Q}_\mathcal{T}, \mathcal{Q}^0_\mathcal{T}, \Sigma, \Delta_\mathcal{T} \rangle$*, where*

- $\mathcal{Q}_\mathcal{T}$ *is a finite set of states,*
- $\mathcal{Q}^0_\mathcal{T} \subseteq \mathcal{Q}_\mathcal{T}$ *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 $\mathrm{Reach}(\mathcal{T})$ *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 verify 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* $\mathcal{P} = \langle \mathcal{Q}_\mathcal{P}, \mathcal{Q}^0_\mathcal{P}, \Sigma, \mathcal{F}, \Delta_\mathcal{P} \rangle$*, where*

- $\mathcal{Q}_\mathcal{P}$ *is a finite set of* states*,*
- $\mathcal{Q}^0_\mathcal{P} \subseteq \mathcal{Q}_\mathcal{P}$ *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* $\mathcal{T} = \langle \mathcal{Q}_\mathcal{T}, \mathcal{Q}^0_\mathcal{T}, \Sigma, \Delta_\mathcal{T} \rangle$ *and* $\mathcal{P} = \langle \mathcal{Q}_\mathcal{P}, \mathcal{Q}^0_\mathcal{P}, \Sigma, \mathcal{F}, \Delta_\mathcal{P} \rangle$ *is the UTGBA* $\mathcal{A} = \mathcal{T} \otimes \mathcal{P}$ *defined by* $\mathcal{A} = \langle \mathcal{Q}_\mathcal{A}, \mathcal{Q}^0_\mathcal{A}, \mathcal{F}, \Delta_\mathcal{A} \rangle$*, where*

- $\mathcal{Q}_\mathcal{A} = \mathcal{Q}_\mathcal{T} \times \mathcal{Q}_\mathcal{P}$ *is the set of states,*
- $\mathcal{Q}^0_\mathcal{A} = \mathcal{Q}^0_\mathcal{T} \times \mathcal{Q}^0_\mathcal{P}$ *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* $\exists \big\langle \langle s, q \rangle, F, \langle s', q' \rangle \big\rangle \in \Delta_\mathcal{A}$ *iff* $\exists \langle s, p, s' \rangle \in \Delta_\mathcal{T}, \exists \langle q, p', F, q' \rangle \in \Delta_\mathcal{P}$ *and* $p = p'$*.*

6.1 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. [6] 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 defined by* $H.e = \{g.e \mid g \in H\}$*.*
- *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* $\mathcal{T} = \langle \mathcal{Q}_\mathcal{T}, \mathcal{Q}^0_\mathcal{T}, \Sigma, \Delta_\mathcal{T} \rangle$ *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*

$$\begin{cases} s_1 \in \mathcal{Q}^0_{\mathcal{T}} \iff s_2 \in \mathcal{Q}^0_{\mathcal{T}} \\ p_2 = g.p_1 \text{ and} \\ \forall \langle s'_1, p'_1, d'_1\rangle \in \Delta_{\mathcal{T}}, s'_1 = d_1, \exists \langle s'_2, p'_2, d'_2\rangle \in \Delta_{\mathcal{T}}, s'_2 = d_2 \text{ and } p'_2 = g.p_2 \end{cases}$$

The action of the group on AP is extended to $\mathrm{Reach}(\mathcal{T})$ *by denoting $g.s$ the unique s_2 such that* $\forall g \in G, \forall \langle s_1, p_1, d_1\rangle \in \Delta_{\mathcal{T}}, s_1 = s, \exists d_2 \in \mathcal{Q}_{\mathcal{T}}, \langle s_2, g.p_1, d_2\rangle \in \Delta_{\mathcal{T}}$*. (The uniqueness is due to the constraint on $\Delta_{\mathcal{T}}$ in definition 7.)*
Because G is group, a consequence of this definition is that $G.\mathcal{Q}^0_{\mathcal{T}} = \mathcal{Q}^0_{\mathcal{T}}$.

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

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

- $\mathcal{Q}^0_{\mathcal{B}} = \{\langle G, G.s, q\rangle \mid s \in \mathcal{Q}^0_{\mathcal{T}}, q \in \mathcal{Q}^0_{\mathcal{P}}\}$
- $\mathcal{Q}_{\mathcal{B}} = \mathcal{Q}^0_{\mathcal{B}} \cup \mathcal{V}$ *where $\mathcal{V}$ is the set of tuples of the form $\langle H, O, q\rangle$ such that $H < G$, $O \subseteq$* $\mathrm{Reach}(\mathcal{T})$*, $q \in \mathcal{Q}_{\mathcal{P}}$, and $H.O = O$.*
- *$\Delta_{\mathcal{B}}$ is defined by construction as follows: $\big\langle \langle H, O, q\rangle, F, \langle H', O', q'\rangle\big\rangle \in \Delta_{\mathcal{B}}$ iff $\exists (s, s', p, p', F) \in O \times O' \times \Sigma \times \Sigma \times 2^{\mathcal{F}}$ such that $\langle s, p, s'\rangle \in \Delta_{\mathcal{T}}, \langle q, p', F, q'\rangle \in \Delta_{\mathcal{P}}$, and $p = p'$. Then $O' = (H \cap G_{p'}).s'$ and $H' \subseteq G_{O'}$.*

If $\mathcal{A} = \mathcal{T} \otimes \mathcal{P} = \langle \mathcal{Q}_{\mathcal{A}}, \mathcal{Q}^0_{\mathcal{A}}, \mathcal{F}, \Delta_{\mathcal{A}}\rangle$, any state $\langle H, O, q\rangle \in \mathcal{Q}_{\mathcal{B}}$ of $\mathcal{B}$ represents the set $\{\langle x, q'\rangle \in \mathcal{Q}_{\mathcal{A}} \mid x \in O \wedge q' = q\}$ of states of $\mathcal{A}$. Hence we can write $\langle x, q'\rangle \in \langle H, O, q\rangle$, and we will prove that $\mathcal{B}$ is a $\wp$-UTGBA over $\mathcal{A}$.

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 [1].

6.2 Correctness of the construction

We now prove that the symbolic synchronized product $\mathcal{B}$ is a $\wp$-UTGBA over $\mathcal{A}$, as per definition 5.

- Since any state $\langle H, O, q\rangle \in \mathcal{Q}_{\mathcal{B}}$ represents the set $\{\langle x, q'\rangle \in \mathcal{Q}_{\mathcal{A}} \mid x \in O \wedge q' = q\}$, we have $\mathcal{Q}_{\mathcal{B}} \subseteq 2^{\mathcal{Q}_{\mathcal{A}}}$ and condition (1) holds.
- Property (2) holds by definition of $\mathcal{B}$.
- Property (3) holds because:

$$\bigcup_{S \in \mathcal{Q}^0_{\mathcal{B}}} S = \{\langle x, q'\rangle \mid s \in \mathcal{Q}^0_{\mathcal{T}}, x \in G.s, q' \in \mathcal{Q}^0_{\mathcal{P}}\} = (G.Q^0_{\mathcal{T}}) \times \mathcal{Q}^0_{\mathcal{P}} = \mathcal{Q}^0_{\mathcal{T}} \times \mathcal{Q}^0_{\mathcal{P}} = \mathcal{Q}^0_{\mathcal{A}}$$

– Property (4) translates as follows:

$$\forall\big\langle\langle s,q\rangle,F,\langle s',q'\rangle\big\rangle \in \Delta_{\mathcal{A}},\ \forall\langle H,O,q\rangle \in \mathrm{Reach}(\mathcal{B}),\ \langle s,q\rangle \in \langle H,O,q\rangle \implies$$
$$\exists\langle H',O',q'\rangle \in \mathcal{Q}_{\mathcal{B}},\ \langle s',q'\rangle \in \langle H',O',q'\rangle,\ \big\langle\langle H,O,q\rangle,F,\langle H',O',q'\rangle\big\rangle \in \Delta_{\mathcal{B}}$$

Let $\big\langle\langle s,q\rangle,F,\langle s',q'\rangle\big\rangle \in \Delta_{\mathcal{A}}$. By the definition of $\mathcal{A}$, there exists two matching transitions $\langle s,p,s'\rangle \in \Delta_{\mathcal{T}}$, and $\langle q,p',F,q'\rangle \in \Delta_{\mathcal{P}}$ where $p = p'$. Let $\langle H,O,q\rangle \in \mathrm{Reach}(\mathcal{B})$ such that $\langle s,q\rangle \in \langle H,O,q\rangle$. By definition of $\Delta_{\mathcal{B}}$, there exists $\big\langle\langle H,O,q\rangle,F,\langle H',O',q'\rangle\big\rangle \in \Delta_{\mathcal{B}}$ where $O' = (H \cap G_{p'}).s'$ and $H' \subseteq G_{O'}$. And then $\langle s',q'\rangle \in \langle H',O',q'\rangle$.

– Finally property (5) translates as:

$$\forall\big\langle\langle H,O,q\rangle,F,\langle H',O',q'\rangle\big\rangle \in \Delta_{\mathcal{B}},\ \forall\langle s',q'\rangle \in \langle H',O',q'\rangle, \text{then}$$
$$\exists\langle s,q\rangle \in \langle H,O,q\rangle, \text{such that } \big\langle\langle s,q\rangle,F,\langle s',q'\rangle\big\rangle \in \Delta_{\mathcal{A}}$$

Let $\big\langle\langle H,O,q\rangle,F,\langle H',O',q'\rangle\big\rangle \in \Delta_{\mathcal{B}}$, and $\langle s',q'\rangle \in \langle H',O',q'\rangle$. By definition of $\Delta_{\mathcal{B}}$, we have $\exists\big\langle\langle x,q\rangle,F,\langle x',q'\rangle\big\rangle \in \Delta_{\mathcal{T}}$ such that $\langle x,q\rangle \in \langle H,O,q\rangle$ and $\langle x',q'\rangle \in \langle H',O',q'\rangle$. Since $\langle x',q'\rangle$ and $\langle s',q'\rangle$ belong to $\langle H',O',q'\rangle$, there exists $g \in H'$ such that $g.x' = s'$. Because $H' < G$ which is congruent with respect to the transition relation, we have $\big\langle\langle g.x,q\rangle,F,\langle g.x',q'\rangle\big\rangle = \big\langle\langle g.x,q\rangle,F,\langle s',q'\rangle\big\rangle \in \Delta_{\mathcal{A}}$.

6.3 Operations needed by the emptiness check

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 $\mathrm{Ref}(\langle H_1,O_1,q_1\rangle,H_2) = \{\langle H_1\cap H_2,O_i,q_1\rangle \mid i \in \mathbb{N}\}$ 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 as follows: $\langle H_1,O_1,q_1\rangle \subseteq \langle H_2,O_2,q_1\rangle$ iff $\mathrm{Ref}(\langle H_1,O_1,q_1\rangle,H_2) \subseteq \mathrm{Ref}(\langle H_2,O_2,q_1\rangle,H_1)$.

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.

$\mathrm{Decomp}(\mathcal{B},\langle R,F,T\rangle,D)$ 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: $\{T_i\}_i = \mathrm{Ref}(\langle H_1,O_1,q_1\rangle,H_2) \setminus \mathrm{Ref}(\langle H_2,O_2,q_1\rangle,H_1)$. (The algorithm is improved by grouping some of these T_is.)

			SP+TEC			SSP+TEC			SSP+NSIEC			SSP+IEC			SSP+AEC		
model	n		st.	tr.	T	st.	tr.	T	st.	tr.	T	st.	tr.	T	st.	tr.	T
WCS3	28	nonempty prod.	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	empty products	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.

6.4 Handling 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 the 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 $\mathcal{P}_C = \mathcal{C} \otimes \mathcal{P}_C$. 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. [6] 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.

7 Performance

The symbolic synchronized product of definition 12 has been implemented using the core of GreatSPN[1] [1], and the emptiness checks we presented are implemented in Spot[2]. Connecting the two tools allowed us to compare different techniques.

Table 1 presents some measurements on two parametrized models: WCS [1] and PO [8]. In both cases increasing the parameter increases the number of states of system. Each of these models was synchronized against 50 property automata: the table has been split to average the cases were the resulting product is empty separately from the cases were 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 much of the 50 cases were empty or not.

[1] `http://www.di.unito.it/~greatspn/`

[2] `http://spot.lip6.fr/`

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 state 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) reduce 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 others methods. It is worth noting that it yielded no false negatives in these experiments! On the PO model, the new emptiness check algorithms are not significantly better because the model offers little occasion for inclusion; still it can be seen that they do no incur any overhead.

8 Conclusion

In this paper, we presented two novel emptiness check algorithms dealing with automata whose states are sets, and exploiting inclusions between these sets. Because there exists multiple methods to build such "set automata", we tried not to tie these algorithms to any specific method by providing a set of formal constraints that the automaton must verified to apply the emptiness check algorithms. We are now considering how it could be applied to techniques like unfolding graphs [3], observation graphs [7].

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 detriment 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 [10] would improve the situation.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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. Conference paper selected for journal publication.

[6] 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.

[7] 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.

[8] 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.

[9] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, Lecture Notes in Computer Science. Springer-Verlag, Apr. 2005. To appear.

[10] 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.

[11] A. Valmari. The state explosion problem. In *Lectures on Petri Nets 1: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer-Verlag, 1998.

[12] 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.