Combining Explicit and Symbolic Approaches for Better On-the-Fly LTL Model Checking

Alexandre Duret-Lutz[1], Kais Klai[2], Denis Poitrenaud[3], and Yann Thierry-Mieg[3]

[1] LRDE, EPITA, Kremlin-Bicêtre, France.
[2] LIPN, Université Paris-Nord, Villetaneuse, France.
[3] LIP6/MoVe, Université Pierre & Marie Curie, Paris, France.

Abstract. We present two new hybrid techniques that replace the synchronized product used in the automata-theoretic approach for LTL model checking. The proposed products are explicit graphs of aggregates (symbolic sets of states) that can be interpreted as Büchi automata. These hybrid approaches allow on the one hand to use classical emptiness-check algorithms and build the graph on-the-fly, and on the other hand, to have a compact encoding of the state space thanks to the symbolic representation of the aggregates. The *Symbolic Observation Product* assumes a globally stuttering property (e.g., LTL$\setminus\mathbf{X}$) to aggregate states. The *Self-Loop Aggregation Product* does not require the property to be globally stuttering (i.e., it can tackle full LTL), but dynamically detects and exploits a form of stuttering where possible. Our experiments show that these two variants, while incomparable with each other, can outperform other existing approaches.

1 Introduction

Model checking for Linear-time Temporal Logic (LTL) is usually based on converting the property into a Büchi automaton, composing the automaton and the model (given as a Kripke structure), and finally checking the language emptiness of the composed system [25]. This verification process suffers from a well known state explosion problem [24]. Among the various techniques that have been suggested as improvement, we can distinguish two large families: explicit and symbolic approaches.

Explicit model checking approaches explore an explicit representation of the product graph. A common optimization builds the graph on-the-fly as required by the emptiness check algorithm: the construction stops as soon as a counterexample is found [5].

Another source of optimization is to take advantage of stuttering equivalence between paths in the Kripke structure when verifying a stuttering-invariant property [9]: this has been done either by ignoring some paths in the Kripke structure [14], or by representing the property using a *testing automaton* [13]. To our knowledge, all these solutions require dedicated algorithms to check the emptiness of the product graph.

Symbolic model checking tackles the state-explosion problem by representing the product automaton symbolically, usually by means of decision diagrams (a concise way to represent large sets or relations). Various symbolic algorithms exist to verify LTL using fixpoint computations (see [10, 22] for comparisons and [15] for the clarity of the presentation). As-is, these approaches do not mix well with stuttering invariant reductions or on-the-fly emptiness checks.

However explicit and symbolic approaches are not exclusive, some combinations have already been studied [2, 11, 21, 16] to get the best of both worlds. They are referred to as **hybrid approaches**.

Most of these approaches consist in replacing the KS by an explicit graph where each node contains sets of states of the KS (called aggregates throughout this paper), that is an abstraction of the KS preserving properties of the original KS. In [2] for instance, each aggregate contains states that share their atomic proposition values, and the successor aggregates contain direct successors of the previous aggregate, thus preserving LTL but not branching temporal properties. In [11] this idea is taken one step further in the context of stuttering invariant properties, and each aggregate now contains sets of consecutive states that share their atomic proposition values. In both of these approaches, an explicit product with the formula automaton is built and checked for emptiness, allowing to stop early (on-the-fly) if a witness trace is found.

The approach of [21] is a bit different, as it builds one aggregate for each state of the Büchi automata (usually few in number), and uses a partitioned symbolic transition relation to check for emptiness of the product, thus resorting to a symbolic emptiness-check (based on a symbolic SCC hull computation).

The hybrid approaches we define in this paper are based on explicit graphs of aggregates (symbolic sets of states) that can be interpreted as Büchi automata. With this combination, we can use classi-

cal emptiness-check algorithms and build the graph on-the-fly, moreover the symbolic representation of aggregates gives us a compact encoding of the state space along with efficient fixpoint algorithms.

The first technique we present extends the Symbolic Observation Graph (SOG) technique [11, 16] (which itself can be seen as a specialization of the work of Biere et al. [2] for stuttering-invariant properties). Given a property, only a subset of atomic propositions of the system need to be observed. The SOG approach aggregates consecutive states of the Kripke structure that share the same values for the observed atomic propositions. The SOG is an aggregated Kripke structure that is stuttering equivalent to the original Kripke structure. We combine this principle with an idea presented by Kokkarinen et al. [17] in the context of partial order reductions: as we progress in the Büchi automaton, the number of atomic propositions to observe diminishes and allows further aggregation. We call this new graph a *Symbolic Observation Product* (SOP), because it replaces the product between the Kripke structure and the Büchi automaton in the explicit approach.

The second technique we present also defines an aggregation graph which is a product: the *Self-Loop Aggregation Product* (SLAP). It uses a different aggregation criterion based on the study of the self-loops around the current state of the Büchi automaton. Roughly speaking, consecutive states of the system are aggregated when they are compatible with the labels of self-loops. Unlike the previous approach, SLAP is not limited to stuttering-invariant properties. It dynamically allows to stutter according to a boolean formula computed as the disjunction of the labels of self-loops of the automata.

This paper is organized as follows. Section 2 introduces our notations, presents the basic automata-theoretic approach and compares it to the (existing) SOG approach. Sections 3 and 4 define our two new hybrid approaches: SOP and SLAP. We explain how we implemented these approaches and evaluate them in Section 5.

2 Preliminaries

2.1 Boolean Formulas

Let AP be a set of (atomic) propositions, and let $\mathbb{B} = \{\bot, \top\}$ represent Boolean values. We denote $\mathbb{B}(\mathrm{AP})$ the set of all Boolean formulas over AP, i.e., formulas built inductively from the propositions AP, $\mathbb{B}$, and the connectives $\wedge$, $\vee$, and $\neg$. If $\mathrm{AP}' \subseteq \mathrm{AP}$, then we have $\mathbb{B}(\mathrm{AP}') \subseteq \mathbb{B}(\mathrm{AP})$ by construction. For any formula f, we will note $FV(f)$ (for Free Variables) the set of propositions that occurs in f, e.g., $FV(b \vee \neg a) = \{a, b\}$.

An assignment is a function $\rho : \mathrm{AP} \to \mathbb{B}$ that assigns a truth value to each proposition. We denote $\mathbb{B}^{\mathrm{AP}}$ the set of all assignments of AP. Given a formula $f \in \mathbb{B}(\mathrm{AP})$ and an assignment $\rho \in \mathbb{B}^{\mathrm{AP}}$, we denote $\rho(f)$ the evaluation of f under ρ.[4] In particular, we will write $\rho \models f$ iff ρ is a satisfying assignment for f, i.e., $\rho \models f \iff \rho(f) = \top$. The set $\mathbb{B}^{\star}(\mathrm{AP}) = \{f \in \mathbb{B}(\mathrm{AP}) \mid \exists \rho \in \mathbb{B}^{\mathrm{AP}}, \rho \models f\}$ contains all satisfiable formulas.

We will use assignments to label the states of the model we want to verify, and the propositional functions will be used as labels in the automaton representing the property to check. The intuition is that a behavior of the model (a sequence of assignments) will match the property if we can find a sequence of formulas in the automaton that are satisfied by the sequence of assignments.

We will write $\rho \stackrel{E}{=} \rho'$ iff $\rho_{|E} = \rho'_{|E}$, where $\rho_{|E}$ denotes the restriction of the function ρ to the domain E. This means that assignments ρ and ρ' match on the propositions E.

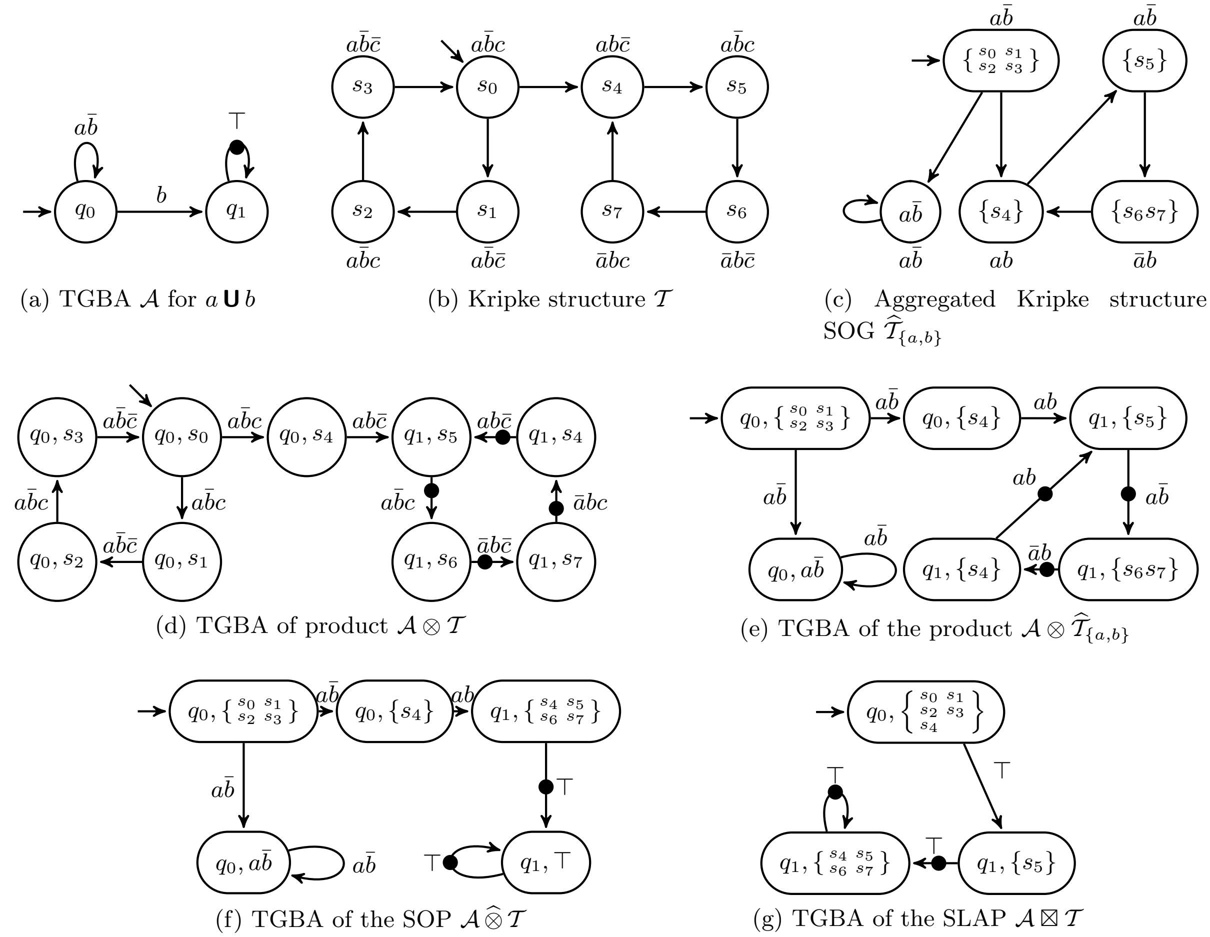
It is sometimes convenient to interpret an assignment ρ as a formula that is only true for this assignment. For instance the assignment $\{a \mapsto \top, b \mapsto \top, c \mapsto \bot\}$ can be interpreted as the formula $a \wedge b \wedge \neg c$. So we may use an assignment where a formula is expected, as if we were abusively assuming that $\mathbb{B}^{\mathrm{AP}} \subset \mathbb{B}(\mathrm{AP})$.

2.2 TGBA

A *Transition-based Generalized Büchi Automaton* (TGBA) is a Büchi automaton in which generalized acceptance conditions are expressed in term of transitions that must be visited infinitely often. The reason we use these automata is that they allow a more compact representation of properties than traditional Büchi automata (even generalized Büchi automata) [7] without making the emptiness check harder [6].

Definition 1 (TGBA). *A Transition-based Generalized Büchi Automata is a tuple* $A = \langle \mathrm{AP}, \mathcal{Q}, \mathcal{F}, \delta, q^0 \rangle$ *where*

[4] This can be defined straightforwardly as $\rho(f \wedge g) = \rho(f) \wedge \rho(g)$, $\rho(\neg f) = \neg \rho(f)$, etc.



(a) TGBA $\mathcal{A}$ for $a\,\mathbf{U}\,b$ (b) Kripke structure $\mathcal{T}$ (c) Aggregated Kripke structure SOG $\widehat{\mathcal{T}}_{\{a,b\}}$

(d) TGBA of product $\mathcal{A} \otimes \mathcal{T}$ (e) TGBA of the product $\mathcal{A} \otimes \widehat{\mathcal{T}}_{\{a,b\}}$

(f) TGBA of the SOP $\mathcal{A} \,\widehat{\otimes}\, \mathcal{T}$ (g) TGBA of the SLAP $\mathcal{A} \boxtimes \mathcal{T}$

Fig. 1: Examples

- AP *is a finite set of atomic propositions,*
- $\mathcal{Q}$ *is a finite set of states,*
- $\mathcal{F} \neq \emptyset$ *is a finite and non-empty set of acceptance conditions,*
- $\delta \subseteq \mathcal{Q} \times \mathbb{B}^\star(\mathrm{AP}) \times 2^{\mathcal{F}} \times \mathcal{Q}$ *is a transition relation. We will commonly denote* $q_1 \xrightarrow{f,ac} q_2$ *an element* $(q_1, f, ac, q_2) \in \delta$,
- $q^0 \in \mathcal{Q}$ *is the initial state.*

An execution (or a run) of A is an infinite sequence of transitions $\pi = (s_1, f_1, ac_1, d_1) \cdots (s_i, f_i, ac_i, d_i) \cdots \in \delta^\omega$ with $s_1 = q^0$ and $\forall i, d_i = s_{i+1}$. We shall simply denote it as $\pi = s_1 \xrightarrow{f_1,ac_1} s_2 \xrightarrow{f_2,ac_2} s_3 \cdots$. Such an execution is *accepting* iff it visits each acceptance condition infinitely often, i.e., if $\forall a \in \mathcal{F}, \forall i > 0, \exists j \geq i, a \in ac_j$. We denote $\mathrm{Acc}(A) \subseteq \delta^\omega$ the set of accepting executions of A.

A behavior of the model is an infinite sequence of assignments: $\rho_1\rho_2\rho_3 \cdots \in (\mathbb{B}^{\mathrm{AP}})^\omega$, while an execution of the automaton A is an infinite sequence of transitions labeled by Boolean formulas. The language of A, denoted $\mathcal{L}(A)$, is the set of behaviors compatible with an accepting execution of A: $\mathcal{L}(A) = \{\rho_1\rho_2 \cdots \in (\mathbb{B}^{\mathrm{AP}})^\omega \mid \exists s_1 \xrightarrow{f_1,ac_1} s_2 \xrightarrow{f_2,ac_2} \cdots \in \mathrm{Acc}(A) \text{ and } \forall i \geq 1, \rho_i \models f_i\}$

The non-emptiness constraint on $\mathcal{F}$ was introduced into definition 1 to avoid considering $\mathcal{F} = \emptyset$ as a separate case. If no acceptance conditions exist, one can be artificially added to some edges, ensuring that every cycle of the TGBA bears one on at least an edge. Simply adding this artificial acceptance condition to all edges might seriously hurt subsequent verification performance, as some emptiness-check algorithms are sensitive to the position of acceptance conditions.

Fig. 1a represents a TGBA for the LTL formula $a\,\mathbf{U}\,b$. The black dot on the self-loop $q_1 \xrightarrow{\top,\{\bullet\}} q_1$ denotes an acceptance conditions from $\mathcal{F} = \{\bullet\}$. The labels on edges $(a\bar{b}, b, \top)$ represent the Boolean expressions over $\mathrm{AP} = \{a, b\}$. There are many other TGBA in Fig. 1, that represent product constructions of this TGBA and the Kripke Structure of Fig. 1b.

A language $\mathcal{L}(A)$ is *stuttering-invariant* if any letter of a word can be repeated without affecting its membership to the language. In other words, $\mathcal{L}(A)$ is stuttering-invariant iff for any finite sequence $u \in (\mathbb{B}^{\mathrm{AP}})^*$, any assignment $\rho \in \mathbb{B}^{\mathrm{AP}}$, and any infinite sequence $v \in (\mathbb{B}^{\mathrm{AP}})^\omega$ we have $u\rho v \in \mathcal{L}(A) \iff u\rho\rho v \in \mathcal{L}(A)$.

Two sequences w_1 and w_2 are *stuttering equivalent* iff they are equal after removing all repeated letters. Two languages $\mathcal{L}(A)$ and $\mathcal{L}(B)$ are *stuttering equivalent* iff any word of $\mathcal{L}(A)$ is stuttering equivalent to a word of $\mathcal{L}(B)$ and vice versa.

LTL$\setminus$ **X** is the set of LTL formulas that do not use the **X** (next-time) operator. It is known that formulas in LTL$\setminus$ **X** describe stuttering-invariant properties (i.e., the language of the corresponding TGBA is stuttering-invariant), and that any stuttering-invariant property can be expressed in LTL$\setminus$ **X** [19]. The TGBA of Fig. 1 corresponds to the LTL formula $a\,\mathbf{U}\,b$ and consequently has a stuttering-invariant language.

2.3 Kripke Structure

For the sake of generality, we use *Kripke Structures* (KS for short) as a framework, since the formalism is well adapted to state-based semantics.

Definition 2 (Kripke structure). *A* Kripke structure *is a 4-tuple* $\mathcal{T} = \langle \mathrm{AP}, \Gamma, \lambda, \Delta, s_0\rangle$ *where:*
- AP *is a finite set of atomic propositions,*
- Γ *is a finite set of* states*,*
- $\lambda : \Gamma \to \mathbb{B}^{\mathrm{AP}}$ *is a state labeling function,*
- $\Delta \subseteq \Gamma \times \Gamma$ *is a* transition relation*. We will commonly denote* $s_1 \to s_2$ *the element* $(s_1, s_2) \in \Delta$*.*
- $s_0 \in \Gamma$ *is the* initial state*.*

Fig. 1b represents a Kripke structure over AP $= \{a, b, c\}$. The state graph of a system is typically represented by a KS whose labeling function gives the truth values of the atomic propositions for a given state of the system. The SOG construction of Fig. 1c also represents a KS; it is an aggregated abstraction built from Fig. 1b by observing only labels a and b.

We now define a synchronized product for a TGBA and a KS, such that the language of the resulting TGBA is the intersection of the languages of the two automata.

Definition 3 (Synchronized product of a TGBA and a Kripke structure). *Let* $\mathcal{A} = \langle \mathrm{AP}', \mathcal{Q}, \mathcal{F}, \delta, q^0\rangle$ *be a TGBA and* $\mathcal{T} = \langle \mathrm{AP}, \Gamma, \lambda, \Delta, s_0\rangle$ *be a Kripke structure over over* AP $\supseteq$ AP$'$*.*

The synchronized product *of* $\mathcal{A}$ *and* $\mathcal{T}$ *is the TGBA denoted by* $\mathcal{A} \otimes \mathcal{T} = \langle \mathrm{AP}, \mathcal{Q}_\otimes, \mathcal{F}, \delta_\otimes, q^0_\otimes\rangle$ *defined as:*
- $\mathcal{Q}_\otimes = \mathcal{Q} \times \Gamma$,
- $\delta_\otimes \subseteq \mathcal{Q}_\otimes \times \mathbb{B}^\star(\mathrm{AP}) \times 2^{\mathcal{F}} \times \mathcal{Q}_\otimes$ *where*

$$\delta_\otimes = \left\{ (q_1, s_1) \xrightarrow{f,ac} (q_2, s_2) \;\middle|\; \begin{array}{l} s_1 \to s_2 \in \Delta,\ \lambda(s_1) = f \textit{ and} \\ \exists g \in \mathbb{B}^\star(\mathrm{AP}) \textit{ s.t. } q_1 \xrightarrow{g,ac} q_2 \in \delta \textit{ and } \lambda(s_1) \models g \end{array} \right\}$$

- $q^0_\otimes = (q^0, s_0)$.

Fig. 1d represents such a product of the TGBA $a\,\mathbf{U}\,b$ of Fig.1a and the Kripke structure of Fig. 1b. State (s_0, q_0) is the initial state of the product. Since $\lambda(s_0) = a\bar{b}c$ we have $\lambda(s_0) \models a\bar{b}$, successors $\{s_1, s_4\}$ of s_0 in the KS will be synchronized through the edge $q_0 \xrightarrow{a\bar{b},\emptyset} q_0$ of the TGBA with q_0. In state (q_0, s_4) the product can progress through the $q_0 \xrightarrow{b,\emptyset} q_1$ edge of the TGBA, since $\lambda(s_4) = ab\bar{c} \models b$. Successor s_5 of s_4 in the KS is thus synchronized with q_1. The TGBA state q_1 now only requires states to verify $\top$ to validate the acceptance condition $\bullet$, so any cycle in the KS from s_5 will be accepted by the product. The resulting edge of the product bears the acceptance conditions contributed by the TGBA edge, and the atomic proposition Boolean formula label that comes from the KS. The size of the product in both nodes and edges is bounded by the product of the sizes of the TGBA and the KS.

The emptiness-check on a TGBA verifies if there exist a cycle that pass through an accepting edge (with the black dot). All of the TGBA product constructions in Fig. 1 agree in having a non-empty language, since language emptiness is the property these abstractions (SOP, SLAP) guarantee to preserve. These specialized synchronized products that are the main contribution of this paper will be discussed as they are defined.

2.4 Symbolic Observation Graph (SOG)

A symbolic observation graph over AP' is an abstraction of a KS over AP ($\mathrm{AP}' \subseteq \mathrm{AP}$) built to allow preservation of stuttering-invariant properties [11, 16].

It uses a symbolic data structure to represent sets of states of the KS that have been aggregated. The SOG is not a quotient graph, since the predicate we use to aggregate states is not an equivalence relation. Hence its worst case size in number of states (aggregates) is bounded by 2^{Γ}, while the number of successors of each aggregate is bounded by $2^{\mathrm{AP}'} - 1$.

However, in practice, particularly when the set of observed propositions AP' is small, which the case of a typical *LTL* formula, the SOG is much smaller than the underlying KS. Since the states in each aggregate are stored symbolically, the size of these aggregates is not necessarily the dominating factor in the overall complexity.

Notations For a set of states $a \subseteq \Gamma$ and a Boolean formula $f \in \mathbb{B}(\mathrm{AP})$, let us denote by $\mathrm{SuccF}(a, f) = \{s' \in \Gamma \mid \exists s \in a, s \to s' \in \Delta \wedge \lambda(s') \models f\}$, i.e., the set of the **Succ**essors states of a **F**iltered to keep only those satisfying f.

Furthermore, we denote by $\mathrm{ReachF}(a, f)$ the least subset of Γ satisfying:
- $a \subseteq \mathrm{ReachF}(a, f)$
- $\mathrm{SuccF}(\mathrm{ReachF}(a, f), f) \subseteq \mathrm{ReachF}(a, f)$

Definition 4 (Homogeneous aggregate). *Let $a \in 2^{\Gamma} \setminus \{\emptyset\}$ be a set of states. We say that a is a* homogeneous aggregate *with respect to a given subset of atomic propositions $\mathrm{AP}' \subseteq \mathrm{AP}$ iff $\forall s, s' \in a, \lambda(s) \stackrel{\mathrm{AP}'}{=} \lambda(s')$. Furthermore, for a homogeneous aggregate a w.r.t. $\mathrm{AP}' \subseteq \mathrm{AP}$, we write $\lambda_{\mathrm{AP}'}(a) = \lambda(s)_{|\mathrm{AP}'}$ for some state $s \in a$.*

A homogeneous aggregate a w.r.t. AP' is then a set of states that share the same values for atomic propositions in AP'. The associated label is the label of one of its states. Obviously, a homogeneous aggregate a w.r.t. AP' is homogeneous w.r.t. any $\mathrm{AP}'' \subseteq \mathrm{AP}'$.

Definition 5 (Symbolic Observation Graph). *Let $\mathcal{T} = \langle \mathrm{AP}, \Gamma, \lambda, \Delta, s_0 \rangle$ be a KS. A* symbolic observation graph *over $\mathrm{AP}' \subseteq \mathrm{AP}$ of $\mathcal{T}$ is the KS over AP' defined as $\mathcal{G}_{\mathrm{AP}'} = \langle \mathrm{AP}', S', \lambda', \Delta', a_0 \rangle$ satisfying :*

1. $S' = \Gamma' \cup \mathbb{B}^{\mathrm{AP}'}$ *with* $\Gamma' = \left\{ a \in 2^{\Gamma} \setminus \{\emptyset\} \;\middle|\; \begin{array}{l} a \text{ is homogeneous w.r.t. } \mathrm{AP}' \\ a = \mathrm{ReachF}(a, \lambda_{\mathrm{AP}'}(a)) \end{array} \right\}$

   Elements of Γ' are called aggregates *and elements of $\mathbb{B}^{\mathrm{AP}'}$ are* divergent states.
2. $\forall a \in S',\ \lambda'(a) = \begin{cases} \lambda_{\mathrm{AP}'}(a) & \text{if } a \in \Gamma' \\ a & \text{if } a \in \mathbb{B}^{\mathrm{AP}'} \end{cases}$
3. $$\begin{aligned} \Delta' = & \{a \to a' \in \Gamma' \times \Gamma' \mid a' = \mathrm{ReachF}(\mathrm{SuccF}(a, \lambda'(a')) \setminus a, \lambda'(a'))\} \\ & \cup \{a \to l \in \Gamma' \times \mathbb{B}^{\mathrm{AP}'} \mid a \text{ contains a cycle and } l = \lambda'(a)\} \\ & \cup \{l \to l \mid l \in \mathbb{B}^{\mathrm{AP}'}\} \end{aligned}$$
4. $a_0 = \mathrm{ReachF}(\{s_0\}, \lambda(s_0)_{|\mathrm{AP}'})$.

Following point 1 of the above Definition, the nodes of a SOG are of two kinds: (1) homogenous aggregates a satisfying $a = \mathrm{ReachF}(a, \lambda_{\mathrm{AP}'}(a))$, i.e., if a state $s \in a$ then each successor s' of s belongs to a as soon as $\lambda(s') \stackrel{\mathrm{AP}'}{=} \lambda(s)$, and, (2), divergent states, labeled with atomic propositions of AP'. The transition relation can be informally explained as follows: three kind of edges can connect the nodes of a SOG. If a and a' are two aggregates of Γ' then $a \to a'$ iff $\lambda'(a) \neq \lambda'(a')$ and each state $s' \in \Gamma$ satisfying $\lambda(s')_{|\mathrm{AP}'} = \lambda'(a')$ and $s \to s'$ is in a'. Given $a \in \Gamma'$ and l a divergent state then $a \to l$ iff a contains a cycle and is labeled with l. Finally each divergent state has a self-loop.

Fig. 1c represents the SOG built over the KS of Fig. 1b by disregarding the value of c. We can see in this product one divergent states labeled $a\bar{b}$ that represents the presence of a cycle in the states of its predecessor aggregate $\{s_0, s_1, s_2, s_3\}$. States are aggregated as long as they agree on the value of the subset of observed atomic propositions. The SOG is still a KS, that allows to check any stuttering-invariant property over the alphabet $\{a, b\}$. For instance, its product with the TGBA of $a\,\mathbf{U}\,b$ produces the TGBA of Fig. 1e. Both this abstraction and its product are smaller than their equivalents based on the plain Kripke structure of Fig. 1b.

Theorem 1 ([16]). *Given a Kripke Structure $\mathcal{T}$ defined on AP, then the SOG $\mathcal{G}_{\mathrm{AP}'}$ of $\mathcal{T}$ built over $\mathrm{AP}' \subseteq \mathrm{AP}$ preserves any stuttering-invariant property $\mathcal{A}$ on AP'. In other words: $\mathcal{L}(\mathcal{A} \otimes \mathcal{T}) \neq \emptyset \iff \mathcal{L}(\mathcal{A} \otimes \mathcal{G}_{\mathrm{AP}'}) \neq \emptyset$.*

3 Symbolic Observation Product (SOP)

A SOP is a dynamic extension of the SOG [11, 16]. Both approaches focus exclusively on stuttering-invariant properties. The SOP is a hybrid synchronized product between the TGBA of a stuttering-invariant property and a KS. In this product, the size of the observed alphabet AP' decreases as the construction progresses (an idea also presented by Kokkarinen et al. [17] in the context of partial order reductions), therefore allowing more aggregations.

3.1 Definition

Given a TGBA $A = \langle \mathrm{AP}, \mathcal{Q}, \mathcal{F}, \delta, q^0 \rangle$, let us define the alphabet $\mathrm{FV}(q)$ of a state $q \in \mathcal{Q}$ as the union of the atomic propositions which can be observed from q: i.e., $\mathrm{FV}(q) = \bigcup_{q_1 \xrightarrow{f,ac} q_2 \in \delta^\star(q)} \mathrm{FV}(f)$ where $\delta^\star(q)$ designates the set of transitions reachable from a state q. For instance, $\mathrm{FV}(q^0) = \mathrm{AP}$. It is clear that for any $q_1 \xrightarrow{f,ac} q_2 \in \delta$, we have $\mathrm{FV}(q_1) \supseteq \mathrm{FV}(q_2)$. The set of observed atomic propositions in a given state and its future reduces or at worse stays stable as we advance through the automaton.

Definition 6 (SOP of a TGBA and a KS). *Given a TGBA $\mathcal{A} = \langle \mathrm{AP}', \mathcal{Q}, \mathcal{F}, \delta, q^0 \rangle$ and a Kripke structure $\mathcal{T} = \langle \mathrm{AP}, \Gamma, \lambda, \Delta, s_0 \rangle$ over $\mathrm{AP} \supseteq \mathrm{AP}'$, the* Symbolic Observation Product *of $\mathcal{A}$ and $\mathcal{T}$ is the TGBA denoted $\mathcal{A} \widehat{\otimes} \mathcal{T} = \langle \mathrm{AP}', \mathcal{Q}_{\widehat{\otimes}}, \mathcal{F}, \delta_{\widehat{\otimes}}, q^0_{\widehat{\otimes}} \rangle$ where:*

– *$\mathcal{Q}_{\widehat{\otimes}} = \mathcal{Q}' \cup \mathcal{D}'$ where states of the automaton are synchronized with aggregates in $\mathcal{Q}'$ and with divergent states in $\mathcal{D}'$:*

$$\mathcal{Q}' = \left\{ (q,a) \in \mathcal{Q} \times (2^\Gamma \setminus \{\emptyset\}) \;\middle|\; \begin{array}{l} a \textit{ is homogeneous w.r.t. } \mathrm{FV}(q) \\ a = \mathrm{ReachF}(a, \lambda_{\mathrm{FV}(q)}(a)) \end{array} \right\}$$

$$\mathcal{D}' = \{ (q,l) \mid q \in \mathcal{Q} \textit{ and } l \in \mathbb{B}^{\mathrm{FV}(q)} \}$$

–
$$\delta_{\widehat{\otimes}} = \left\{ (q_1,a_1) \xrightarrow{l,ac} (q_2,a_2) \;\middle|\; \begin{array}{l} (q_1,a_1) \in \mathcal{Q}', (q_2,a_2) \in \mathcal{Q}', l = \lambda_{\mathrm{FV}(q_1)}(a_1), \\ \exists f \in \mathbb{B}(\mathrm{AP}) \textit{ s.t. } q_1 \xrightarrow{f,ac} q_2 \in \delta, \textit{ and } l \models f, \\ \exists l' \in \mathbb{B}^{\mathrm{FV}(q_2)} \textit{ s.t. } a_2 = \mathrm{ReachF}(\mathrm{SuccF}(a_1,l') \setminus a_1, l') \end{array} \right\}$$
$$\cup \left\{ (q_1,a) \xrightarrow{l_1,ac} (q_2,l_2) \;\middle|\; \begin{array}{l} (q_1,a) \in \mathcal{Q}', (q_2,l_2) \in \mathcal{D}', l_1 = \lambda_{\mathrm{FV}(q_1)}(a), \\ a \textit{ contains a cycle}, l_2 = l_{1|\mathrm{FV}(q_2)}, \\ \exists f \in \mathbb{B}(\mathrm{AP}) \textit{ s.t. } q_1 \xrightarrow{f,ac} q_2 \in \delta, \textit{ and } l_1 \models f \end{array} \right\}$$
$$\cup \left\{ (q_1,l_1) \xrightarrow{l_1,ac} (q_2,l_2) \;\middle|\; \begin{array}{l} (q_1,l_1) \in \mathcal{D}', (q_2,l_2) \in \mathcal{D}', l_2 = l_{1|\mathrm{FV}(q_2)}, \\ \exists f \in \mathbb{B}(\mathrm{AP}) \textit{ s.t. } q_1 \xrightarrow{f,ac} q_2 \in \delta, \textit{ and } l_1 \models f \end{array} \right\}$$

– $q^0_{\widehat{\otimes}} = (q_0, \mathrm{ReachF}(\{s_0\}, \lambda(s_0)_{|\mathrm{FV}(q_0)}))$

Let us explain the intuition behind the three terms of the transition relation. The first rule strongly resembles the SOG aggregation rule, except that the aggregate built from the successors of states in a_1 that bear the appropriate label only observes the atomic propositions in $\mathrm{FV}(q_2)$. This rule is the main ingredient that allows to observe less atomic propositions as the product progresses, and hence be more efficient. The next two rules define the cycle detection routine similar to the SOG, but taking into account the reduction of the set of atomic propositions to be observed.

Fig. 1f represents a SOP built from our example KS and the TGBA of $a \,\mathbf{U}\, b$. Because in state q_1 of the formula the observed alphabet is empty, the SOP aggregates the states of the cycle $\{s_4, s_5, s_6, s_7\}$ visible on the right of the product that uses the SOG (Fig. 1e). This cycle is then identified by the cycle detection rules, and visible as a divergent state in the SOP.

3.2 Proof of correctness

Our ultimate goal is to establish that, given a KS and a TGBA, the emptiness of the language of the corresponding SOP is equivalent to the emptiness of the language of the original synchronized product (see Theorem 2). This result is progressively demonstrated in the following. We proceed by construction i.e., if there exists an accepting run of the SOP then we build an accepting run of the original product and vice versa. In order to ease the proof of the first direction, we define a new synchronized product having a language stuttering equivalent to that of the original synchronized product (Lemma 1). Then, the desired accepting run is built in this new product (Lemma 2 and Lemma 3) and not in the original

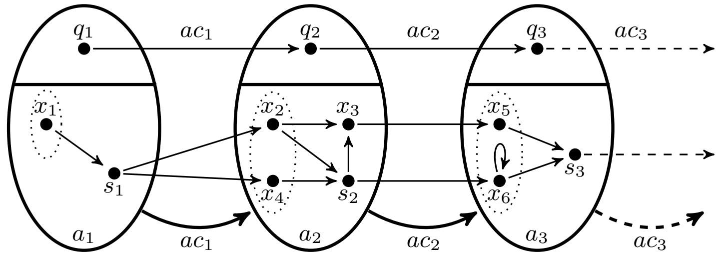


Fig. 2: A prefix $(q_1, a_1) \xrightarrow{ac_1} (q_2, a_2) \xrightarrow{ac_2} (q_2, a_2)$ of a run of some SOP $\mathcal{A}\widehat{\otimes}\mathcal{T}$ (with *different* $\mathcal{A}$ and $\mathcal{T}$ from Fig. 1) is shown using big ellipses and bended arrows. The straight lines also shows the underlying connections between the states $\{q_1, q_2, q_3, \ldots\}$ of the automaton $\mathcal{A}$ and between the states $\{s_1, s_2, \ldots, x_1, x_2 \ldots\}$ of the Kripke structure $\mathcal{T}$ that have been aggregated as $a_1, a_2, a_3 \ldots$ The acceptance conditions have been depicted as ac_i and the labels of the transitions have been omitted for clarity. The dotted ellipses show the set of input states ($In(a_1)$, $In(a_2)$, $In(a_3)$) as used in the proof of Lemma 2.

synchronized product. Hence, the desired result follows immediately (Corollary 1). Conversely, the proof of the second direction (each accepting run of the original product corresponds to an accepting run of the SOP) is not based on such a product but is nevertheless facilitated by two intermediate lemmas (Lemma 4 and Lemma 5).

Definition 7 (Stuttering synchronized product of a TGBA and a Kripke structure). *Let* $\mathcal{A} = \langle \mathrm{AP}, \mathcal{Q}, \mathcal{F}, \delta, q^0 \rangle$ *be a stuttering-invariant TGBA and* $\mathcal{T} = \langle \mathrm{AP}, \Gamma, \lambda, \Delta, s_0 \rangle$ *be a Kripke structure over the same atomic proposition set* AP*.*

The stuttering synchronized product *of A and $\mathcal{T}$ is the TGBA denoted by* $\mathcal{A}\tilde{\otimes}\mathcal{T} = \langle \mathrm{AP}, \mathcal{Q}_{\tilde{\otimes}}, \mathcal{F}, \delta_{\tilde{\otimes}}, q^0_{\tilde{\otimes}} \rangle$ *defined as:*

- $\mathcal{Q}_{\tilde{\otimes}} = \mathcal{Q} \times \Gamma$,
- $\delta_{\tilde{\otimes}} \subseteq \mathcal{Q}_{\times} \times \mathbb{B}^\star(\mathrm{AP}) \times 2^{\mathcal{F}} \times \mathcal{Q}_{\tilde{\otimes}}$ *where*

$$\delta_{\tilde{\otimes}} = \left\{ (q_1, s_1) \xrightarrow{f, ac} (q_2, s_2) \;\middle|\; \begin{array}{l} s_1 \rightarrow s_2 \in \Delta,\ \lambda(s_1) = f \textit{ and} \\ \exists g \in \mathbb{B}^\star(\mathrm{AP}) \textit{ s.t. } q_1 \xrightarrow{g, ac} q_2 \in \delta \textit{ and } \lambda(s_1) \models g \end{array} \right\}$$
$$\cup \left\{ (q_1, s_1) \xrightarrow{f, \emptyset} (q_1, s_2) \;\middle|\; \begin{array}{l} s_1 \rightarrow s_2 \in \Delta,\ \lambda(s_1) = f \textit{ and} \\ \lambda(s_1) \overset{\mathrm{FV}(q_1)}{=} \lambda(s_2) \end{array} \right\}$$

- $q^0_{\tilde{\otimes}} = (q^0, s_0)$.

Lemma 1. *Let $\mathcal{A}$ and $\mathcal{T}$ be defined as in Definition 7. We have* $\mathrm{Acc}(\mathcal{A}\tilde{\otimes}\mathcal{T}) \neq \emptyset \iff \mathrm{Acc}(\mathcal{A} \otimes \mathcal{T}) \neq \emptyset$.

Proof. By definition, the product $\mathcal{A}\tilde{\otimes}\mathcal{T}$ contains all the transitions of $\mathcal{A} \otimes \mathcal{T}$, and adds only stuttering transitions of the form $(q_i, s_i) \xrightarrow{f_i, ac_i} (q_{i+1}, s_{i+1})$ such that $q_i = q_{i+1}$, $f_i \overset{\mathrm{FV}q_i}{=} f_{i+1}$, and $ac_i = \emptyset$. Hence, the language $\mathcal{L}(\mathcal{A}\tilde{\otimes}\mathcal{T})$ is stuttering equivalent to the language $\mathcal{L}(\mathcal{A} \otimes \mathcal{T})$. Therefore $\mathcal{L}(\mathcal{A}\tilde{\otimes}\mathcal{T}) \neq \emptyset \iff \mathcal{L}(\mathcal{A} \otimes \mathcal{T}) \neq \emptyset$ and the lemma follows. □

Lemma 2. *Let $\mathcal{A}$ and $\mathcal{T}$ be defined as in Definition 7. Let $(q_1, a_1) \xrightarrow{ac} (q_2, a_2) \in \delta_{\widehat{\otimes}}$ be a transition of a SOP $\mathcal{A}\widehat{\otimes}\mathcal{T}$ such that $(q_2, a_2) \in \mathcal{Q}'$. For any state $s_2 \in a_2$ there exists at least one (possibly indirect) ancestor $s_1 \in a_1$ such that $(q_1, s_1) \xrightarrow{ac} (q_2, t_1) \rightarrow (q_2, t_2) \rightarrow \cdots (q_2, t_n) \rightarrow (q_2, s_2)$ is a sequence of the stuttering product $\mathcal{A}\tilde{\otimes}\mathcal{T}$ with $\forall i$, $t_i \in a_2$.*

For example, consider transition $(q_1, a_1) \xrightarrow{ac_1} (q_2, a_2)$ on Fig. 2, and some state in a_2, say s_2. Then $s_1 \in a_1$ is an indirect ancestor of s_2 s.t. $(q_1, s_1) \xrightarrow{ac_1} (q_2, x_2) \rightarrow (q_2, s_2)$.

Proof. Let us define the set of input states of the aggregate a_2 as $In(a_2) = \{s' \in a_2 \mid \exists s \in a_1, s \rightarrow s' \in \Delta\}$. This set cannot be empty since $(q_1, a_1) \xrightarrow{ac} (q_2, a_2)$.

Consider a state $s_2 \in a_2$. By construction of a_2, s_2 is reachable from some state in $t_1 \in In(a_2)$, so there exists a path $t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow s_2$ in the Kripke structure. Furthermore, all these states $t_1, t_2, \ldots, s_2$ are homogeneous w.r.t. $\mathrm{FV}(q_2)$, and the property is stuttering invariant, so there exists a path $(q_2, t_1) \rightarrow (q_2, t_2) \rightarrow \cdots \rightarrow (q_2, s_2)$ in the stuttering product $\mathcal{A}\tilde{\otimes}\mathcal{T}$.

Moreover, since $t_1 \in In(a_2)$, there exists a state s_1 in a_1 such that $(q_1, s_1) \rightarrow (q_2, t_1)$.

Consequently, the path $(q_1, s_1) \xrightarrow{ac} (q_2, t_1) \rightarrow (q_2, t_2) \rightarrow \cdots \rightarrow (q_2, s_2)$ satisfies the lemma. □

Lemma 3. *Let $\mathcal{A}$ and $\mathcal{T}$ be defined as in Definition 7. If there exists $\sigma \in \mathrm{Acc}(\mathcal{A}\widehat{\otimes}\mathcal{T})$ an infinite run accepted by the SOP, then there exists an accepting run $\pi \in \mathrm{Acc}(\mathcal{A}\tilde{\otimes}\mathcal{T})$ in the stuttering synchronized product.*

Proof. There are two cases to consider:

1. Either σ contains some divergent states, and then, by definition of $\delta_{\widehat{\otimes}}$, σ must necessarily have a finite prefix made of states from $\mathcal{Q}'$ followed by an infinite suffix made of states of $\mathcal{D}'$ and the last state of that prefix has an aggregate that contains a cycle.
   Let us denote $\sigma = (q_1, a_1) \xrightarrow{ac_1} (q_2, a_2) \xrightarrow{ac_2} \cdots (q_k, a_k) \xrightarrow{ac_k} (q_{k+1}, l_1) \xrightarrow{ac_{k+1}} (q_{k+2}, l_2) \cdots$ such an accepting run of $\mathcal{A}\widehat{\otimes}\mathcal{T}$.
   Let $s_k, s_{k+1}, \ldots, s_{k+n-1}$ be a cycle of a_k. Applying Lemma 2 from a_k to a_1, we can build a (possibly larger) sequence $\pi_p = (q_1, s_1) \xrightarrow{ac_1} \cdots (q_k, s_k)$ of transitions of $\mathcal{A}\tilde{\otimes}\mathcal{T}$. Since $s_1 \in a_1$, i.e., it belongs to the initial aggregate, it is accessible from s_0 by definition of $q^0_{\widehat{\otimes}}$. Therefore π_p can be prefixed by a sequence starting from (q^0, s_0); let π_i be this complete prefix, going from (q^0, s_0) to (q_k, s_k).
   Let us now complete π_i to an infinite sequence. Because all the states $s_k, s_{k+1}, \ldots, s_{k+n-1}$ are homogeneous w.r.t. $\mathrm{FV}(q_k)$, they are also homogeneous w.r.t. $\mathrm{FV}(q_{k+i})$ for any $i \geq 0$. By definition of the stuttering synchronized product, $\pi_s = (q_k, s_k) \xrightarrow{ac_k} \cdots (q_{k+n}, s_{k+n}) \xrightarrow{ac_{k+n}} \cdots (q_{k+i}, s_{k+(i \mod n)}) \xrightarrow{ac_{k+i}} \cdots$ is a path of $\mathcal{A}\tilde{\otimes}\mathcal{T}$.
   Consequently, the infinite sequence $\pi_i\pi_s$ starts from the initial state, and visits the same acceptance conditions of σ, thus $\pi_i\pi_s \in \mathrm{Acc}(\mathcal{A}\tilde{\otimes}\mathcal{T})$.
2. Or σ traverses only states from $\mathcal{Q}'$.
   Let us denote $\sigma = (q_1, a_1) \xrightarrow{ac_1} (q_2, a_2) \xrightarrow{ac_2} (q_3, a_3) \xrightarrow{ac_3} \cdots$ such an accepting run of $\mathcal{A}\widehat{\otimes}\mathcal{T}$. Let us build an infinite tree in which all nodes (except the root) are states of $\mathcal{A}\tilde{\otimes}\mathcal{T}$. Let us call $\top$ the root, at depth 0. The set of nodes at depth $n > 0$ is exactly the finite set of pairs $\{(q_n, s) \mid s \in a_n\} \subseteq \mathcal{Q} \times \Gamma$. The parent of any node at level 1 is $\top$. For any $i > 0$, the parent of a node (q_{i+1}, s') with $s' \in a_{i+1}$ is the node (q_i, s) for any state $s \in a_i$ such that (q_i, s) is a (possibly indirect) ancestor of (q_{i+1}, s') such that we observe ac_i on the path between these two states. We know such a state (q_i, s) exists because of Lemma 2. As a consequence of this parenting relation, every edge in this tree, except those leaving the root, correspond to a path between two states of $\mathcal{A}\tilde{\otimes}\mathcal{T}$.
   Because the set of nodes at depth $n > 0$ is finite, this infinite tree has finite branching. By König's Lemma it therefore contains an infinite branch. By following this branch and ignoring the first edge, we can construct a path of $\mathcal{A}\tilde{\otimes}\mathcal{T}$ that starts in (q_1, s_1) for some $s_1 \in a_1$, and that visits at least all the acceptance conditions ac_i of σ in the same order (and maybe more). To prove that this accepting path we have constructed actually occurs in a run of $\mathcal{A}\tilde{\otimes}\mathcal{T}$, it remains to show that (q_1, s_1) is a state that is accessible from the initial state of $\mathcal{A}\tilde{\otimes}\mathcal{T}$.
   Obviously $q_1 = q^0$ because $(q_1, a_1) = q^0_{\widehat{\otimes}}$ is the initial state of $\mathcal{A}\widehat{\otimes}\mathcal{T}$. Furthermore we have $s_1 \in a_1$, so by definition of $q^0_{\widehat{\otimes}}$, (q^0, s_1) must be reachable from (or equal to) (q^0, s_0) in $\mathcal{A}\tilde{\otimes}\mathcal{T}$.

□

Corollary 1. *If there exists $\sigma \in \mathrm{Acc}(\mathcal{A}\widehat{\otimes}\mathcal{T})$ an infinite run accepted by the SOP, then there exists an accepting run $\pi \in \mathrm{Acc}(\mathcal{A}\otimes\mathcal{T})$ in the synchronized product.*

Proof. Follows from Lemma 3 and Lemma 1. □

Lemma 4. *Let $\mathcal{A}$ and $\mathcal{T}$ be defined as in Definition 7. For a given n and a finite path $\pi_n = (q_0, s_0) \xrightarrow{f_0, ac_0} (q_1, s_1) \cdots \xrightarrow{f_{n-1}, ac_{n-1}} (q_n, s_n)$ of $\mathcal{A}\otimes\mathcal{T}$, there exists a finite path $\sigma_m = (q_{\varphi(0)}, a_0) \xrightarrow{ac_{\varphi(0)}} (q_{\varphi(1)}, a_1) \cdots \xrightarrow{ac_{\varphi(m-1)}} (q_{\varphi(m)}, a_m)$ of $\mathcal{A}\widehat{\otimes}\mathcal{T}$, with $m \leq n$, where*

$$\varphi(0) = \max\left\{ j \;\middle|\; \forall k \in \{0, \ldots, j\},\, q_k = q_j \wedge \lambda(s_k) \overset{\mathrm{FV}(q_j)}{=} \lambda(s_j) \right\}$$

$$\textit{for } i > 0 \quad \varphi(i) = \max\left\{ j \;\middle|\; \forall k \in \{\varphi(i-1)+1, \ldots, j\},\, q_k = q_j \wedge \lambda(s_k) \overset{\mathrm{FV}(q_j)}{=} \lambda(s_j) \right\}$$

and $\{s_0, \ldots, s_{\varphi(0)}\} \subseteq a_0$ and for $i > 0$, $\{s_{\varphi(i-1)+1}, \ldots, s_{\varphi(i)}\} \subseteq a_i$.

Proof. Let prove this by induction on the length of the finite path. The property is true for a path of length $n = 0$ by definition of $q^0_{\widehat{\otimes}}$. Now assume that the lemma is true for some length n and let us consider a path $p_{n+1} = (q_0, s_0) \xrightarrow{f_0, ac_0} (q_1, s_1) \cdots \xrightarrow{f_n, ac_n} (q_{n+1}, s_{n+1})$ of length $n+1$. By the induction hypothesis, we know that there exists a finite path $\sigma'_m = (q_{\varphi'(0)}, a_0) \xrightarrow{ac_{\varphi'(0)}} (q_{\varphi'(1)}, a_1) \cdots \xrightarrow{ac_{\varphi'(m-1)}} (q_{\varphi'(m)}, a_m)$ of $\mathcal{A} \widehat{\otimes} \mathcal{T}$, and a function φ' that correspond to the prefix of length n of π_{n+1}.

We consider the following two cases:

1. If $q_{n+1} = q_n$ and $\lambda(s_{n+1}) \overset{\mathrm{FV}(q_n)}{=} \lambda(s_n)$ then $s_{n+1} \in a_m$ by definition of $\delta_{\widehat{\otimes}}$. Therefore the path $\sigma_m = (q_{\varphi(0)}, a_0) \xrightarrow{ac_{\varphi(0)}} (q_{\varphi(1)}, a_1) \cdots \xrightarrow{ac_{\varphi(m-1)}} (q_{\varphi(m)}, a_m\})$ and the function φ defined as $\forall i < m$, $\varphi(i) = \varphi'(i)$ and $\varphi(m) = n + 1$, satisfy the lemma.
2. Otherwise, $q_{n+1} \neq q_n$ or $\lambda(s_{n+1}) \overset{\mathrm{FV}(q_n)}{\neq} \lambda(s_n)$. In that case, according the definition of $\delta_{\widehat{\otimes}}$, there exists an aggregate a_{m+1} such that $(q_n, a_m) \xrightarrow{ac_n} (q_{n+1}, a_{m+1})$ and $s_{n+1} \in a_{m+1}$.
   Since $n = \varphi'(m)$. We can define φ as $\forall i \leq m$, $\varphi(i) = \varphi'(i)$ and $\varphi(m + 1) = n + 1$, and build σ_{m+1} by extending σ'_m: $\sigma_{m+1} = (q_{\varphi(0)}, a_0) \xrightarrow{ac_{\varphi(0)}} (q_{\varphi(1)}, a_1) \cdots \xrightarrow{ac_{\varphi(m-1)}} (q_{\varphi(m)}, a_m) \xrightarrow{ac_n} (q_{\varphi(m+1)}, a_{m+1})$.
   This path satisfies the lemma.

□

Lemma 5. *Let $\mathcal{A}$ and $\mathcal{T}$ be defined as in Definition 7. If there exists an infinite path $\pi \in \mathrm{Acc}(\mathcal{A} \otimes \mathcal{T})$ accepting in $\mathcal{A} \otimes \mathcal{T}$. Then there exists an accepting path in $\mathcal{A} \widehat{\otimes} \mathcal{T}$ as well.*

Proof. $\mathcal{A} \otimes \mathcal{T}$ has a finite number of states, so if $\mathrm{Acc}(\mathcal{A} \otimes \mathcal{T}) \neq \emptyset$ then it contains at least one infinite path $\pi \in \mathrm{Acc}(\mathcal{A} \otimes \mathcal{T})$ that can be represented as a finite prefix followed by a finite cycle that is repeated infinitely often.

Let us denote this lasso-shaped path by $\pi = (q_0, s_0) \xrightarrow{ac_0} \cdots (q_k, s_k) \xrightarrow{ac_k} \cdots (q_n, s_n)$ with $(q_n, s_n) = (q_k, s_k)$.

Note that because $q_k, q_{k+1}, \ldots, q_n$ is a cycle in $\mathcal{A}$, we have $\mathrm{FV}(q_k) = \mathrm{FV}(q_{k+1}) = \ldots = \mathrm{FV}(q_{n-1})$.

We consider two possible cases:

1. If $\lambda(s_k) \overset{\mathrm{FV}(q_k)}{=} \lambda(s_{k+1}) \overset{\mathrm{FV}(q_k)}{=} \ldots \overset{\mathrm{FV}(q_k)}{=} \lambda(s_{n-1})$, these states are homogeneous and they form a cycle. We can apply Lemma 4 on prefix $(q_0, s_0) \xrightarrow{ac_0} \cdots (q_k, s_k)$ to build a path $\sigma_m = (q_{\varphi(0)}, a_0) \xrightarrow{ac_{\varphi(0)}} (q_{\varphi(1)}, a_1) \cdots \xrightarrow{ac_{\varphi(m-1)}} (q_{\varphi(m)}, a_m)$ such that $q_k = q_{\varphi(m)}$ and $s_k \in a_m$. Since the states $s_k, \ldots s_{n-1}$ are homogeneous, we also have $\{s_k, \ldots s_{n-1}\} \subseteq a_m$.
   Because a_m contains a cycle, there exist transitions $(q_k, a_m) \xrightarrow{ac_k} (q_{k+1}, l) \xrightarrow{ac_{k+1}} (q_{k+2}, l) \cdots \xrightarrow{ac_{n-1}} (q_n = q_k, l) \xrightarrow{ac_k} (q_{k+1}, l)$ according to $\delta_{\widehat{\otimes}}$.
   The infinite sequence $\sigma_m \xrightarrow{ac_k} \Big((q_{k+1}, l) \xrightarrow{ac_{k+1}} (q_{k+2}, l) \cdots \xrightarrow{ac_{n-1}} (q_k, l) \xrightarrow{ac_k}\Big)^\omega$ is accepting and satisfies the lemma.
2. Otherwise if the states $s_k, \ldots, s_{n-1}$ are not homogeneous w.r.t. $\mathrm{FV}(q_k)$, then we can apply Lemma 4 on the entire path π in order to build a path $\sigma_m = (q_{\varphi(0)}, a_0) \xrightarrow{ac_{\varphi(0)}} (q_{\varphi(1)}, a_1) \cdots \xrightarrow{ac_{\varphi(l-1)}} (q_{\varphi(l)}, a_l) \cdots \xrightarrow{ac_{\varphi(m-1)}} (q_{\varphi(m)}, a_m)$ such that $s_k \in a_l$ and $s_k = s_n \in a_m$.
   If $a_l = a_m$ then σ_m is lasso-shaped and preserves the acceptance conditions visited by π. Hence the lemma is verified.
   Unfortunately it is possible that the aggregate a_m and a_k are different because they were built from different predecessors. In that case, consider the lasso-shaped path, where the cycle has been unrolled 2^Γ times. Then applying Lemma 4 allows to build a path with σ_m that, among other states, traverses $2^\Gamma + 1$ states of the form $\{(q_k, a_l)\}_{l \in 0 \ldots 2^\Gamma}$ with all a_l containing the state $s_k = s_n$. Since an aggregate is a subset of Γ, at least two of these (q_k, a_l) are equal, and therefore we can construct a lasso-shaped accepting run that satisfies the lemma.

□

Theorem 2. *Let $\mathcal{A}$ be a TGBA, and $\mathcal{T}$ be a Kripke structure. The SOP of $\mathcal{A}$ and $\mathcal{T}$ accepts a run if and only if the synchronized product of these two structures accepts a run. In other word, we have $\mathrm{Acc}(\mathcal{A} \otimes \mathcal{T}) \neq \emptyset \iff \mathrm{Acc}(\mathcal{A} \widehat{\otimes} \mathcal{T}) \neq \emptyset$.*

Proof. $\Longleftarrow$ follows from Corollary 1; $\Longrightarrow$ follows from Lemma 5. □

4 Self-Loop Aggregation Product (SLAP)

This section presents a hybrid algorithm that is not restricted to stuttering-invariant properties. It is a specialized synchronized product that aggregates states of the KS as long as the TGBA state does not change, and no *new* acceptance conditions are visited.

4.1 Definition

The notion of self-loop aggregation is captured by $\mathrm{SF}(q, ac)$, the **S**elf-loop **F**ormulas (labeling edges $q \rightarrow q$) that are weaker in terms of visited acceptance conditions than ac.

When synchronizing with an edge of the property TGBA bearing ac leading to q, successive states of the Kripke will be aggregated as long as they model $\mathrm{SF}(q, ac)$. More formally, for a TGBA state q and a set of accepting condition $ac \subseteq \mathcal{F}$, let us define

$$\mathrm{SF}(q, ac) = \bigvee_{q \xrightarrow{f, ac'} q \in \delta \text{ s.t. } ac' \subseteq ac} f$$

Moreover, for $a \subseteq \Gamma$ and $f \in \mathbb{B}(\mathrm{AP})$, we define $\mathrm{FSucc}(a, f) = \{s' \in \Gamma \mid \exists s \in a,\ s \rightarrow s' \in \Delta \wedge \lambda(s) \models f\}$. That is, first **F**ilter a to only keep states satisfying f, then produce their **Succ**essors. The difference between SuccF and FSucc is whether the filter is applied on the source or destination states. Similarly to ReachF, we denote by $\mathrm{FReach}(a, f)$ the least subset of Γ satisfying both $a \subseteq \mathrm{FReach}(a, f)$ and $\mathrm{FSucc}(\mathrm{FReach}(a, f), f) \subseteq \mathrm{FReach}(a, f)$.

Definition 8 (SLAP of a TGBA and a KS). *Given a TGBA* $\mathcal{A} = \langle \mathrm{AP}', \mathcal{Q}, \mathcal{F}, \delta, q^0 \rangle$ *and a Kripke structure* $\mathcal{T} = \langle \mathrm{AP}, \Gamma, \lambda, \Delta, s_0 \rangle$ *over* $\mathrm{AP} \supseteq \mathrm{AP}'$ *, the* Self-Loop Aggregation Product *of* $\mathcal{A}$ *and* $\mathcal{T}$ *is the TGBA denoted* $\mathcal{A} \boxtimes \mathcal{T} = \langle \emptyset, \mathcal{Q}_\boxtimes, \mathcal{F}, \delta_\boxtimes, q^0_\boxtimes \rangle$ *where:*
- $\mathcal{Q}_\boxtimes = \mathcal{Q} \times (2^\Gamma \setminus \{\emptyset\})$
- $\delta_\boxtimes = \left\{ (q_1, a_1) \xrightarrow{\top, ac} (q_2, a_2) \;\middle|\; \begin{array}{l} \exists f \in \mathbb{B}(\mathrm{AP}') \text{ s.t. } q_1 \xrightarrow{f, ac} q_2 \in \delta, \\ \quad q_1 = q_2 \Rightarrow ac \neq \emptyset, \text{ and} \\ \quad a_2 = \mathrm{FReach}(\mathrm{FSucc}(a_1, f), \mathrm{SF}(q_2, ac)) \end{array} \right\}$
- $q^0_\boxtimes = (q^0, \mathrm{FReach}(\{s_0\}, \mathrm{SF}(q^0, \emptyset)))$

Note that because of the way the product is built, it is not obvious what Boolean formula should label the edges of the SLAP product. Since in fact this label is irrelevant when checking language emptiness, we label all arcs of the SLAP with $\top$ and simply denote $(q_1, a_1) \xrightarrow{ac} (q_2, a_2)$ any transition $(q_1, a_1) \xrightarrow{\top, ac} (q_2, a_2)$ of the SLAP.

Fig. 1g represents the SLAP built from our example KS, and the TGBA of $a \,\mathbf{U}\, b$. The initial state of the SLAP iteratively aggregates successors of states verifying $\mathrm{SF}(q^0, \emptyset) = a\bar{b}$. Then following the edge $q^0 \xrightarrow{b, \emptyset} q_1$, states are aggregated with condition $\mathrm{SF}(q_1, \emptyset) = \bot$. Hence q_1 is synchronized with successors of states in $\{s_0, s_1, s_2, s_3, s_4\}$ satisfying b (i.e., successors of $\{s_4\}$). Finally, when synchronizing with edge $q_1 \xrightarrow{\top, \bullet} q_1$, we have $\mathrm{SF}(q_1, \{\bullet\}) = \top$, hence all states of the cycle $\{s_4, s_5, s_6, s_7\}$ are added.

4.2 Proof of correctness

As for the SOP, we aim at demonstrating that, given a KS and a TGBA, the emptiness of the language of the corresponding SLAP is equivalent to the emptiness of the language of the original synchronized product. This result is progressively demonstrated in the following by means of several intermediate lemmas.

Lemma 6. *Let* $\mathcal{A}$ *and* $\mathcal{T}$ *be defined as in Definition 8. Let* $(q_1, a_1) \xrightarrow{ac} (q_2, a_2) \in \delta_\boxtimes$ *be a transition of the SLAP* $\mathcal{A} \boxtimes \mathcal{T}$*. For any state* $s_2 \in a_2$ *there exists at least one (possibly indirect) ancestor* $s_1 \in a_1$ *such that* $(q_1, s_1) \xrightarrow{ac} (q_2, t_1) \xrightarrow{\alpha_1} (q_2, t_2) \xrightarrow{\alpha_2} \cdots (q_2, t_n) \xrightarrow{\alpha_n} (q_2, s_2)$ *is a sequence of the synchronized product* $\mathcal{A} \otimes \mathcal{T}$ *with* $\forall i,\ t_i \in a_2$*, and* $\forall i,\ \alpha_i \subseteq ac$.

For example consider transition $(q_1, a_1) \xrightarrow{ac} (q_2, a_2)$ on Fig. 3, and some state in a_2, say s_2. Then $s_1 \in a_1$ is an indirect ancestor of s_2 s.t. $(q_1, s_1) \xrightarrow{ac} (q_2, x_2) \xrightarrow{\alpha_2} (q_2, s_2)$.

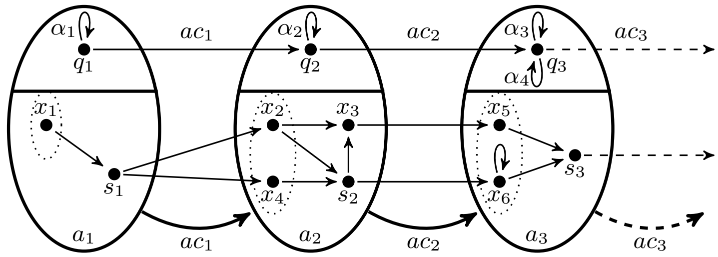


Fig. 3: A prefix $(q_1, a_1) \xrightarrow{ac_1} (q_2, a_2) \xrightarrow{ac_2} (q_2, a_2)$ of a run of some SLAP $\mathcal{A} \boxtimes \mathcal{T}$ (with *different* $\mathcal{A}$ and $\mathcal{T}$ from Fig. 1) is shown using big ellipses and bended arrows. The straight lines also shows the underlying connections between the states $\{q_1, q_2, q_3, \dots\}$ of the automaton $\mathcal{A}$ and between the states $\{s_1, s_2, \dots, x_1, x_2 \dots\}$ of the Kripke structure $\mathcal{T}$ that have been aggregated as $a_1, a_2, a_3 \dots$ The acceptance conditions have been depicted as ac_i or α_i and the labels of the transitions have been omitted for clarity. The dotted ellipses show the set of input states ($In(a_1)$, $In(a_2)$, $In(a_3)$) as used in the proof of Lemma 6.

Proof. Let us define the set of input states of the aggregate a_2 as $In(a_2) = \{s' \in a_2 \mid \exists s \in a_1, s \to s' \in \Delta\}$. This set cannot be empty since $(q_1, a_1) \xrightarrow{ac} (q_2, a_2)$.

Consider a state $s_2 \in a_2$. By construction of a_2, s_2 is reachable from some state in $t_1 \in In(a_2)$, so there exists a path $t_1 \to t_2 \to \dots \to s_2$ in the Kripke structure.

By definition of $\delta_{\boxtimes}$, if $t_1, t_2, \dots, s_2$ belong to a_2, the transitions between these states of $\mathcal{T}$ have been synchronized with self-loops $q_2 \xrightarrow{\alpha_i} q_2$ of $\mathcal{A}$ with $\alpha_i \subseteq ac$. Therefore the sequence $(q_2, t_1) \xrightarrow{\alpha_1} (q_2, t_2) \xrightarrow{\alpha_2} \cdots (q_2, t_n) \xrightarrow{\alpha_n} (q_2, s_2)$ is a sequence of the synchronized product $\mathcal{A} \otimes \mathcal{T}$.

Moreover, since $t_1 \in In(a_2)$, there exists a state s_1 in a_1 such that $(q_1, s_1) \xrightarrow{ac} (q_2, t_1)$.

Consequently, the path $(q_1, s_1) \xrightarrow{ac} (q_2, t_1) \xrightarrow{\alpha_1} (q_2, t_2) \xrightarrow{\alpha_2} \cdots (q_2, t_n) \xrightarrow{\alpha_n} (q_2, s_2)$ satisfies the lemma. $\square$

Lemma 7. *If there exists $\sigma \in \mathrm{Acc}(\mathcal{A} \boxtimes \mathcal{T})$ an infinite run accepted by the SLAP, then there exists an accepting run $\pi \in \mathrm{Acc}(\mathcal{A} \otimes \mathcal{T})$ in the classical product.*

Proof. Let us denote $\sigma = (q_1, a_1) \xrightarrow{ac_1} (q_2, a_2) \xrightarrow{ac_2} (q_3, a_3) \xrightarrow{ac_3} \cdots$ an accepting run of $\mathcal{A} \boxtimes \mathcal{T}$. Let us build an infinite tree in which all nodes (except the root) are states of $\mathcal{A} \otimes \mathcal{T}$. Let us call $\top$ the root, at depth 0. The set of nodes at depth $n > 0$ is exactly the finite set of pairs $\{(q_n, s) \mid s \in a_n\} \subseteq \mathcal{Q} \times \Gamma$.

The parent of any node at level 1 is $\top$. For any $i > 0$, the parent of a node (q_{i+1}, s') with $s' \in a_{i+1}$ is the node (q_i, s) for is any state $s \in a_i$ such that (q_i, s) is a (possibly indirect) ancestor of (q_{i+1}, s') such that we observe ac_i on the path between these two states. We know such a state (q_i, s) exists because of Lemma 6. As a consequence of this parenting relation, every edge in this tree, except those leaving the root, correspond to a path between two states of $\mathcal{A} \otimes \mathcal{T}$.

Because the set of nodes at depth $n > 0$ is finite, this infinite tree has finite branching. By König's Lemma it therefore contains an infinite branch. By following this branch and ignoring the first edge, we can construct a path of $\mathcal{A} \otimes \mathcal{T}$ that starts in (q_1, s_1) for some $s_1 \in a_1$, and that visits at least all the acceptance conditions ac_i of σ in the same order (and maybe more). To prove that this accepting path we have constructed actually occurs in a run of $\mathcal{A} \otimes \mathcal{T}$, it remains to show that (q_1, s_1) is a state that is accessible from the initial state of $\mathcal{A} \otimes \mathcal{T}$.

Obviously $q_1 = q^0$ because $(q_1, a_1) = q^0_{\boxtimes}$ is the initial state of $\mathcal{A} \boxtimes \mathcal{T}$. Furthermore we have $s_1 \in a_1$, so by definition of $q^0_{\boxtimes}$, (q^0, s_1) must be reachable from (or equal to) (q^0, s_0) in $\mathcal{A} \otimes \mathcal{T}$. $\square$

Lemma 8. *For a given n and a finite path $\pi_n = (q_0, s_0) \xrightarrow{f_0, ac_0} (q_1, s_1) \cdots \xrightarrow{f_{n-1}, ac_{n-1}} (q_n, s_n)$ of $\mathcal{A} \otimes \mathcal{T}$, there exists a finite path $\sigma_n = (q'_0, a_0) \xrightarrow{ac_{\varphi(0)}} (q'_1, a_1) \cdots \xrightarrow{ac_{\varphi(m-1)}} (q'_m, a_m)$ of $\mathcal{A} \boxtimes \mathcal{T}$, with $m \leq n$, $q_n = q'_m$, $s_n \in a_m$ and $\varphi_n : \{0, \dots, m-1\} \to \{0, \dots, n-1\}$ is a strictly increasing function such that $\forall j \, (\exists i, \varphi_n(i) = j \iff ac_i \neq \emptyset)$.*

Proof. Let us prove this lemma by induction on n. It is true if $n = 0$: Given $\pi_0 = (q_0, s_0)$, the path $\sigma_0 = (q'0, a_0) = q^0_{\boxtimes} = (q_0, \mathrm{FReach}(\{s_0\}, \{\lambda(s_0)\} \cap \lambda(q_0, \emptyset))$ satisfies the conditions (with φ being a null function).

Let us now assume that the lemma is true for $n+1$ assuming it is true for n. Given a path $\pi_{n+1} = \pi_n \xrightarrow{f_n, ac_n} (q_{n+1}, s_{n+1})$, we know by hypothesis that we have a matching σ_n for π_n. Let us consider how to extend σ_n into σ_{n+1} to handle the new transition $(q_n, s_n) \xrightarrow{f_n, ac_n} (q_{n+1}, s_{n+1})$ of π_{n+1}.

There are two cases to consider:

1. If $q_n = q_{n+1}$ and $acc_n = \emptyset$ and $\lambda(s_{n+1}) \models \mathrm{SF}(q_n, ac)$, then by definition of FSucc and SF the last state of σ_n, (q'_m, a_m) is such that $s_{n+1} \in a_m$ and $q'_m = q_n = q_{n+1}$. In that case $\sigma_{n+1} = \sigma_n$, and $\varphi_{n+1} = \varphi_n$.
2. If $q_n \neq q_{n+1}$ or $acc_n \neq \emptyset$ or $\lambda(s_{n+1}) \not\models \mathrm{SF}(q_n, ac)$, then because $\lambda(s_n) \models f_n$ and $s_n \rightarrow s_{n+1}$, by definition of $\delta_{\boxtimes}$ there exists $(q'_m, a_m) \xrightarrow{acc_n} (q'_{m+1}, a_{m+1})$ such that $s_{n+1} \in a_{m+1}$ and $q'_{m+1} = q_{n+1}$. In this case, we can define $\sigma_{n+1} = \sigma_n \xrightarrow{acc_n} (q'_{m+1}, a_{m+1})$ with $\forall i < n$, $\varphi_{n+1}(i) = \varphi_n(i)$ and $\varphi_{n+1}(n) = n$.

So by induction this lemma is true for all $n \in \mathbb{N}$. □

Lemma 9. *If there exists an infinite path $\pi \in \mathrm{Acc}(\mathcal{A} \otimes \mathcal{T})$ accepting in $\mathcal{A} \otimes \mathcal{T}$. Then there exists an accepting path in $\mathcal{A} \boxtimes \mathcal{T}$ as well.*

Proof. $\mathcal{A} \otimes \mathcal{T}$ has a finite number of states, so if $\mathrm{Acc}(\mathcal{A} \otimes \mathcal{T}) \neq \emptyset$ then it contains at least one infinite path $\pi \in \mathrm{Acc}(\mathcal{A} \otimes \mathcal{T})$ that can be represented as a finite prefix followed by a finite cycle that is repeated infinitely often.

Lemma 8 tells us that any prefix π_n of π corresponds to some prefix σ_n of a path in $\mathcal{A} \boxtimes \mathcal{T}$ in which the acceptance conditions of π_n occur in the same order. We have $|\sigma_n| \leq |\pi_n| = n$ but because π will visit all acceptance conditions infinitely often, and these transitions will all appear in σ_n (only transition without acceptance conditions can be omitted from $\delta_{\boxtimes}$), we can find some value of n for which $|\sigma_n|$ is arbitrary large. Because $|\sigma_n|$ can be made larger than the size of the SLAP, at some point this finite sequence will have to loop in a way that visits the acceptance conditions exactly in the same order as they appear in the cycle part of π. By repeating this cycle part of σ_n we can therefore construct an infinite path σ that is accepted by $\mathcal{A} \boxtimes \mathcal{T}$. □

Theorem 3. *Let $\mathcal{A}$ be a TGBA, and $\mathcal{T}$ be a Kripke structure. We have*

$$\mathrm{Acc}(\mathcal{A} \otimes \mathcal{T}) \neq \emptyset \iff \mathrm{Acc}(\mathcal{A} \boxtimes \mathcal{T}) \neq \emptyset$$

In other words, the SLAP of $\mathcal{A}$ and $\mathcal{T}$ accepts a run if and only if the synchronized product of these two structures accepts a run.

Proof. $\Longleftarrow$ follows from Lemma 7; $\Longrightarrow$ follows from Lemma 9. □

4.3 Mixing SLAP and Fully Symbolic Approaches

This section informally presents a variation on the SLAP algorithm, to use a fully symbolic algorithm in cases where the automaton state will no longer evolve.

The principle is the following: when the product has reached a state where the formula automaton state is terminal (i.e., it has itself as only successor), we proceed to use a fully symbolic search for an accepted path in the states of the current aggregate. This variant is called SLAP-FST, standing for Fully Symbolic search in Terminal states. Note that we suppose here that such a terminal state allows accepting runs, otherwise semantic simplifications would have removed the state from the TGBA.

In this variant, if q_1 is a terminal state, i.e., $\nexists q_1 \xrightarrow{f,ac} q_2 \in \delta$, with $q_1 \neq q_2$, a state (q_1, a_1) of the product has itself as sole successor through an arc labeled $(\top, \mathcal{F})$ if and only if a_1 admits a solution computed using a fully symbolic algorithm, or has no successors otherwise.

The fully symbolic search uses the self-loop arcs on the formula TGBA state to compute the appropriate transition relation(s), and takes into account possibly multiple acceptance conditions.

The rationale is that discovering this behavior when the aggregate is large, and particularly if there are long prefixes before reaching the SCC that bears all acceptance conditions, tends to create large SLAP structures in explicit size. The counterpart is that when no such solution exists, the fully symbolic SCC hull search may be quite costly.

In practice this variation on the SLAP was proposed after manually examining cases where SLAP performance was disappointing, typically because the SLAP was much larger in explicit size than the SOP. As discussed in the performance section, this variation is on average more effective than the basic SLAP algorithm.

5 Experimentations

In this section we present experimental results comparing several algorithms for hybrid or fully symbolic LTL model-checking. We use two different benchmark sets, one based on Petri net models using randomly generated LTL formulas, and one based on BEEM (Benchmarks for Explicit Model checkers [18]) models using meaningful LTL properties. We first present the context of these experiments, then focus on the results for Petri nets before detailing results for BEEM.

5.1 Implementation

We have implemented these new techniques (SOP, SLAP and SLAP-FST), the SOG [11, 16] as well as the classical fully symbolic algorithms (OWCTY [15] and EL [8]) and the hybrid approach of Biere et al. [2] (noted BCZ in the following) to allow comparisons. The software, available from `ddd.lip6.fr`, builds upon three existing components: Spot, SDD/ITS, and LTSmin.

Spot (`http://spot.lip6.fr`) is a model checking library [7]: it provides bricks to build your own model checker based on the automata theoretic approach using TGBAs. It has been evaluated as "one of the best explicit LTL model-checkers" [20]. Spot provides translation algorithms from LTL to TGBA, an implementation of a product between a Kripke structure and a TGBA (def. 3), and various emptiness-check algorithms to decide if the language of a TGBA is empty (among other things). The library uses abstract interfaces, so any object that can be wrapped to conform to the Kripke or TGBA interfaces can interoperate with the algorithms supplied by Spot.

SDD/ITS (`http://ddd.lip6.fr`) is a library representing Instantiable Transition Systems efficiently using Hierarchical Set Decision Diagrams [23]. ITS are essentially an abstract interface for (a variant of) labeled transition systems, and several input formalisms are supported (discrete time Petri nets, automata, and compositions thereof). SDD are a particular type of decision diagram that a) allow hierarchy in the state encoding, yielding smaller representations, b) support rewriting rules that allow the library to automatically [12] apply the symbolic saturation algorithm [4]. These features allow the SDD/ITS package to offer very competitive performance.

LTSmin[5] [3] is a tool allows to build a symbolic representation of the transition relation of a system using an explicit firing engine in background. The tool supports a wide range of input formalisms and explicit engines. For our experiments, we used LTSmin to build ETF files representing the transition relation. These files are then our input model, they were wrapped to conform to the ITS interface, thus allowing to apply our algorithms to any of the formalisms accepted by LTSmin or by ITS. We used the DVE variant of LTSmin to process the models from the BEEM benchmark. As noted by Blom et al. [3] this benchmark is not particularly favorable to symbolic approaches.

The fully symbolic OWCTY algorithm is implemented directly on top of the ITS interface; it uses an ITS representing the TGBA derived from the LTL formula by Spot composed (at the ITS formalism level) with the ITS representing the system. The resulting ITS is then analyzed using OWCTY with the forward transition relation.

The SOG is implemented as an object conforming to Spot's Kripke interface. It loads an ITS model, then builds the SOG on the fly, as required by the emptiness check of the product with the formula automaton.

Both SOP and SLAP are implemented as objects conforming to Spot's product interface. The SOP and the SLAP classes both take an ITS model and a TGBA (the formula automaton) as input parameters, and build their specialized product on the fly, driven by the emptiness-check algorithm.

5.2 Benchmark description

We use here classic scalable Petri net examples taken from Ciardo's benchmark set [4]: slotted ring, Kanban, flexible manufacturing system, and dining philosophers. Table 1 gives the size of each model.

The formulas considered include a selection of random LTL formulas, which were filtered to have a (basic TGBA/Kripke) product size of at least 1000 states. We also chose to have as many verified formulas (empty products) as violated formulas (non-empty products) to avoid favoring on-the-fly algorithms too much. To produce TGBA with several acceptance conditions, this benchmark includes 200 formulas for each model built from fairness assumptions of the form: $(\mathbf{G}\,\mathbf{F}\,p_1 \wedge \mathbf{G}\,\mathbf{F}\,p_2 \ldots) \implies \varphi$.

We also used 100 random formulas that use the next operator, and hence are not stuttering invariant (these where not used for SOG that does not support them).

[5] `http://fmt.cs.utwente.nl/tools/ltsmin`

model	state space	model	state space
fms5	2.9×10^6	kanban5	2.5×10^6
fms10	2.5×10^9	kanban10	1×10^9
philo10	4.6×10^6	ring6	5.8×10^5
philo50	2.3×10^{33}	ring7	6.2×10^6
philo100	5.1×10^{66}	ring10	8.3×10^9

Table 1: Number of reachable states in the selected models.

			OWCTY	EL	BCZ	SOG	SOP	SLAP	SLAP-FST
without **X**	empty	Win	103 (3%)	173 (5%)	53 (1%)	161 (4%)	735 (22%)	1256 (38%)	1703 (52%)
	(3229 cases)	Lose	260 (8%)	272 (8%)	2909 (90%)	481 (14%)	256 (7%)	246 (7%)	94 (2%)
		Fail	221 (6%)	253 (7%)	1786 (55%)	302 (9%)	219 (6%)	213 (6%)	87 (2%)
	non empty	Win	2 (0%)	10 (0%)	196 (4%)	513 (12%)	645 (15%)	2393 (59%)	1293 (31%)
	(4048 cases)	Lose	1846 (45%)	1378 (34%)	1924 (47%)	305 (7%)	318 (7%)	70 (1%)	40 (0%)
		Fail	804 (19%)	818 (20%)	1070 (26%)	263 (6%)	275 (6%)	69 (1%)	33 (0%)
with **X**	empty	Win	14 (1%)	13 (1%)	2 (0%)			810 (73%)	823 (74%)
	(1108 cases)	Lose	21 (1%)	21 (1%)	1081 (97%)			1 (0%)	1 (0%)
		Fail	0 (0%)	0 (0%)	355 (32%)			0 (0%)	0 (0%)
	non empty	Win	12 (0%)	7 (0%)	778 (23%)			1912 (57%)	1872 (55%)
	(3348 cases)	Lose	1697 (50%)	1627 (48%)	470 (14%)			54 (1%)	48 (1%)
		Fail	257 (7%)	272 (8%)	129 (3%)			29 (0%)	29 (0%)

Table 2: On all experiments (grouped with respect to the existence of a counterexample and the use of a **X** operator in the LTL formula), we count the number of cases a specific method has (Win) the best time or (Lose) it has either run out of time or it has the worst time amongst successful methods. The Fail line shows how much of the Lost cases were timeouts. The sum of a line may exceed 100% if several methods are equally placed.

We killed any process that exceeded 120 seconds of runtime, and set the garbage collection threshold at 1.3GB. Cases where all considered methods performed under 0.1s were filtered out from the results presented here: theses trivial cases represent only 4.2% of the entire benchmark, and were too fast too be allow any pertinent comparison.

5.3 General comparison

Table 2 gives a synthetic overview of the results presented hereafter. SLAP or SLAP-FST are the fastest methods in over half of all cases, and they are rarely the slowest. Furthermore, they have the least failure rate. This table also shows that BCZ has the highest failure rate and that the fully symbolic algorithms (OWCTY, EL) have trouble with non-empty products.

Table 2 presents only the best and the worst methods. While Fig. 4 and 5 allow to compare the different methods in a finer manner.

For each experiment (model/formula pair) we first collect the maximum time reached by a technique that did not fail, then compute for the other approaches what percentage of this maximum was used. The vertical segments visible at 100% thus show the number of runs for which this technique was the worst of those that did not fail. Any failures are plotted arbitrarily at 120%. This gives us a set of values between 0% and 120% for which we plot the cumulative distribution function. For instance, if a curve goes through the (20%,2000) point, it means that for this technique, 2000 experiments took at most 20% of the time taken by the worst technique for the same experiments.

The behavior at 120% represents the "Fail" line of previous table, while the behavior at 100% represents the difference between the "Slow" and "Fail" lines ("Slow" methods include methods that failed).

The left plot of Fig 4 for the non-empty cases shows that the on-the-fly mechanism allows all hybrid algorithms (SLAP, SLAP-FST, SOG, SOP, BCZ) to outperform the symbolic ones (OWCTY, EL). However as seen previously, BCZ still fails more often than other methods. The SLAP and SLAP-FST method take less than 10% of the time of the slowest method in 80% of the cases. On left of Fig 5, the same effect is visible, although BCZ actually has less failures than the fully symbolic algorithms.

The right plots for the empty cases show that fully symbolic algorithm behave relatively far better (all methods have to explore the full product anyway). BCZ spends too much time exploring enormous products, and timeouts.

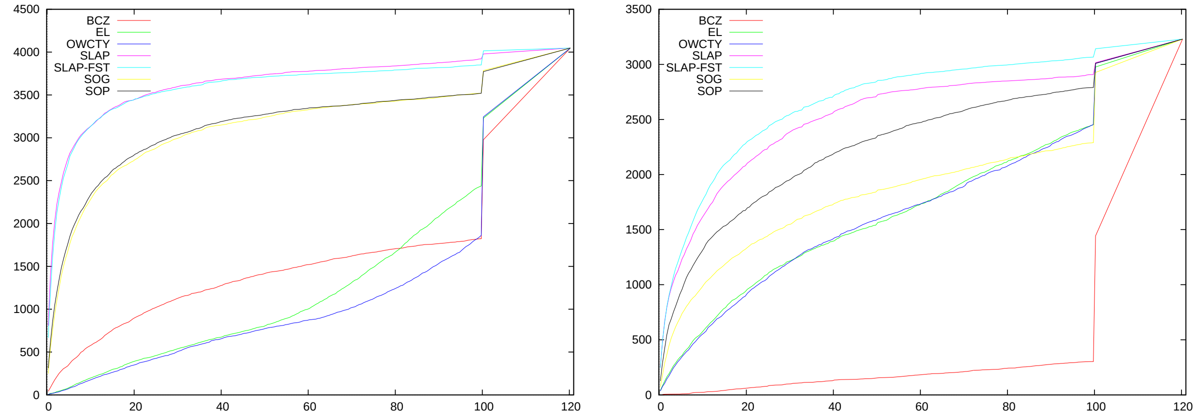


Fig. 4: Cumulative plots comparing the time of all methods on stuttering invariant properties only. Non-empty products are shown on the left, and empty products on the right.

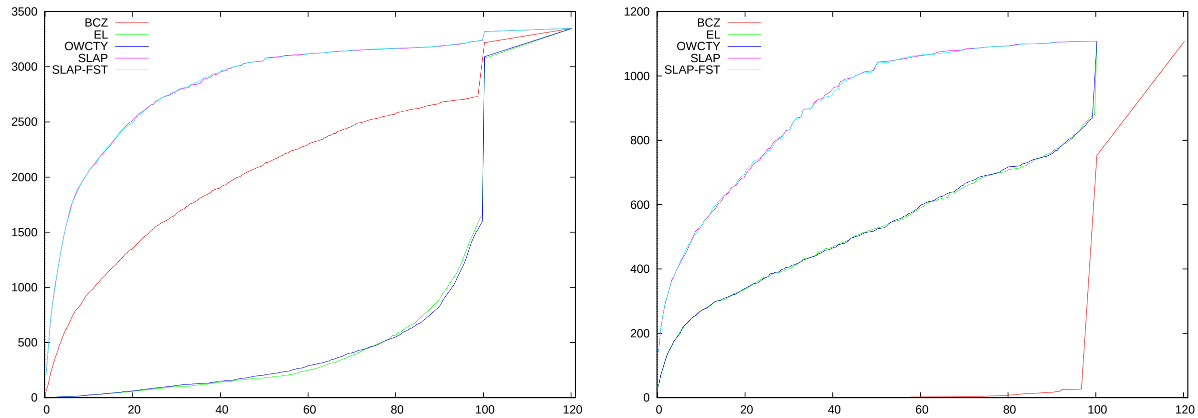


Fig. 5: Cumulative plots comparing the time of all methods (except SOG and SOP) for non stuttering invariant properties. Non-empty products are shown on the left, and empty products on the right.

For stuttering-invariant properties SLAP-FST and SLAP have similar performance, with a slight edge for SLAP-FST when the product is empty; however on Fig 5 SLAP and SLAP-FST are not significantly different.

EL appears slightly superior to OWCTY in the non-empty case, while they have similar performances in the empty case. This is mostly discernible on stuttering-invariant properties.

SOG and SOP show good results when there is a counterexample, and they perform better than BCZ in most cases. However SOG and SOP only support stuttering-invariant properties. As shown in Fig 5 BCZ is a good alternative to fully symbolic algorithms in presence of a counter-example; it is however systematically outperformed by the new algorithms we propose in this paper.

5.4 SLAP versus SLAP-FST

To study the differences between SLAP and SLAP-FST consider the scatter plots from Fig. 6. The performances are presented using a logarithmic scale. Each point represents an experiment, i.e., a model and formula pair. We plot experiments that failed (due to timeout) as if they had taken 360 seconds, so they are clearly separated from experiments that didn't fail (by the wide white band).

In these plots we have 11733 experiments, of which 132 proved too hard to solve for either algorithm within the time limit. Overall SLAP algorithm solved 17 problems that SLAP-FST did not, and SLAP-FST solved 179 instances that SLAP did not. SLAP was at least a hundred times slower than SLAP-FST in 5 cases, ten times slower in 50 cases, and twice as slow in 164 cases. SLAP-FST was one hundred times slower in 3 cases, ten times slower in 34 cases, and twice as slow in 396 cases.

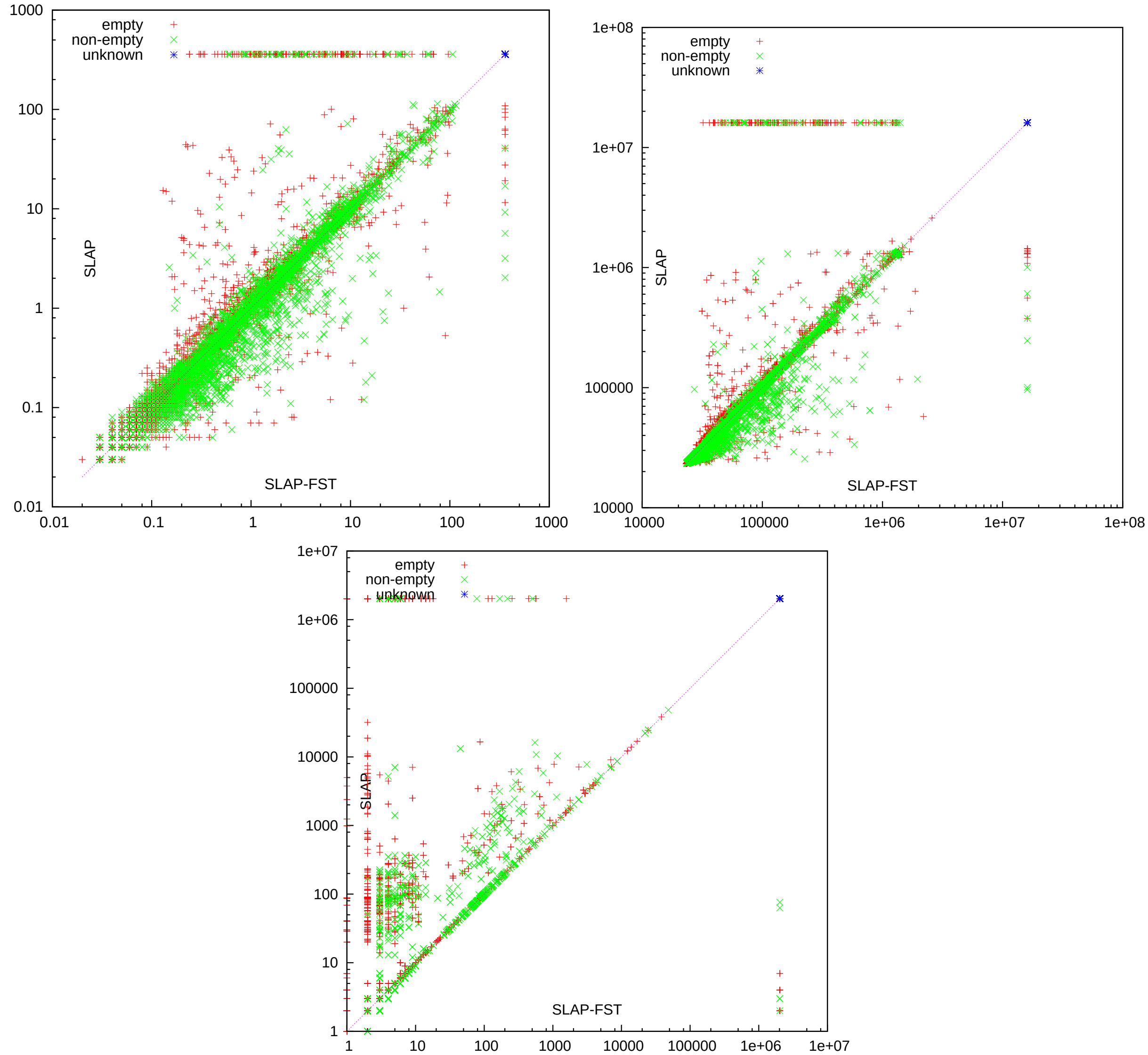


Fig. 6: Comparison of SLAP-FST against SLAP. Top left: time (in seconds); top right: memory (in kilobytes); bottom: product size (in states).

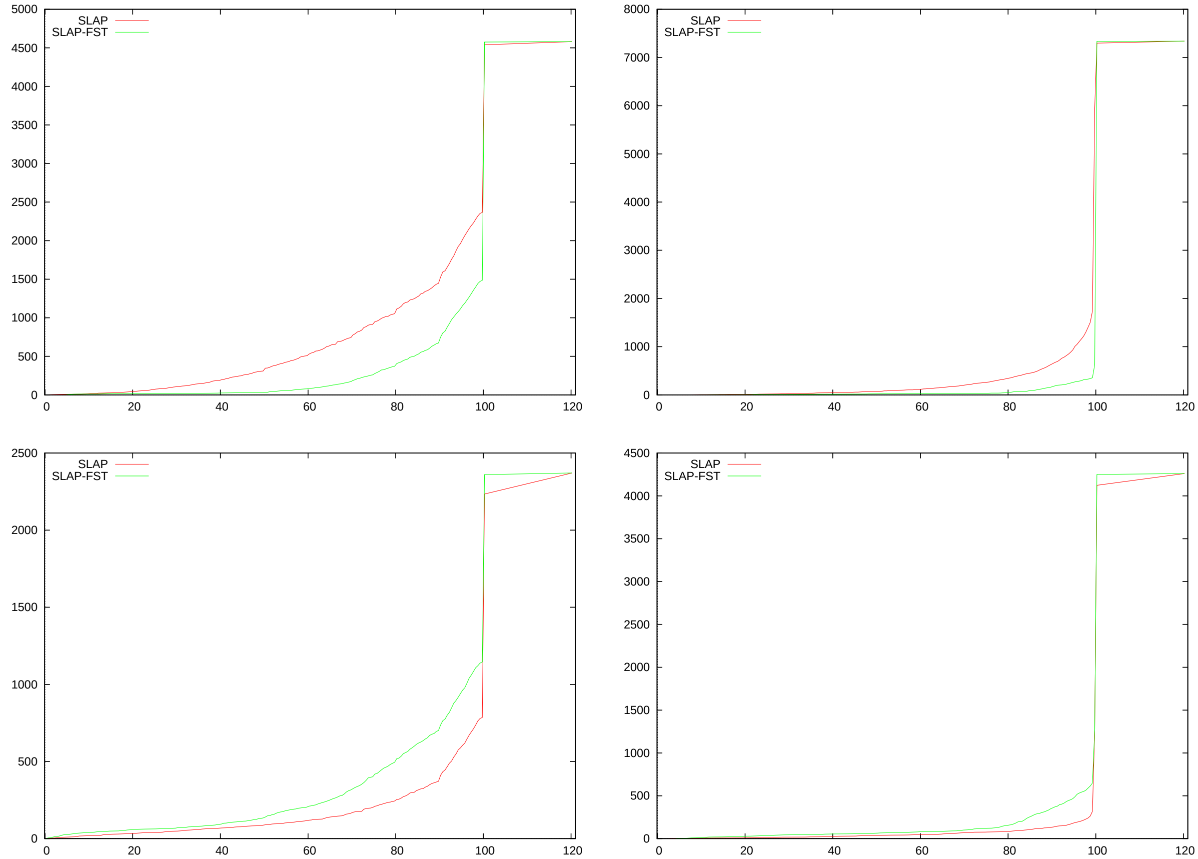


Fig. 7: Compared cumulative performances in time (left) and memory (right) of SLAP variants. Top for non-empty products, bottom for empty ones.

SLAP is on the average faster and consume less memory than SLAP-FST for non-empty products, but fails more often. SLAP-FST is better overall for empty products. Indeed the explicit product size of SLAP-FST is always smaller than that of SLAP, and often by several orders of magnitude. In some cases the SLAP degenerates to a state-space proportional to size of the explicit product while the SLAP-FST is able to keep the symbolic advantage.

The cumulative plots of Fig. 7 make this advantage even more visible. Indeed at the cost of slight memory overhead, and a more significant overall time overhead (when counter-examples are present), SLAP-FST produces much smaller explicit structures (hence wins significantly for empty products).

5.5 SLAP-FST versus other techniques

In Fig. 8 and 9 we compare SLAP-FST against the other methods, using the same kind of logarithmic scatter plots. These plots only use stuttering invariant properties so they can be more easily compared. Unsurprisingly, the only methods that appear competitive are SOG and SOP; but to the advantage of SLAP-FST, SOG and SOP are not able to handle non stuttering-invariant properties.

5.6 Fully symbolic algorithms: EL vs. OWCTY

These two algorithms are worth comparing because they differ only is in the way that the acceptance conditions are alternated throughout the fixpoint computation.

In Fig. 10, we have 11738 experiments, of which 1225 proved too hard to solve for either algorithm within the time limit. Overall EL algorithm solved 58 problems that OWCTY did not, and OWCTY solved 119 instances that EL did not. EL was at least ten times slower than OWCTY in 4 cases, and at least twice as slow in 141 cases, whereas OWCTY was twice as slow in 91 cases.

Overall these plots show very little perceptible difference for non-empty products and seem to slightly favor OWCTY for empty products. Given the overall aspect of these plots that do not stray much from the diagonal, we can state that both algorithms have comparable empirical complexities on this benchmark.

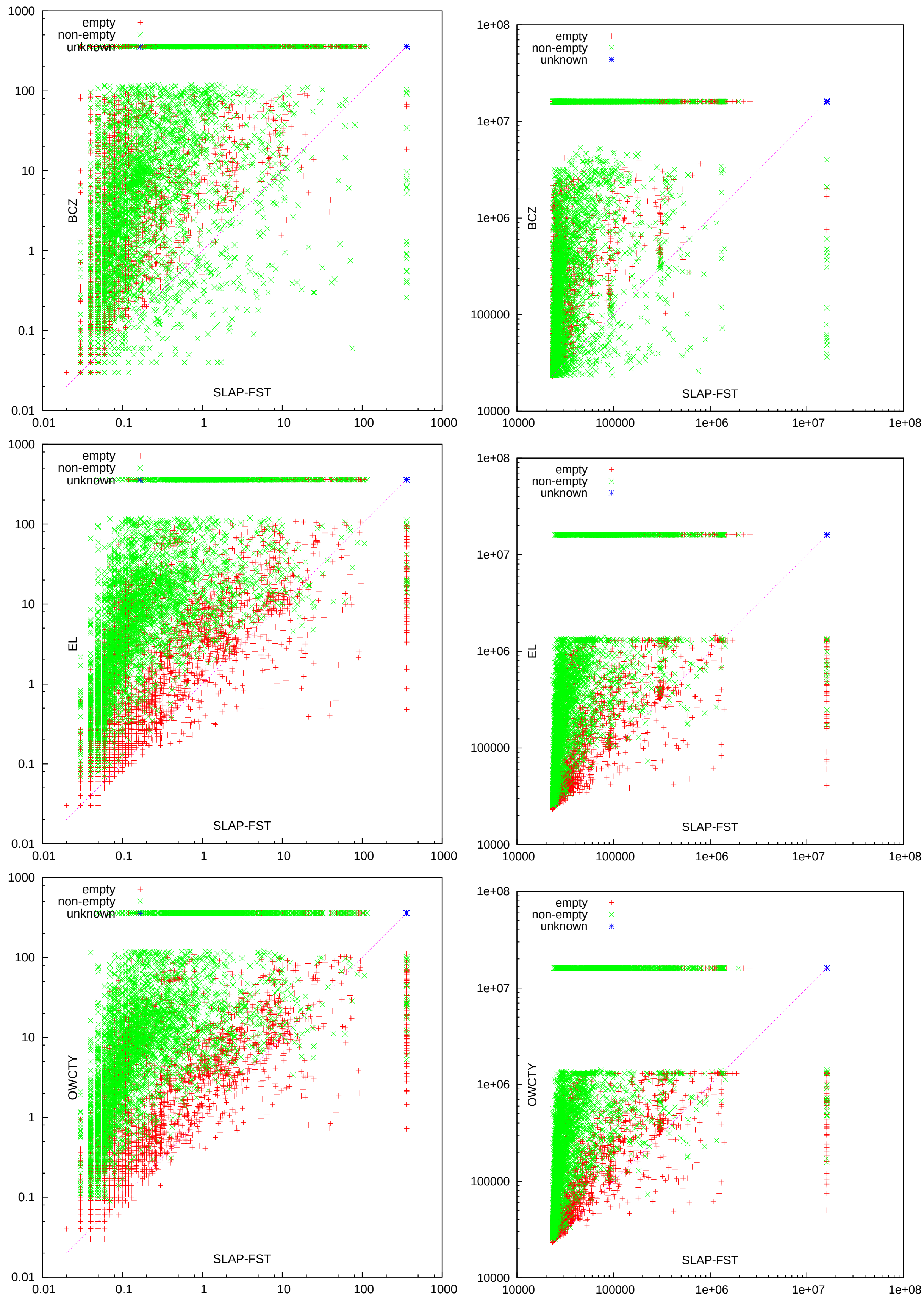


Fig. 8: Comparison of SLAP-FST against BCZ, EL, and OWCTY in time (left) and memory (right).

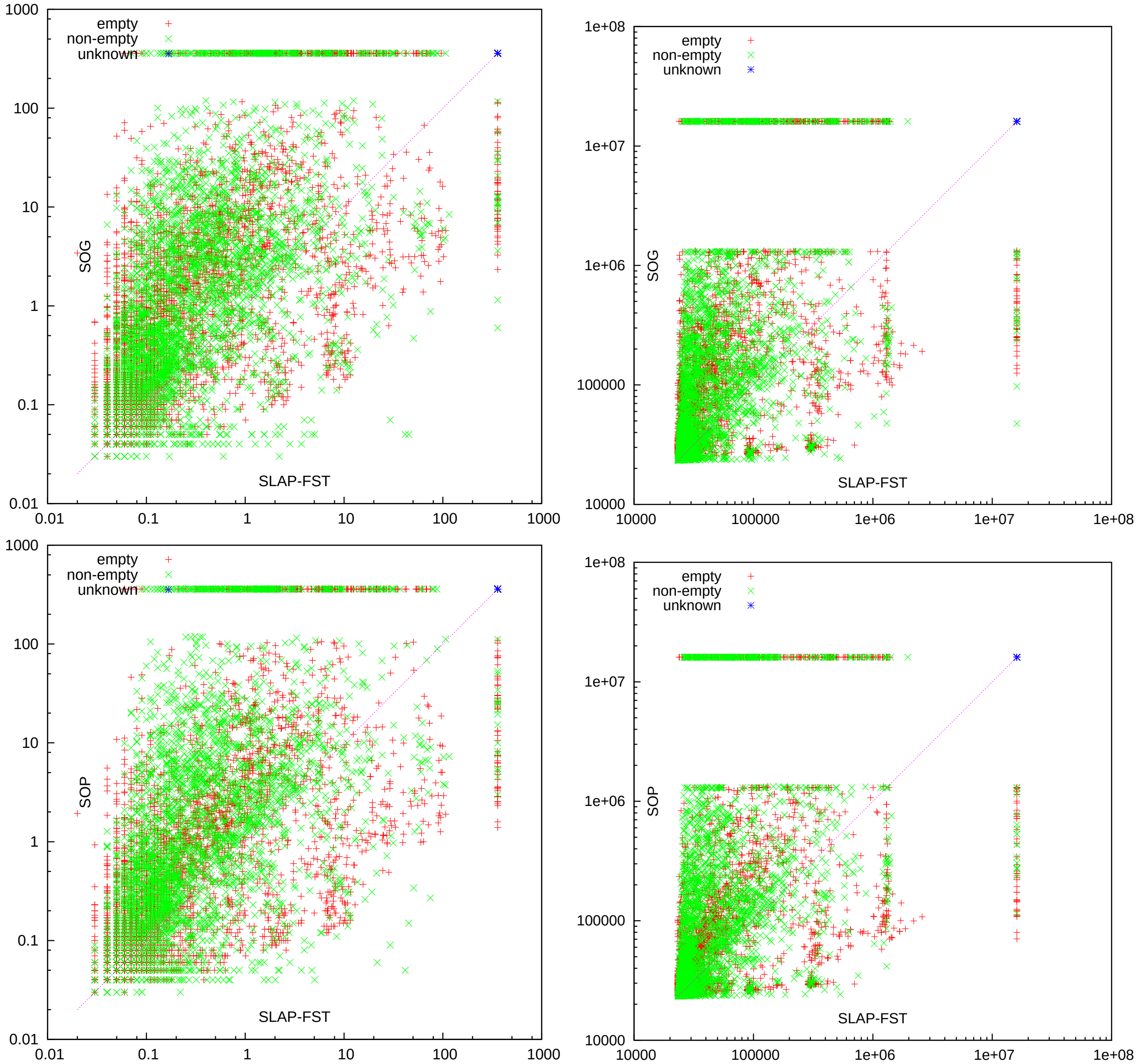


Fig. 9: Comparison of SLAP-FST against SOG and SOP in time (left) and memory (right).

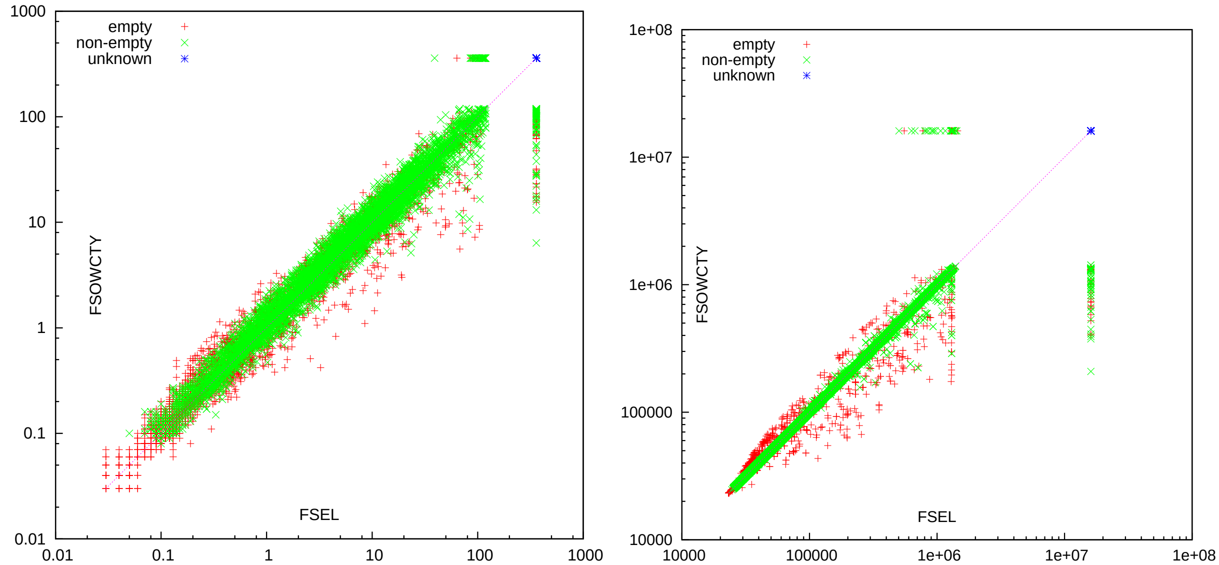


Fig. 10: Performances in time (left, in seconds) and memory (right, in kilobytes) of fully symbolic algorithms for 11738 experiments.

Although the scatter plot does not highlight this fact very blatantly, the density of experiments where EL outperformed OWCTY is actually quite high. The cumulative plots from Fig. 11 make this more visible. They also show that the difference between the two algorithms only very rarely exceed a ratio of 2.

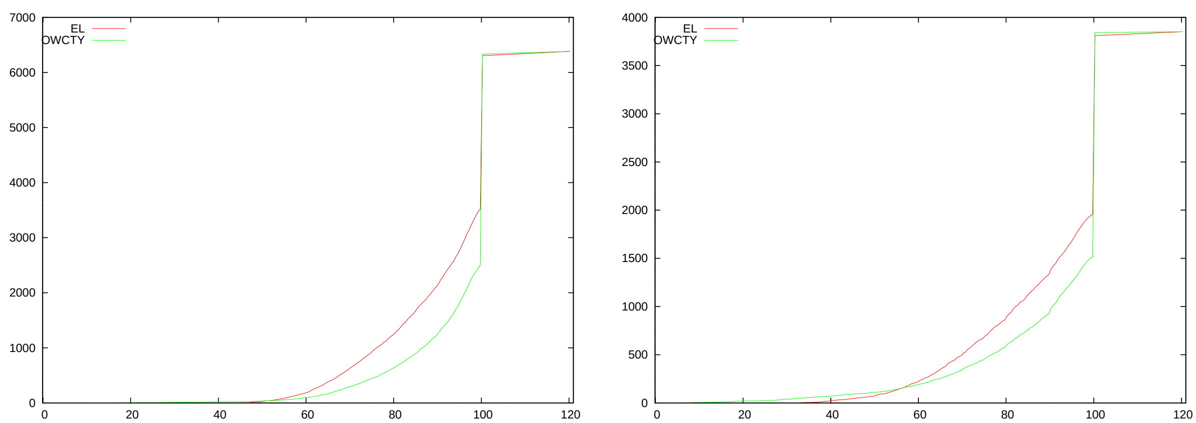


Fig. 11: Compared cumulative performances in time of fully symbolic algorithms with (left) or without counterexamples (right).

5.7 Hybrid stuttering invariant algorithms: SOG vs. SOP

We compare here the SOP and SOG algorithms.

In these plots (Fig. 12), we have 7277 experiments, of which 399 proved too hard to solve for either algorithm within the time limit. Overall SOG algorithm solved 95 problems that SOP did not, and SOP solved 166 instances that SOG did not. SOG was at least a hundred times slower than SOP in 11 cases, ten times slower in 132 cases, and twice as slow in 1326 cases. A contrario, SOP was ten times slower than SOG in 13 cases, and twice as slow in 280 cases.

Overall these plots show that SOP significantly outperforms SOG in many problem instances, particularly when the full product needs to be built (i.e., it is empty so on-the-fly mechanism does not come into play).

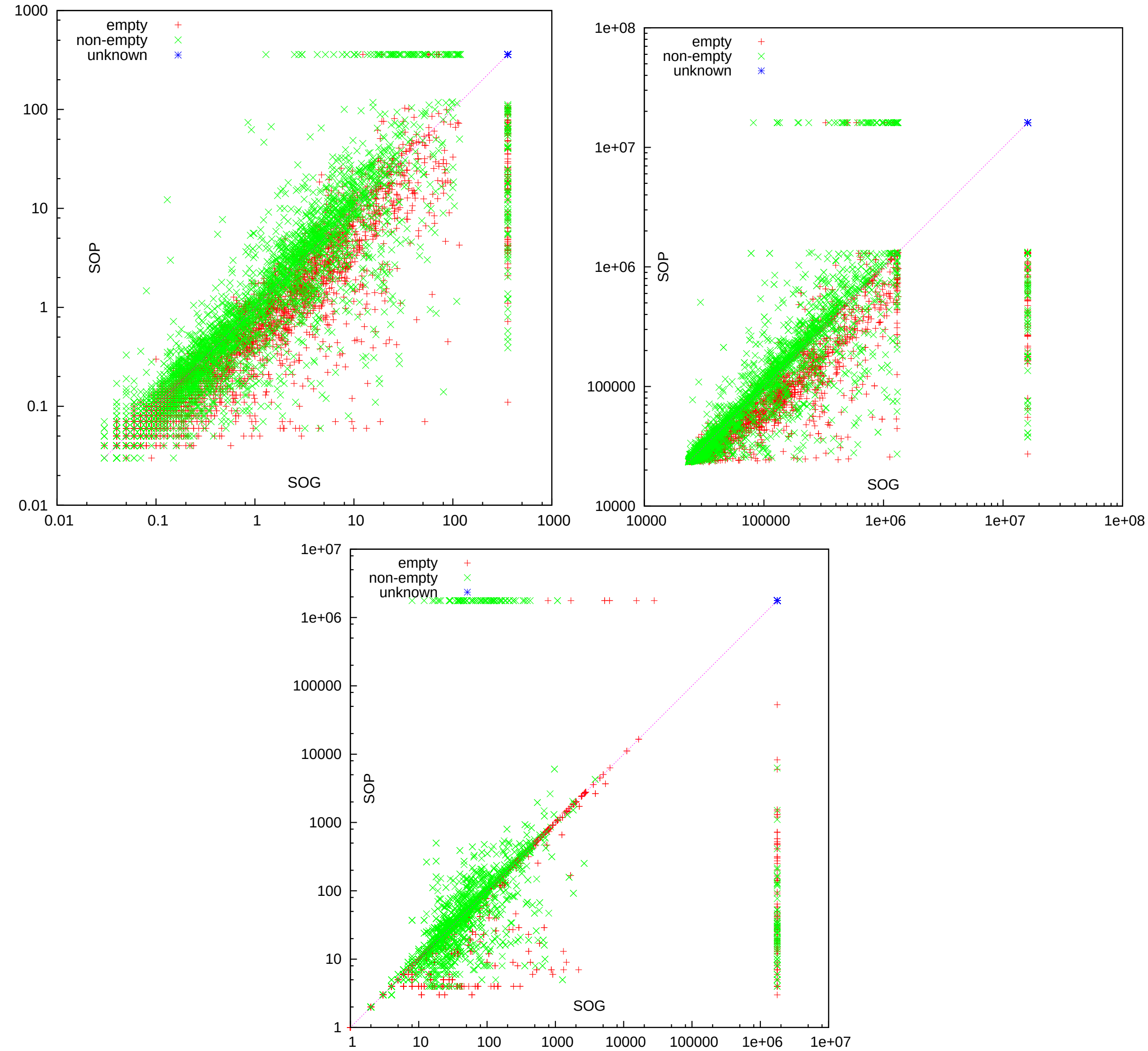


Fig. 12: Performances in time (top-left, in seconds) and memory (top-right, in kilobytes) of observation graph algorithms for 7277 experiments, and product size (bottom). The product size show the number of states of the SOP against the number of state of the product between the Kripke structure and the SOG.

The following cumulative plots (Fig. 13) make this more visible. It also shows that SOP does not necessarily outperform SOG, particularly when the product is non-empty. In fact the SOP may in some cases be much larger in explicit size than the SOG. This is shown in the bottom graph of figure 12.

5.8 BEEM models

We performed extensive experimentations using the BEEM (Benchmarks for Explicit Model checkers [18]) models and LTL formulas.

The BEEM database contains a large set of examples modeling various network protocols, mutual exclusion or consensus problems. We used all the examples for which LTL formulas are provided, and ran the verification for both the formula and its negation, to increase the number of formulas.

Surprisingly enough, all LTL formulas provided by BEEM are stuttering invariant. Thus we also generated a few random formulas which are not stuttering invariant. This benchmark is interesting as it shows some concurrent software oriented examples, and real formulas. However, the number of formulas is quite limited with respect to the previous benchmark. These formulas are also simpler, hence less able to discriminate the various algorithms that depend on the formula automaton. The number of reachable states in these models is also much lower than in the Petri net examples; this makes it more difficult to measure the impact of large Kripke structures on the explicit size of hybrid algorithms. The transition

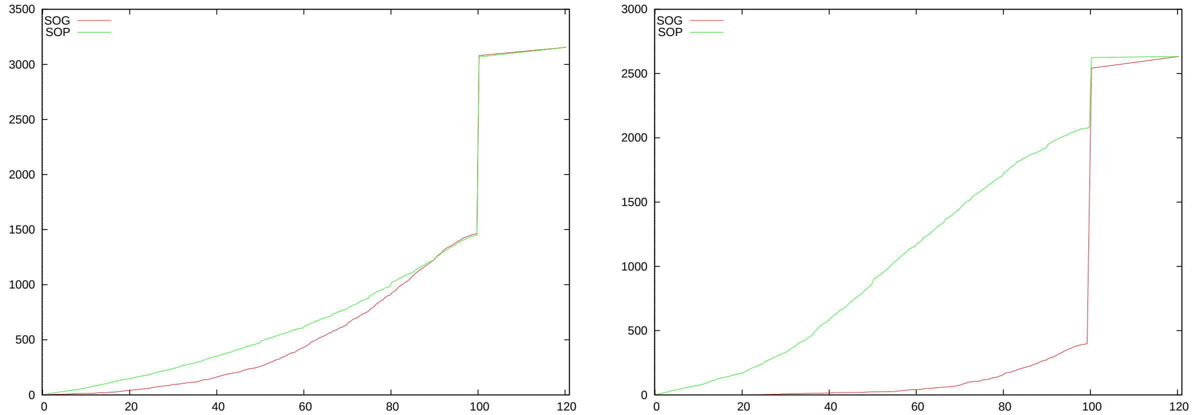


Fig. 13: Compared cumulative performances in time of hybrid stuttering-invariant algorithms with (4048 experiments, left) or without counterexamples (3229 experiments, right).

relation, which is built by interacting with the LTSmin package is also less efficient than that of the Petri net benchmark, and takes less advantage of the automatic saturation features [12] of the SDD library that are heavily sollicited in the SOG, SOP and SLAP algorithms.

A total of 729 formula/model pairs were computed for each algorithm, of which 292 are not stutterin invariant and not used for SOG or SOP. We filtered out model/formula pairs that took less than 0.1 seconds to solve for all methods.

Table 3 gives a synthetic overview of the results presented hereafter and Fig. 14 details these measures with a cumulative distribution function plot.

For stuttering-invariant examples, when the product is empty SLAP or SLAP-FST are the fastest methods in over half of all cases, and they are rarely the slowest. Furthermore, they have the least failure rate, whether the product is empty or not. This table also shows that BCZ has the highest failure rate when the product is empty, although its is the fastest method in one third of cases whether the product is empty or not. SOG and SOP perform honorably when the product is non-empty (hence the on the fly mechanism comes into play), but behave quite porrly in the empty product case. The fully symbolic algorithms (OWCTY, EL) have trouble with non-empty products, but have a low failure rate when the product is empty though they rarely win. On this benchmark set (like on the Petri net measures) EL seems to perform slightly better than OWCTY.

For non stuttering-invariant properties (which were randomly generated), BCZ behaves very well whether the product is empty or not. In non-empty case, as in our other measures, fully symbolic algorithms EL and OWCTY behave poorly with a lot of failures and slowest runtimes. SLAP and SLAP-FST are outperformed by BCZ on this benchmark set, but remain competitive. As shown in the cumulative distribution plot of Fig. 14(bottom), SLAP and SLAP-FST overtake the BCZ curve around 50significant number of problem instances has very good performance (as shown by the steep start for BCZ curve), but then flattens out, while the slope of SLAP and SLAP-FST is more regular. SLAP and SLAP-FST seem to perform more poorly on this benchmark set than in our other measures, however the number of experiments is much more limited here.

The performances are presented as scatter plots using logarithmic scale. Each point represents an experiment, i.e., a model and formula pair. We killed any process that exceeded 800 seconds of runtime; hence for some formulas we were not able to compute the answer. We plot experiments that failed (due to timeout) as if they had taken 2400 seconds, so they are clearly separated from experiments that didn't fail (by the wide white band).

The comparison between SOP and SOG slightly favors SOP for these examples. The two methods often have very similar complexity, as shown by the numerous points on the diagonal. These correspond to cases where the alphabet was not significantly reduced during the verification. Overall SOP outperforms SOG in practically all problem instances. Since SOP is similar but overall better than SOG, we compare other methods to SOP rather than SOG in the other plots.

The comparison between SOP and EL favors SOP for non empty products (unsatisfied formulas) and EL for empty products (satisfied formulas). This means that the on-the-fly mechanism of SOP often allows to answer quite fast (when an accepting cycle exists), but when the full product needs to be explored EL is actually more effective than SOP.

			OWCTY	EL	BCZ	SOG	SOP	SLAP	SLAP-FST
without **X**	empty	Win	2 (2%)	3 (3%)	31 (31%)	3 (3%)	1 (1%)	30 (30%)	40 (40%)
	(100 cases)	Lose	22 (22%)	10 (10%)	61 (61%)	26 (26%)	25 (25%)	3 (3%)	2 (2%)
		Fail	8 (8%)	7 (7%)	38 (38%)	16 (16%)	16 (16%)	2 (2%)	0 (0%)
	non empty	Win	0 (0%)	0 (0%)	106 (32%)	24 (7%)	101 (30%)	70 (21%)	57 (17%)
	(329 cases)	Lose	152 (46%)	199 (60%)	48 (14%)	28 (8%)	21 (6%)	19 (5%)	23 (6%)
		Fail	69 (20%)	72 (21%)	15 (4%)	16 (4%)	13 (3%)	19 (5%)	20 (6%)
with **X**	empty	Win	10 (10%)	10 (10%)	47 (51%)			18 (19%)	30 (32%)
	(92 cases)	Lose	37 (40%)	33 (35%)	21 (22%)			23 (25%)	14 (15%)
		Fail	4 (4%)	3 (3%)	0 (0%)			4 (4%)	4 (4%)
	non empty	Win	1 (0%)	2 (1%)	125 (62%)			39 (19%)	43 (21%)
	(200 cases)	Lose	102 (51%)	108 (54%)	33 (16%)			15 (7%)	24 (12%)
		Fail	40 (20%)	40 (20%)	3 (1%)			14 (7%)	17 (8%)

Table 3: On all experiments (grouped with respect to the existence of a counterexample and the use of a **X** operator in the LTL formula), we count the number of cases a specific method has (Win) the best time or (Lose) it has either run out of time or it has the worst time amongst successful methods. The Fail line shows how much of the Lost cases were timeouts. The sum of a line may exceed 100% if several methods are equally placed.

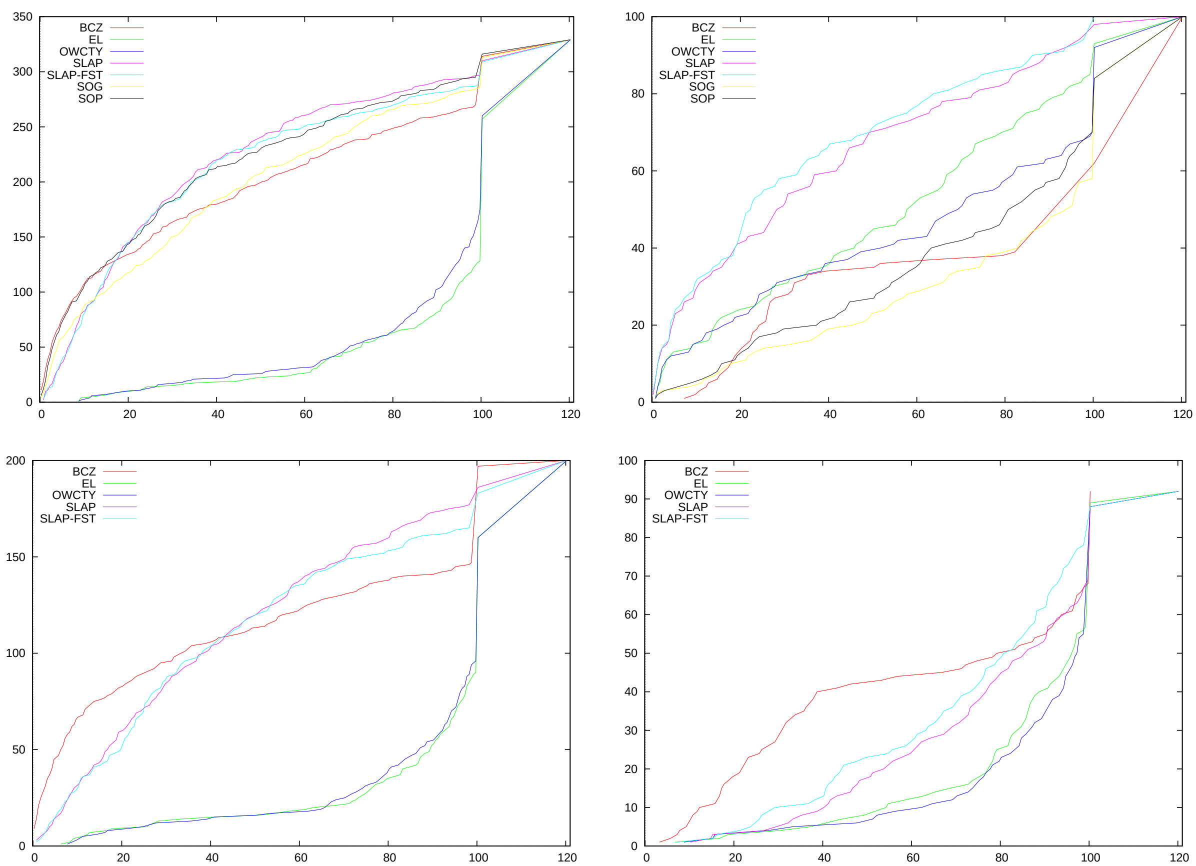


Fig. 14: Cumulative plots comparing the time of all methods on the BEEM models. Non-empty products are shown on the left, and empty products on the right. Top for stuttering invariant properties and bottom for LTL formulae with the **X** operator.

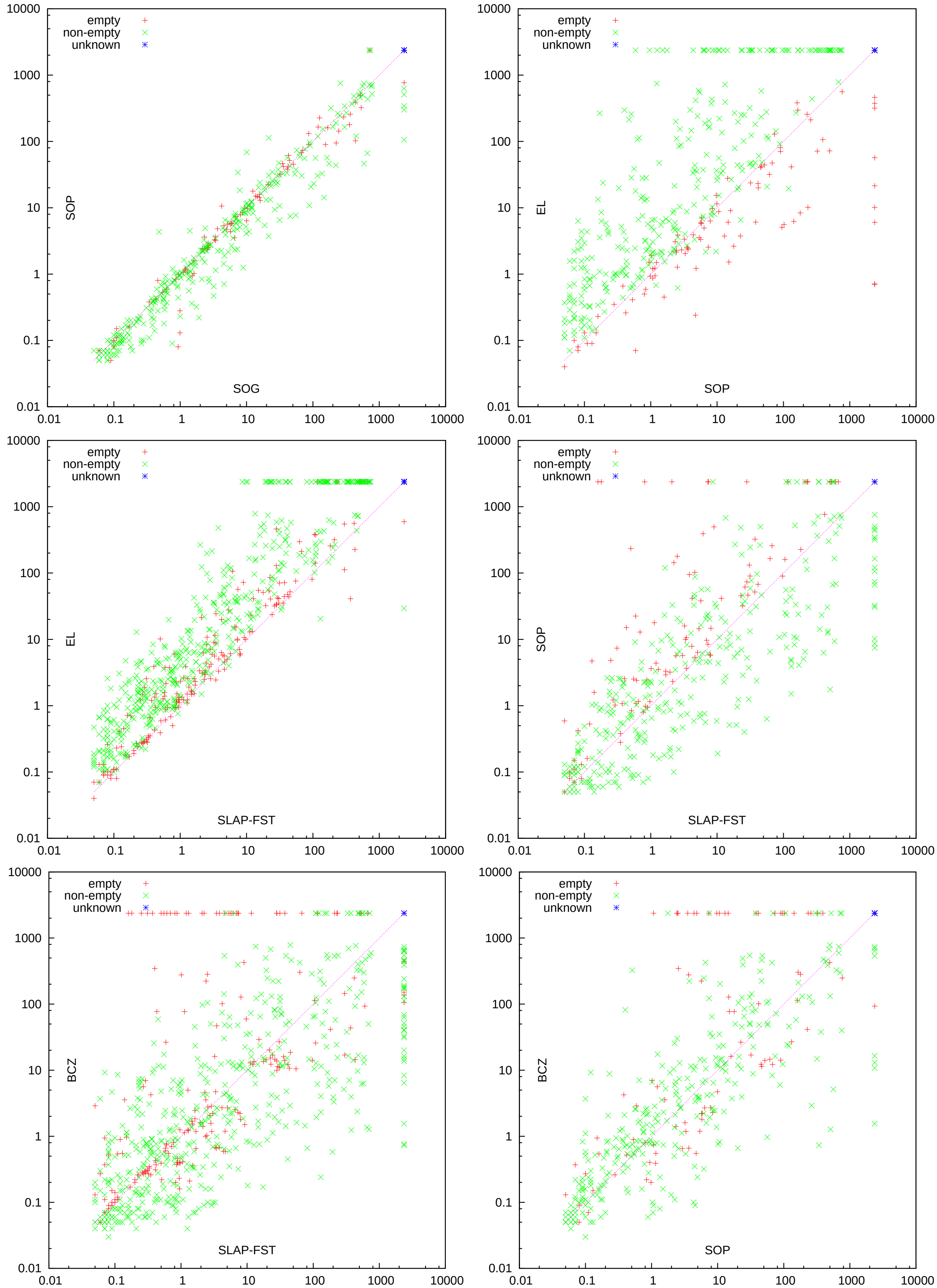


Fig. 15: Performances in time for 721 experiments using the BEEM models. Plots with SOG or SOP only contain the 429 stuttering-invariant experiments.

The comparison between EL and SLAP-FST clearly favors SLAP-FST in practically all experiments, with a significant portion of experiments that failed with EL but not with SLAP-FST. SLAP-FST like SOP benefits from the on-the-fly verification that often allows to answer without building the full product. But because the SLAP-FST is often quite small in explicit size, it performs quite well whether the product is empty or not.

The plot comparing SOP to SLAP-FST is much more ambiguous. SLAP clearly outperforms SOP when the product is empty, but there are many problem instances where SOP find an accepting cycle faster. This could be due to the DFS order chosen during the on-the-fly verification, which makes these results difficult to interpret, as either of the algorithms could be lucky. However, a significant number of problem instances were solved by SOP and not by SLAP-FST; hence overall both algorithms can be useful in practice.

Comparison of SLAP-FST to BCZ, and of SOP to BCZ show that these three methods can be complementary. BCZ unfortunately has a large number of failures for empty products on stuttering invariant examples, but can perform quite well in a significant number of problem instances.

Comparison to the LTL model-checker provided by DiVine was attempted but when running in non compiled mode the run times are prohibitive, and when using the compiled mode the results are wrong on a significant number of problem instances, hence we have little confidence in the current distribution of DiVine.

6 Conclusion and Perspectives

We have presented two new hybrid techniques: the *symbolic observation product* (SOP), is a generalization of the *symbolic observation graph* (SOG) that diminishes the set of observed atomic propositions as we progress in the product, and the *Self-Loop Aggregation Product* (SLAP) that exploits the self-loops of the property automaton even if it does not express a stuttering formula.

During our evaluation, we have found that SOP improves SOG, and outperform fully symbolic algorithms EL and OWCTY in the presence of counterexamples. SLAP surpasses both SOG (always), OWCTY (always), and SOP (when the product is empty). BCZ performs better than EL or OWCTy when the product is not empty and more poorly otherwise; the only set of experiments where it shows favorable results is the BEEM models with randomly generated formulas using the next LTL operator. When the product is not empty, SOP and SLAP techniques seem complementary. SLAP-FST provides the overall best performance and lowest failure rate of all the methods we compared.

This work opens several perspectives.

Firstly, the above two techniques replace the product used in the traditional automata-theoretic approach to model-checking in order to reduce the product graph while preserving the result of the emptiness-check. Another technique with the same goal is the *Symbolic Synchronized Product* (SSP) [1]. The SSP studies the symmetries of the model with respect to the current state of the property automaton, to aggregate symmetrically equivalent states. A classical emptiness-check of the SSP is possible, but Baarir and Duret-Lutz [1] also suggested two emptiness checks variants taking advantage of the inclusion between the aggregates. It would be interesting to see if a similar inclusion-aware emptiness checks could be used with SOG, SOP, and SLAP.

Secondly, representing a stuttering property as a *testing automata* [13] is another way to take advantage of stuttering transitions in the model. In the product between a KS and a testing automaton, the latter does not move when the KS is stuttering. A possible perspective would be to adapt our stuttering-based techniques (SOG and SOP) to aggregate all states from the KS corresponding to one state of the testing automaton.

Finally, since the SOG is a KS, and the SLAP is built upon a KS, it is possible to construct the SLAP of SOG. This is something we did not implement due to technical issues: in this case the aggregates are sets of sets of states.

References

1. S. Baarir and A. Duret-Lutz. Emptiness check of powerset Büchi automata. In *Proc. of ACSD'07*, pp. 41–50. IEEE Computer Society.
2. A. Biere, E. M. Clarke, and Y. Zhu. Multiple state and single state tableaux for combining local and global model checking. In *Proc. of CSD'99*, volume 1710 of *LNCS*, pp. 163–179. Springer.
3. S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In *Proc. of CAV'10*, volume 6174 of *LNCS*, pp. 354–359. Springer.

4. G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proc. of TACAS'03*, volume 2619 of *LNCS*, pp. 379–393. Springer.
5. C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithm for the verification of temporal properties. In *Proc. of CAV'90*, volume 531 of *LNCS*, pp. 233–242. Springer.
6. 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*, pp. 143–158. Springer.
7. A. Duret-Lutz and D. Poitrenaud. Spot: an extensible model checking library using transition-based generalized Büchi automata. In *Proc. of MASCOTS'04*, pp. 76–83. IEEE Computer Society Press.
8. E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional μ-calculus (extended abstract). In *Proc. of LICS'86*, pp. 267–278. IEEE Computer Society.
9. K. Etessami. Stutter-invariant languages, ω-automata, and temporal logic. In *Proc. of CAV'99*, volume 1633 of *LNCS*, pp. 236–248. Springer.
10. K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Proc. of TACAS'01*, volume 2031 of *LNCS*, pp. 420–434. Springer.
11. 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*, pp. 198–210. Springer.
12. A. Hamez, Y. Thierry-Mieg, and F. Kordon. Hierarchical set decision diagrams and automatic saturation. In *Proc. of ICATPN'08*, volume 5062 of *LNCS*, pp. 211–230. Springer.
13. H. Hansen, W. Penczek, and A. Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. In *Proc. of FMICS'02*, volume 66(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier.
14. R. Kaivola and A. Valmari. The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic. In *Proc. of CONCUR'92*, volume 630 of *LNCS*, pp. 207–221. Springer.
15. Y. Kesten, A. Pnueli, and L. on Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. of ICALP'98*, volume 1443 of *LNCS*, pp. 1–16. Springer.
16. K. Klai and D. Poitrenaud. MC-SOG: An LTL model checker based on symbolic observation graphs. In *Proc. of Petri Nets'08*, volume 5062 of *LNCS*, pp. 288–306. Springer.
17. I. Kokkarinen, D. Peled, and A. Valmari. Relaxed visibility enhances partial order reduction. In *Proc. of CAV'97*, volume 1254 of *LNCS*, pp. 328–339. Springer.
18. R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *Proc. of SPIN'07*, volume 4595 of *LNCS*, pp. 263–267. Springer.
19. D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997.
20. K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In *Proc. of SPIN'07*, volume 4595 of *LNCS*, pp. 149–167. Springer.
21. R. Sebastiani, S. Tonetta, and M. Y. Vardi. Symbolic systems, explicit properties: on hybrid approches for LTL symbolic model checking. In *Proc. of CAV'05*, volume 3576 of *LNCS*, pp. 350–363. Springer.
22. F. Somenzi, K. Ravi, and R. Bloem. Analysis of symbolic SCC hull algorithms. In *Proc. of FMCAD'02*, volume 2517 of *LNCS*, pp. 88–105. Springer.
23. Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, and F. Kordon. Hierarchical set decision diagrams and regular models. In *Proc. of TACAS'09*, volume 5505 of *LNCS*, pp. 1–15. Springer.
24. A. Valmari. The state explosion problem. In *Lectures on Petri Nets 1: Basic Models*, volume 1491 of *LNCS*, pp. 429–528. Springer, 1998.
25. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of Banff'94*, volume 1043 of *LNCS*, pp. 238–266. Springer.