Self-Loop Aggregation Product — A New Hybrid Approach to 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 the *Self-Loop Aggregation Product* (SLAP), a new hybrid technique that replaces the synchronized product used in the automata-theoretic approach for LTL model checking. The proposed product is an explicit graph of aggregates (symbolic sets of states) that can be interpreted as a Büchi automaton. The criterion used by SLAP to aggregate states from the Kripke structure is based on the analysis of self-loops that occur in the Büchi automaton expressing the property to verify. Our hybrid approach allows 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. Our experiments show that this technique often outperforms other existing (hybrid or fully symbolic) 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 [20]. This verification process suffers from a well known state explosion problem.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 [4].

Another source of optimization is to take advantage of stuttering equivalence between paths in the Kripke structure when verifying a stuttering-invariant property [8]: this has been done either by ignoring some paths in the Kripke structure [13], or by representing the property using a *testing automaton* [12]. 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 fix-point computations (see [9, 18] for comparisons and [14] 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, 10, 17, 15] to get the best of both worlds. They are referred to as **hybrid approaches**. Most of these approaches consist in replacing the Kripke structure by an explicit graph where each node contains sets of states (called aggregates throughout this paper), that is an abstraction preserving properties of the original structure. For instance in Biere et al.'s approach [2], 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. The Symbolic Observation Graph [10] takes this idea one step further in the context of stuttering invariant properties: each aggregate 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.

Sebastiani et al.'s approach [17] 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 approach we define in this paper is based on explicit graphs of aggregates (symbolic sets of states) that can be interpreted as Büchi automata. With this combination, we can use classical 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 aggregation criterion is 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. We allow to stutter according to a boolean formula computed as the disjunction of the labels of self-loops of the automaton. This aggregation graph is called the *Self-Loop Aggregation Product* (SLAP) and preserves full Büchi expressible properties.

This paper is organized as follows. Section 2 introduces our notations and presents the basic automata-theoretic approach. Section 3 defines our new hybrid construction SLAP. We explain how we implemented this approach and how it compares to others in Section 4.

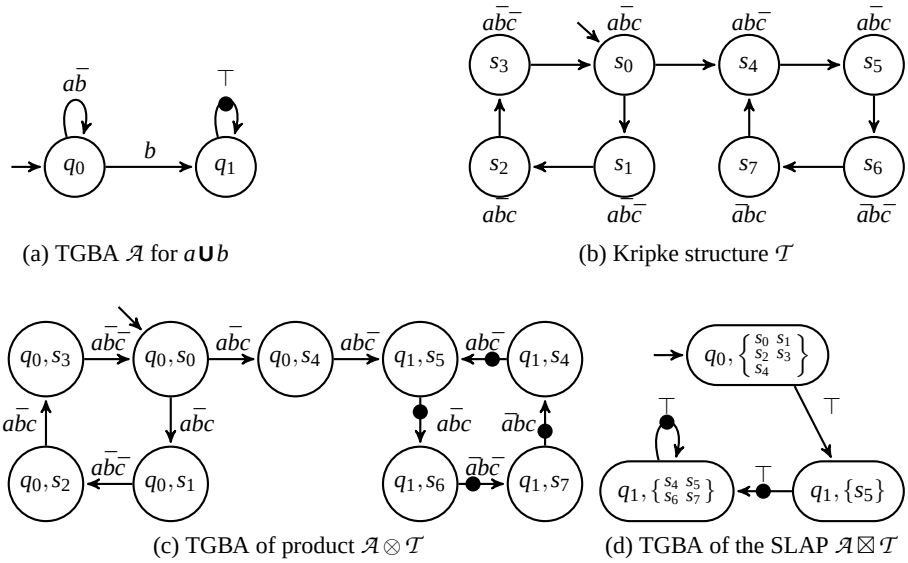
2 Preliminaries

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 $\neg$.

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

[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) TGBA of product $\mathcal{A}\otimes\mathcal{T}$

(d) TGBA of the SLAP $\mathcal{A}\boxtimes\mathcal{T}$

Fig. 1: Examples

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.

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 [5].

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

- AP *is a finite set of atomic propositions,*
- Q *is a finite set of states,*

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

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 $\mathrm{AP} = \{a, b, c\}$. The state graph of a system is typically represented by a KS, where state labels in the KS give the atomic proposition truth values in a given state of the system.

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

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

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

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

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

Fig. 1c 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 $\lambda(s_0) = abc$ we have $\lambda(s_0) \models ab$, successors $\{s_1, s_4\}$ of s_0 in the KS will be synchronized through the edge $q_0 \xrightarrow{ab, \emptyset} q_0$ 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 $\lambda(s_4) = abc \models b$. 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.

3 Self-Loop Aggregation Product (SLAP)

This section presents 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.

3.1 Definition

The notion of self-loop aggregation is captured by $\mathrm{SF}(q, ac)$, the **S**elf-loop **F**ormulas (labeling edges $q \to q$) 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 $\mathrm{FSucc}(a, f) = \{s' \in \Gamma \mid \exists s \in a, s \to s' \in \Delta \wedge \lambda(s) \models f\}$. That is, first **F**ilter a to only keep states satisfying f, then produce their **Succ**essors. 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 4 (SLAP of a TGBA and a KS). *Given a TGBA* $\mathcal{A} = \langle \mathrm{AP}', Q, \mathcal{F}, \delta, q^0 \rangle$ *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* $\mathcal{A} \boxtimes \mathcal{T} = \langle \emptyset, Q_\boxtimes, \mathcal{F}, \delta_\boxtimes, q^0_\boxtimes \rangle$ *where:*

- $Q_\boxtimes = Q \times (2^\Gamma \setminus \{\emptyset\})$
- $\delta_\boxtimes = \left\{ (q_1, a_1) \xrightarrow{\top, ac} (q_2, a_2) \;\middle|\; \begin{array}{l} \exists f \in \mathbb{B}(\mathrm{AP}') \text{ s.t. } q_1 \xrightarrow{f, ac} q_2 \in \delta, \\ \qquad q_1 = q_2 \Rightarrow ac \neq \emptyset, \text{ and} \\ \qquad a_2 = \mathrm{FReach}(\mathrm{FSucc}(a_1, f), \mathrm{SF}(q_2, ac)) \end{array} \right\}$
- $q^0_\boxtimes = (q^0, \mathrm{FReach}(\{s_0\}, \mathrm{SF}(q^0, \emptyset)))$

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 $(q_1, a_1) \xrightarrow{\top, ac} (q_2, a_2)$.

$Q \times 2^\Gamma$ might seem very large but, as we will see in section 4.2 in practice the reachable states of the SLAP is a much smaller set than that of the product $Q \times \Gamma$. Furthermore the FReach operation can be efficiently implemented as a symbolic least fix point.

Fig. 1d 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 $\{s_4\}$). Because $\mathrm{SF}(q_1, \emptyset) = \bot$ the successors of $\{s_5\}$ are not gathered when building $(q_1, \{s_5\})$. 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.

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 SLAP is equivalent to the emptiness of the language of the original synchronized product (see Theorem 1). This result is progressively demonstrated in the following. We proceed by construction, i.e., if there exists an accepting run of the SLAP then we build an accepting run of the original product and vice versa. In order to ease the proof, we introduce some intermediate lemmas.

Lemma 1. *Let* $\mathcal{A}$ *and* $\mathcal{T}$ *be defined as in Definition 4. 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. 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. $(q_1, s_1) \xrightarrow{ac} (q_2, x_2) \xrightarrow{\alpha_2} (q_2, s_2)$.

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

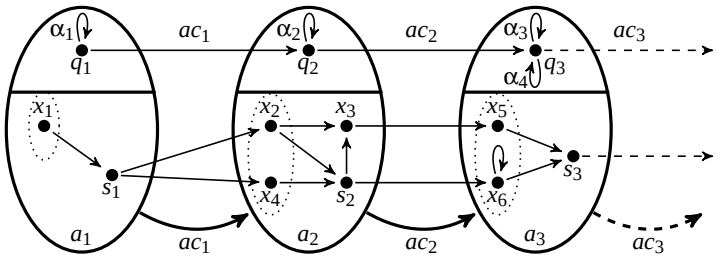


Fig. 2: A prefix $(q_1,a_1) \xrightarrow{ac_1} (q_2,a_2) \xrightarrow{ac_2} (q_2,a_2)$ of a run of some SLAP $\mathcal{A} \boxtimes \mathcal{T}$ (with *different* $\mathcal{A}$ and $\mathcal{T}$ from Fig. 1) is shown using big ellipses and bended arrows. The straight lines also shows the underlying connections between the states $\{q_1,q_2,q_3,\ldots\}$ 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 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 1.

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 $t_1 \to t_2 \to \cdots \to s_2$ in the Kripke structure.

By definition of $\delta_{\boxtimes}$, if $t_1,t_2,\ldots,s_2$ 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. □

Lemma 2. *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 $\{(q_n,s) \mid s \in a_n\} \subseteq Q \times \Gamma$.

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 1. 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 $\mathcal{A} \otimes \mathcal{T}$. □

Lemma 3. *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 $\varphi_n : \{0, \ldots, m-1\} \to \{0, \ldots, n-1\}$ 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 demonstrate 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 $s_n \to s_{n+1}$, 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 4. *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 3 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 1. *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 2; $\Longrightarrow$ follows from Lemma 4. □

3.3 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 TGBA 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. As discussed in the performance section, this variation is on average more effective than the basic SLAP algorithm.

4 Experimentations

4.1 Implementation

We have implemented several hybrid or fully symbolic algorithms within our framework to allow fair algorithmic comparisons. The software, available from `ddd.lip6.fr`, builds upon two existing components: Spot and SDD/ITS.

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 ”the best explicit LTL model-checker” [16]. 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 [19]. 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 [11] apply the symbolic saturation algorithm [3]. These features allow the SDD/ITS package to offer very competitive performance.

The fully symbolic OWCTY (One-Way Catch Them Young) and EL (Emerson-Lei) algorithms [9, 18] were implemented directly on top of the ITS interface; they use 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 or EL with the forward transition relation.

The SOG [10] (Symbolic Observation Graph) and BCZ [2] (Biere-Clarke-Zhu) are implemented as objects conforming to Spot's Kripke interface. They load an ITS model, then build the SOG or BCZ on the fly, as required by the emptiness check of the product with the formula automaton.

The SLAP is implemented as an object conforming to Spot's product interface. The SLAP class takes an ITS model and a TGBA (the formula automaton) as input parameters, and builds its specialized product on the fly, driven by the emptiness-check algorithm.

4.2 Benchmark

We use here classic scalable Petri net examples taken from Ciardo's benchmark set [3]: slotted ring, Kanban, flexible manufacturing system, and dining philosophers. The model occurences we used had from a few million to 10^{66} reachable states. More details are available in our technical report [6].

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: $(\mathbf{GF}\,p_1 \wedge \mathbf{GF}\,p_2 \ldots) \implies \varphi$.

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

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: these trivial cases represent only 4.2% of the entire benchmark, and were too fast to allow any pertinent comparison.

Table 1 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

		OWCTY	EL	BCZ	SOG	SLAP	SLAP-FST
empty	Fast	118 (3%)	189 (5%)	53 (1%)	595 (18%)	1359 (42%)	1811 (56%)
(3227 cases)	Slow	259 (8%)	271 (8%)	2909 (90%)	509 (15%)	245 (7%)	93 (2%)
	Fail	220 (6%)	252 (7%)	1785 (55%)	301 (9%)	212 (6%)	86 (2%)
non empty	Fast	3 (0%)	10 (0%)	209 (5%)	782 (19%)	2510 (62%)	1406 (34%)
(4046 cases)	Slow	1869 (46%)	1390 (34%)	1940 (47%)	315 (7%)	70 (1%)	40 (0%)
	Fail	803 (19%)	817 (20%)	1069 (26%)	262 (6%)	69 (1%)	33 (0%)

Table 1: On all experiments (grouped with respect to the existence of a counterexample), we count the number of cases a specific method has (Fast) the best time or (Slow) 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.

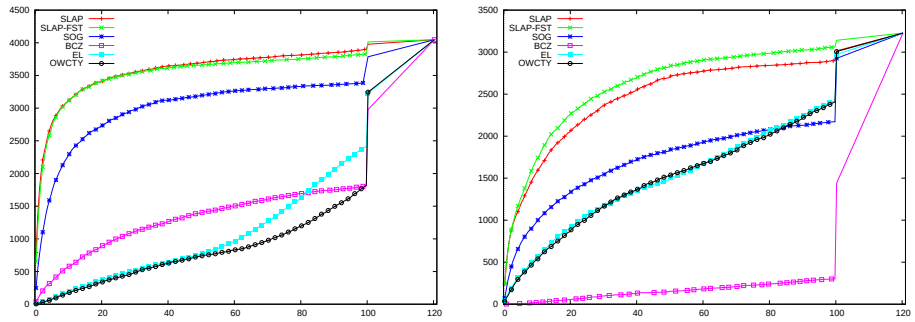


Fig. 3: Cumulative plots comparing the time of all methods. Non-empty products are shown on the left, and empty products on the right.

has the highest failure rate and that the fully symbolic algorithms (OWCTY, EL) have trouble with non-empty products.

Table 1 presents only the best and the worst methods. While Fig. 3 allows 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 for the non-empty cases shows that the on-the-fly mechanism allows all hybrid algorithms (SLAP, SLAP-FST, SOG, BCZ) to outperform the symbolic ones

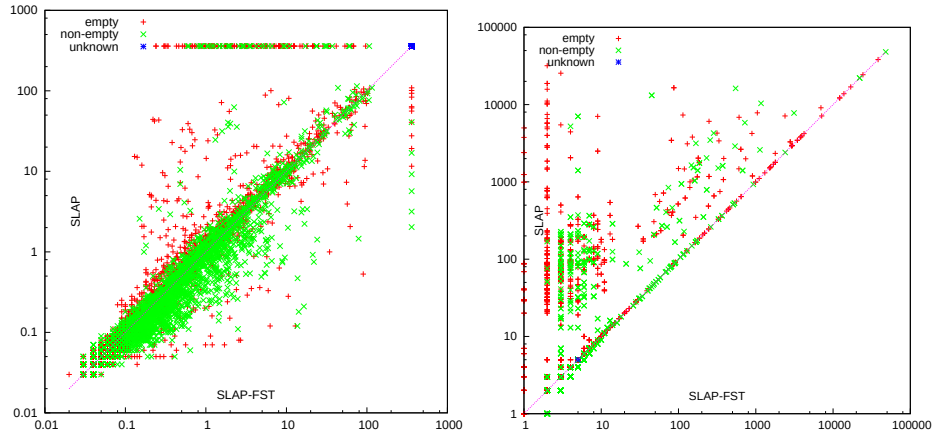


Fig. 4: Comparison of SLAP-FST against SLAP. Left: time; Right: product size.

(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.

The right plot for the empty cases shows 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.

SLAP-FST and SLAP have similar performance, with a slight edge for SLAP-FST when the product is empty.

EL appears slightly superior to OWCTY in the non-empty case, while they have similar performances in the empty case.

SOG shows good results when there is a counterexample, and it performs better than BCZ in most cases. However SOG only supports stuttering-invariant properties.

To study the differences between SLAP and SLAP-FST consider the scatter plots from Fig. 4. 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).

SLAP is on the average faster (and consume less memory [6]) than SLAP-FST, but fails more often. 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.

In Fig. 5 we compare SLAP-FST to the four other methods from the literature, using the same kind of logarithmic scatter plots in time. Unsurprisingly, the only method that appears competitive is SOG; but to our advantage, SOG is not able to handle non stuttering-invariant properties.

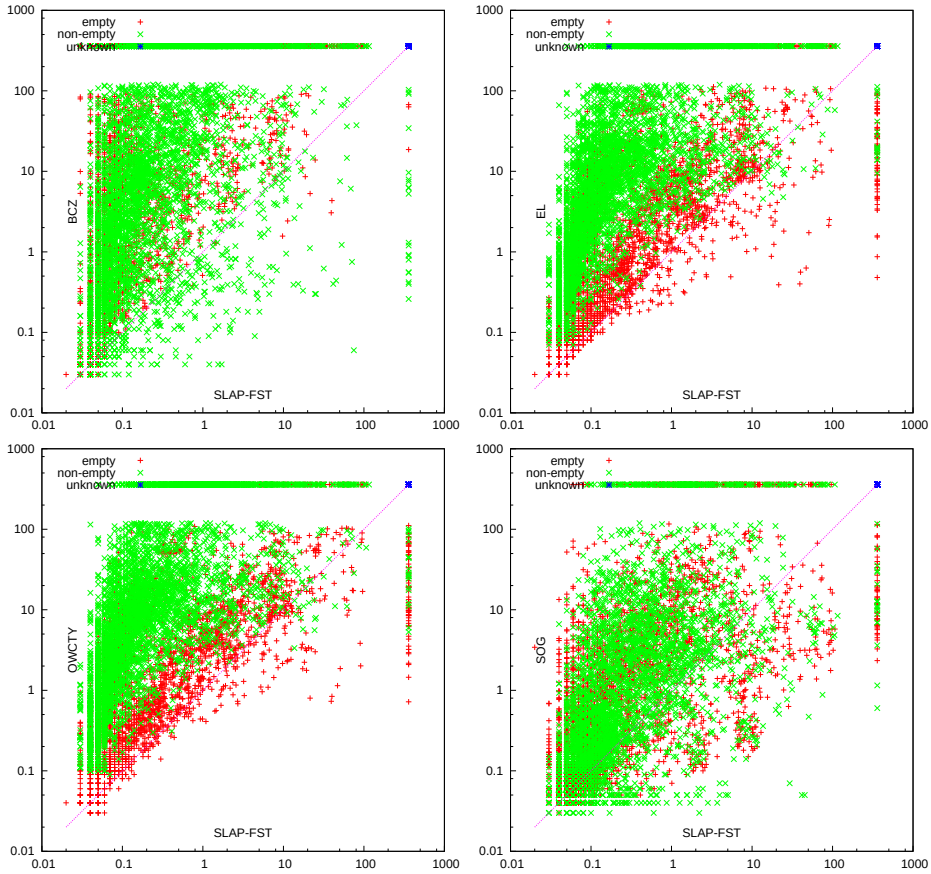


Fig. 5: Comparison of SLAP-FST against the four other methods.

5 Conclusion and Perspectives

We have presented a new hybrid technique, the *Self-Loop Aggregation Product*, 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 SLAP (and especially its variant SLAP-FST) significantly outperforms the other hybrid and symbolic methods we implemented. In presence of a counterexample we can benefit from the on-the-fly mechanism, while purely symbolic methods like EL and OWCTY cannot. On empty products, the SLAP-FST has a small explicit size, allowing to outperform other hybrid algorithms.

This work opens several perspectives.

It would be interesting to compare our approach to the property-driven partitioning [17] even if this hybrid algorithm uses a fully symbolic emptiness check and is not based on an aggregation criterion.

Another class of methods we would like to compare against, are purely explicit ones, in particular those based on partial order reductions.

The SLAP technique replaces 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.

We also used this idea to improve the SOG, by working at the product-level and reducing the set of observed propositions according to the current state of the TGBA. This technique called Symbolic Observation Product (SOP) is described in our technical report [6].

Another idea would be to take advantage of the inclusion between the aggregates to detect cycles earlier. This would require a dedicated emptiness check such as those proposed by Baarir and Duret-Lutz [1].

Finally, since the SOG is a Kripke structure, 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. G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proc. of TACAS'03*, volume 2619 of *LNCS*, pp. 379–393. Springer.
4. 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.
5. J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *Proc. of SPIN'05*, volume 3639 of *LNCS*, pp. 143–158. Springer.
6. A. Duret-Lutz, K. Klai, D. Poitrenaud, and Y. Thierry-Mieg. Combining explicit and symbolic approaches for better on-the-fly LTL model checking. Technical Report 1106.5700, arXiv, June 2011. Extended version of the present paper, presenting two new techniques instead of one. `http://arxiv.org/abs/1106.5700`.
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. K. Etessami. Stutter-invariant languages, ω-automata, and temporal logic. In *Proc. of CAV'99*, volume 1633 of *LNCS*, pp. 236–248. Springer.
9. 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.
10. 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.
11. 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.
12. 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.

13. 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.
14. 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.
15. 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.
16. K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In *Proc. of SPIN'07*, volume 4595 of *LNCS*, pp. 149–167. Springer.
17. 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.
18. 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.
19. 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.
20. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of Banff'94*, volume 1043 of *LNCS*, pp. 238–266. Springer.