Form Methods Syst Des (2007) 31: 177–196
DOI 10.1007/s10703-007-0036-3

GSTE is partitioned model checking

Roberto Sebastiani · Eli Singerman · Stefano Tonetta · Moshe Y. Vardi

Published online: 26 May 2007

Abstract Verifying whether an ω-regular property is satisfied by a finite-state system is a core problem in model checking. Standard techniques build an automaton with the complementary language, compute its product with the system, and then check for emptiness. Generalized symbolic trajectory evaluation (GSTE) has been recently proposed as an alternative approach, extending the computationally efficient symbolic trajectory evaluation (STE) to general ω-regular properties. In this paper, we show that the GSTE algorithms are essentially a partitioned version of standard symbolic model-checking (SMC) algorithms, where the partitioning is driven by the property under verification. We export this technique of property-driven partitioning to SMC and show that it typically does speed up SMC algorithms.

A shorter version of this paper has been presented at CAV'04 (R. Sebastiani et al., Lecture Notes in Comput. Sci., vol. 3114, pp. 143–160, 2004).
R. Sebastiani supported in part by the CALCULEMUS! IHP-RTN EC project, code HPRN-CT-2000-00102, by a MIUR COFIN02 project, code 2002097822_003, and by a grant from the Intel Corporation.
M.Y. Vardi supported in part by NSF grants CCR-9988322, CCR-0124077, CCR-0311326, IIS-9908435, IIS-9978135, EIA-0086264, and ANI-0216467 by BSF grant 9800096, and by a grant from the Intel Corporation.

R. Sebastiani
Dipartimento di Informatica e Telecomunicazioni, Università di Trento, Trento, Italy
e-mail: rseba@dit.unitn.it

E. Singerman
Intel Israel Design Center, Haifa, Israel
e-mail: eli.singerman@intel.com

S. Tonetta (✉)
Faculty of Informatics, University of Lugano, Lugano, Switzerland
e-mail: tonettas@lu.unisi.ch

M.Y. Vardi
Department of Computer Science, Rice University, Houston, USA
e-mail: vardi@cs.rice.edu

Keywords Symbolic model checking · GSTE · Property-driven partitioning

Abbreviations
MC Model Checking
SMC Symbolic MC
STE Symbolic Trajectory Evaluation
GSTE Generalized STE
FG Fair Graph
BA Büchi Automaton
AG Assertion Graph

1 Introduction

Verifying whether an ω-regular property is satisfied by a finite-state system is a core problem in Model Checking (MC) [10, 34, 45]. Standard MC techniques build a complementary *Büchi automaton* (BA), whose language contains all violations of the desired property. They then compute the product of this automaton with the system, and then check for emptiness [34, 44]. To check emptiness, one has to compute the set of *fair states*, i.e., those states of the product automaton that are extensible to a fair path. This computation can be performed in linear time by using a depth-first search [13]. The main obstacle to this procedure is *state-space explosion*, i.e., the product is usually too big to be handled. Symbolic model checking (SMC) [4] tackles this problem by representing the product automaton symbolically, usually by means of BDDs. Most symbolic model checkers compute the fair states by means of some variant of the doubly-nested-fixpoint Emerson-Lei algorithm (EL) [16, 18, 39].

Another approach to formal verification is that of Symbolic Trajectory Evaluation (STE) [42], in which one tries to show that the system satisfies the desired property by using symbolic simulation and quaternary symbolic abstraction. This often enables quick response time, but is restricted to very simple properties, constructed from Boolean implication assertions by means of conjunction and the temporal next-time operator [7]. In particular, STE is limited to bounded properties [32]. Recently, GSTE [49, 51] has been proposed as an extension of STE that can handle all ω-regular properties. In this framework, properties are specified by means of *Assertion Graphs* (AG). The GSTE algorithm augments symbolic simulation with a fixpoint iteration. GSTE inherited from STE many techniques such as quaternary symbolic abstraction, symbolic indexing [49, 51] and functional vectors [48]. Recent work on GSTE [48, 50] has described various case studies and has focused mainly on abstraction in GSTE. The fundamental relation between GSTE and SMC, however, has not been completely clarified. The basic relationship between AGs and BAs is sketched in [29], but the algorithmic relationship between GSTE and SMC has not been studied.

In this work, we analyze the property-specification language and the checking algorithm used by GSTE and compare them to those used in SMC. (We deal neither with abstraction nor with the state representation, which are orthogonal issues.) We first fill in the details not given in [29] to show that assertion graphs are essentially *universal* ω-automata [35], which require all runs to be accepting. Universal automata enjoy the advantage of easy complementation; in fact, they can be viewed as nondeterministic automata for the complementary property (this feature is attained in the COSPAN system by using deterministic automata [25]). Formally, given a BA, one can easily construct an AG for the complementary language, and vice versa. This permits us to do a direct comparison between the algorithms underlying GSTE and SMC.

We then point out that the GSTE algorithms are essentially a partitioned version of the standard SMC algorithms. SMC algorithms operate on subsets of the product state space $S \times V$, where S is the state space of the system and V is the state space of the complementary automaton. We show that GSTE operates on partitioned subsets of the product state space. The partitioning is driven by the automaton state space. The GSTE analog of a subset $Q \subseteq S \times V$ is the partition $\{Q_v : v \in V\}$, where $Q_v = \{s : (s, v) \in Q\}$. The GSTE algorithms are in essence an adaptation of the standard SMC algorithms to the partitioned state space. Thus, rather than operate on subsets of product state space P, GSTE operates on arrays of subsets of S, representing a partitioning of P. We refer to such partitioning as *property-driven partitioning*.

Finally, we proceed to explore the benefits of property-driven partitioning in the framework of SMC. We use NuSMV [8] as our experimental platform in the context of LTL model checking. We added to NuSMV the capability of property-driven partitioned SMC, both for safety LTL properties and for full LTL properties, and compared the performance of SMC with partitioned SMC. We find that property-driven partitioning is an effective technique for SMC, as partitioned SMC is typically faster than SMC. The major factor seems to be the reduction in the number of BDD variables, which more than compensates for the additional algorithmic overhead for handling a partitioned state space.

Partitioning techniques have often been proposed in order to tackle the state space explosion problem. (We refer here to *disjunctive* partitioning, rather than to the orthogonal technique of *conjunctive* partitioning, which is used to represent and/or manipulate large state spaces [3, 22, 36].) Static partitioning techniques, which require an analysis of the state space, have been discussed, in [37, 38]. Dynamic partitioning techniques, which are driven by heuristics to reduce BDD size, have been discussed, in [5, 6, 19, 30]. Partitioning has been used in [24, 27] to develop a distributed approach to SMC. Wang et al. [46], Wang and Hachtel [47] decomposes the system into strongly-connected components (SCCs) and applies a particular language-emptiness procedure to every SCC according to its "strength."

Property-driven partitioning is orthogonal to previous partitioning techniques. Unlike dynamic partitioning techniques, no expensive BDD-splitting heuristics are required. Unlike previous static partitioning techniques, property-driven partitioning is fully automated and no analysis of the system state space is needed. The technique is also of interest because it represents a novel approach to automata-theoretic verification. So far, automata-theoretic verification means that either both system and property automaton state spaces are represented explicitly (e.g. in SPIN [28]) or symbolically (in NuSMV [8] or in Cadence SMV[1]). Just like GSTE, property-driven partitioning enables a hybrid approach, in which the property automaton, whose state space is often quite manageable, is represented explicitly, while the system, whose state space is typically exceedingly large is represented symbolically. Another hybrid approach [1, 9] applies a mixed depth-first/breadth-first search to the powerset automaton. In [9], only reachability is taken into account and the experiments focus on planning problems. In [1], the search is applied to the product with a tableau corresponding to an LTL formula and it works on-the fly. In this case, the authors manage to keep the size of product linear with the size of the system (by splitting redundant sets of states), but no experimental results are provided. Finally, Henzinger et al. [26] translates BAs into a variant of the equational μ-calculus based on the post-image operator. Though it can be considered as an application of property-driven partitioning, the attention is focused on the forward nature of the approach.

[1] See `www-cad.eecs.berkeley.edu/~kenmcmil/smv/`.

For branching-time logics, symbolic procedures Burch et al. [4] verify a CTL formula by computing, for every subformula, the states of the system that satisfy it. The automata-theoretic counterpart of branching-time logics are automata over infinite trees. It is shown in [33] that, by translating CTL formulas into weak alternating automata, one can achieve optimal decision procedures. The "weakness" of the automata induces a partitioning on the state space of the product of the automaton with the system and a partial order on the block of the partition. The algorithm proceeds up the partial order by computing for each partition block which states of the product have a non-empty language. Since the translation produces automata with one state for every subformula, the standard symbolic model-checking algorithm can be viewed as a partitioned version of this automata-theoretic approach. This, together with the automata-theoretic treatment of CTL* model checking in [33], suggests that it would be possible to apply to CTL* model checking a combination of the standard property-driven partitioning for CTL [4] with the property-driven partitioning for LTL described here.

The paper begins with an overview of the basic notions of SMC [12] and GSTE [49] in Sect. 2: first, BAs and AGs are defined in a new perspective that clarifies the common underlying structure; we then describe SMC and GSTE model checking procedures. In Sect. 3, first, we prove that AGs and BAs are equivalent; then, we analyze the checking algorithms of GSTE and show that they are partitioned versions of standard SMC algorithms. In Sect. 4, we export property-driven partitioning to SMC and we report on the comparison of SMC with partitioned SMC in the framework of NuSMV. We conclude in Sect. 5 with a discussion of future research directions.

2 Büchi automata and assertion graphs

In this section, we introduce the specification languages and the checking algorithms used by SMC [12] and GSTE [49]. In SMC, we can specify properties by means of BAs, while GSTE uses AGs. Both the languages have a finite and a fair semantics. The finite semantics is checked with a fixpoint computation, while the fair one requires a doubly-nested fixpoint computation.

We define a system M as a tuple $\langle S, S_I, T\rangle$, where S is the set of states, $S_I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation. We use capital letters such as $Y, Z, \ldots$, to denote subsets of S. We define functions $post, pre : 2^S \longrightarrow 2^S$ such that $post(Y) = \{s' \in S \mid (s, s') \in T, s \in Y\}$ and $pre(Y) = \{s' \in S \mid (s', s) \in T, s \in Y\}$. A finite (resp., infinite) trace in M is a finite (resp., infinite) sequence σ of states such that $\sigma[i+1] \in post(\sigma[i])$ for all $1 \leq i < |\sigma|$ (resp., $i \geq 1$). A trace σ is initial iff $\sigma(1) \in S_I$. We define $L_f(M)$ as the set of all initial finite traces of M and $L(M)$ as the set of all initial infinite traces. We define S^* (resp. S^ω) as the set of all finite (resp., infinite) sequences of states in M. (Thus, $L_f(M) \subseteq S^*$ and $L(M) \subseteq S^\omega$.)

In the following, we propose a new representation for BAs and AGs: both can be seen as an extension of Fair Graphs (FG). This is the structure which AGs and BAs have in common. As we shall see, while an AG is an FG with two labeling functions, a BA is an FG with just one labeling function. We use labels on vertices rather than on edges (as in GSTE [49]). This does not affect the generality of our framework and allows for an easier comparison between GSTE and SMC as well as an experimental evaluation in the framework of NuSMV. Moreover, labels are defined as sets of system's states. (In practice, labels are given as predicates on system states; a predicate describes the set of states that satisfy it.)

2.1 Fair Graphs, Büchi automata and assertion graphs

Fair Graphs are essentially graphs with the addition of a fairness condition.

Definition 1 A Fair Graph G is a tuple $\langle V, V_I, E, \mathcal{F}\rangle$ where V is the set of vertices, $V_I \subseteq V$ is the set of initial vertices, $E \subseteq V \times V$ is a total relation representing the set of edges, and $\mathcal{F} = \{F_1, \ldots, F_n\}$, with $F_j \subseteq V$ for $1 \leq j \leq n$, is the set of fair sets.

A finite (resp., infinite) path in G is a finite (resp., infinite) sequence ρ of vertices such that $(\rho[i], \rho[i+1]) \in E$ for all $1 \leq i < |\rho|$ (resp., $i \geq 1$). ρ is initial iff $\rho[1] \in V_I$. ρ is fair iff it visits every set $F \in \mathcal{F}$ infinitely often. If $\mathcal{F} = \{F\}$, we define $L_f(G)$ as the set of all finite initial paths of G whose last state belongs to F. We define $L(G)$ as the set of all fair initial paths. We say that a path of G is accepting if it belongs to $L_f(G)$ or to $L(G)$.

For every $v \in V$ we define the set of successor vertices $E(v) = \{v' \in V \mid (v, v') \in E\}$ and the set of predecessor vertices $E^-(v) = \{v' \in V \mid v(v', v) \in E\}$. (The operators E and E^- are analogous to *post* and *pre*. They are used for clarity of notation.)

A labeling function is a function $\gamma : V \longrightarrow 2^S$. Given a set of vertices $V' \subseteq V$, we define the restriction $\gamma_{|_{V'}}$ of γ to V' as follows: $\gamma_{|_{V'}}(v) = \gamma(v)$ if $v \in V'$, and $\gamma_{|_{V'}}(v) = \emptyset$ otherwise. Typically, we use α, β, γ to denote labeling functions. Notice that a labeling function γ can be considered and represented as a set of subsets of S: $\{\gamma(v)\}_{v \in V}$. With abuse of notation, given two labeling functions α and γ, we will write $\alpha \subseteq \gamma$ (resp., $\alpha \cap \gamma$, $\alpha \cup \gamma$) to mean, for all $v \in V$, $\alpha(v) \subseteq \gamma(v)$ (resp., $\alpha(v) \cap \gamma(v)$, $\alpha(v) \cup \gamma(v)$).

Definition 2 Given a finite (resp. infinite) sequence of states σ in M, a path ρ in G of the same length l (resp., both infinite) and a function $\gamma : V \longrightarrow 2^S$, we say that σ satisfies ρ under γ (denoted $\sigma \models_\gamma \rho$) iff $\sigma[i] \in \gamma(\rho[i])$ for all $1 \leq i \leq l$ (resp., $i \geq 1$).

A Büchi automaton (BA) is essentially an FG with the addition of a labeling function. A sequence of states is accepted by a BA iff it satisfies the labeling function along at least one accepting path of the FG. In the following, BAs express complementary properties, that is, their language contains all violations of the desired property.

Formally, a Büchi automaton B is a tuple $\langle G, \mathcal{L}\rangle$ where $G = \langle V, V_I, E, \mathcal{F}\rangle$ is a fair graph, and $\mathcal{L} : V \longrightarrow 2^S$ is the labeling function. We define the set $L_f(B)$ (resp., $L(B)$) as the set of finite (resp., infinite) sequences of states of M accepted by B:

Definition 3

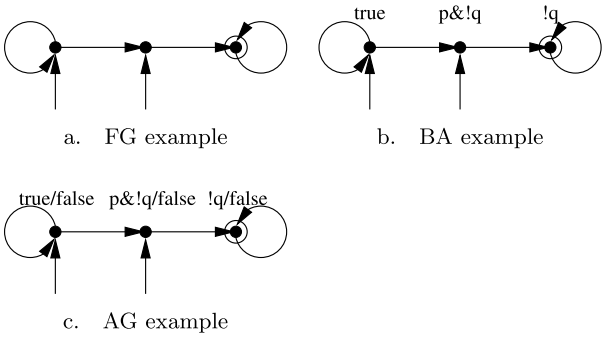
- *Finite semantics*: if $\mathcal{F} = \{F\}$, $L_f(B) = \{\sigma \in S^* \mid$ there exists a finite path $\rho \in L_f(G)$ with $|\sigma| = |\rho| = l$, $\rho[l] \in F$ and $\sigma \models_{\mathcal{L}} \rho\}$;
- *Fair semantics*: $L(B) = \{\sigma \in S^\omega \mid$ there exists a fair path $\rho \in L(G)$ with $\sigma \models_{\mathcal{L}} \rho\}$.

Since a BA has the complementary language of the specification, the model checking problem consists of verifying whether $L_f(M) \cap L_f(B) = \emptyset$, in the case of finite semantics, $L(M) \cap L(B) = \emptyset$, in the case of fair semantics.

An Assertion Graph (AG) is essentially an FG with the addition of two labeling functions: the antecedent and the consequent. An AG accepts a sequence of states iff, along all accepting paths, either the sequence does not satisfy the antecedent or it satisfies the consequent.

Formally, an Assertion Graph A is a tuple $\langle G, ant, cons\rangle$ where $G = \langle V, V_I, E, \mathcal{F}\rangle$ is a fair graph, $ant : V \longrightarrow 2^S$ is the antecedent function, and $cons : V \longrightarrow 2^S$ is the consequent

Fig. 1



a. FG example b. BA example

c. AG example

function. Given a sequence of states σ in M and a path ρ in G of the same length, we say that σ satisfies ρ in A (denoted $\sigma \models_A \rho$) iff $\sigma \models_{ant} \rho \Rightarrow \sigma \models_{cons} \rho$. We define the set $L_f(A)$ (resp., $L(A)$) as the set of finite (resp., infinite) sequences of states of M accepted by A:

Definition 4

- *Finite semantics*: [2] if $\mathcal{F} = \{F\}$, $L_f(A) = \{\sigma \in S^* \mid$ for every finite path $\rho \in L_f(G)$, if $|\sigma| = |\rho| = l$ and $\rho[l] \in F$ for some l, then $\sigma \models_A \rho\}$;
- *Fair semantics*: $L(A) = \{\sigma \in S^\omega \mid$ for every fair path $\rho \in L(G)$, $\sigma \models_A \rho\}$.

The model checking problem for an AG consists of verifying whether $L_f(M) \subseteq L_f(A)$, in the case of finite semantics, $L(M) \subseteq L(A)$, in the case of fair semantics.

Example 1 An example of FG is depicted in Fig. 1a. The vertices are represented by points, the edges by arrows. An arrow without the starting vertex points to a vertex to indicate that it is initial. For simplicity, in the example we have only one fair set. The circle around the rightmost vertex means that it belongs to this fair set.

Examples of BA and AG are depicted resp. in Figs. 1b and 1c. They have the same underlying FG. In the AG, the labels are represented in the format *ant*/*cons*. p and q are propositional properties. With the fair semantics, the AG corresponds to the LTL property $G(p \rightarrow Fq)$, while the BA has the complementary language.

As we will see in Sect. 3, given a BA, one can easily construct an AG for the complementary language, and vice versa.

2.2 SMC algorithms

Given a system $M = \langle S, S_I, T\rangle$ and a BA $B = \langle\langle V, V_I, E, \mathcal{F}\rangle, \mathcal{L}\rangle$, SMC first computes the product P between B and M. Then, in the case of finite semantics, it finds the set of vertices reachable from the initial vertices and checks if it intersects a certain set of vertices F_P in P; in the case of fair semantics it finds the set of fair vertices, i.e., those which are extensible to fair paths, and it checks if it intersects the set of initial vertices.

The product between M and B is a BA defined as follows: $P = \langle\langle V_P, I_P, E_P, \mathcal{F}_P\rangle, \mathcal{L}_P\rangle$ where $V_P = \{(s, v) \mid s \in M, v \in V, s \in \mathcal{L}(v)\}$, $I_P = \{(s, v) \in V_P \mid s \in S_I, v \in V_I\}$, $E_P =$

[2] In [29] the finite semantics is called *terminal*. Moreover, the authors, as in [49], define a third and a fourth semantics called *strong* and *infinite*, which we ignore in this paper.

Fig. 2

Algorithm *traversal*(P)
1. $R := I_P$
2. $N := I_P$
3. **repeat**
4. $\quad Z := \mathbf{EY}[N]$
5. $\quad N := Z \backslash R$
6. $\quad R := R \cup Z$
7. **until** $N = \emptyset$
8. **return** R

Fig. 3

Algorithm *fairstates*(P)
1. $Y := V_P$
2. **repeat**
3. $\quad Y' := Y$
4. $\quad$ **for** $F_P \in \mathcal{F}_P$
5. $\quad\quad Z := \mathbf{E}[Y \mathbf{U} (Y \wedge F_P)]$
6. $\quad\quad Y := Y \wedge \mathbf{EX}[Z]$
7. **until** $Y' = Y$
8. **return** Y

$\{((s, v), (s', v')) \mid (s, v) \in V_P, (s', v') \in V_P, (s, s') \in T, (v, v') \in E\}$, $\mathcal{F}_P = \{F_{P1}, \ldots, F_{Pn}\}$ where $F_{Pj} = \{(s, v) \in V_P \mid v \in F_j\}$, $\mathcal{L}_P(s, v) = \{s\}$.

In the case of finite semantics $\mathcal{F} = \{F\}$, so that $\mathcal{F}_P = \{F_P\}$, where $F_P = \{(s, v) \in V_P \mid v \in F\}$. Then, it is easy to see that $L_f(P) = L_f(M) \cap L_f(B)$. Moreover, every finite path of P corresponds to a finite trace of M accepted by B. Thus, to verify that $L_f(P) = \emptyset$, we can just compute the set of reachable vertices and check that it does not intersect F_P. Usually, this set is found with a traversal algorithm like the one described in Fig. 2: starting from the initial states, it applies the *post* operation (denoted here with **EY**) until a fixpoint is reached.

Similarly, in the case of fair semantics, it is easy to see that $L(P) = L(M) \cap L(B)$. Moreover, every fair path of P corresponds to an infinite trace of M accepted by B. Thus, to verify that $L(P) = \emptyset$ we can just compute the set of fair vertices and check that it does not intersect I_P. The standard algorithm to compute the set of fair vertices is the Emerson-Lei algorithm (EL) described in Fig. 3 [16, 17]: the set is iteratively approximated with the set of vertices that can reach each fairness condition; this is computed with the *pre* operator (denoted here with **EX**) and a backward traversal (**EU**). SMC tools typically implement variants of this doubly-nested fixpoint computation, cf. [18, 39].

2.3 GSTE algorithms

The algorithm used by GSTE to check the AG in the different semantics is described in Fig. 4. The function *GSTE_fairstates* of line 2 is called only in the case of fair semantics and it is described in Fig. 5. *GSTE_fairstates* restricts the antecedent function to the states of the system that are extensible to fair paths. In the lines 3–9 of Fig. 4, α is defined iteratively until a fixpoint is reached. First, α is initialized to be the restriction of *ant* to the set of initial vertices and to the set of initial states. Then, at every iteration, a state s is added to $\alpha(v)$ iff $s \in ant(v)$ and there exists a state $s' \in \alpha(v')$ such that s is reachable from s' in one step and v is reachable from v' in one step. When the fixpoint is reached, $\alpha(v)$ contains s iff there

Algorithm *GSTE*(M, A)
1. **if** fair semantics
2. **then** $A := GSTE\_fairstates(M, A)$
3. $\alpha := ant_{|V_I}$
4. **for** $v \in V$, $\alpha(v) := \alpha(v) \cap S_I$
5 **repeat**
6. $\alpha' := \alpha$
7. **for** $v \in V$, $\alpha(v) := \alpha'(v) \cup \bigcup_{v' \in E^-(v)} post(\alpha'(v')) \cap ant(v)$
8. **until** $\alpha' = \alpha$
9. **if** fair semantics
10. **then return** $\alpha \subseteq cons$
11. **else return** $\alpha_{|_F} \subseteq cons$

Fig. 4

Algorithm *GSTE_fairstates*(M, A)
1. **repeat**
2. $ant' := ant$
3. **for** $F \in \mathcal{F}$,
4. **for** $v \in V$, $\alpha(v) := \bigcup_{v' \in E(v), v' \in F} pre(ant(v')) \cap ant(v)$
5. **repeat**
6. $\alpha' := \alpha$
7. **for** $v \in V$, $\alpha(v) := \alpha'(v) \cup \bigcup_{v' \in E(v)} pre(\alpha'(v')) \cap ant(v)$
8. **until** $\alpha' = \alpha$
9. $ant := \alpha$
10. **until** $ant' = ant$
11. **return** A

Fig. 5

exists an initial path ρ of the assertion graph and an initial trace σ of the system of the same length l such that $\rho[l] = v$, $\sigma[l] = s$ and $\sigma \models_{ant} \rho$.

With an analogous fixpoint computation (lines 4–8), *GSTE_fairstates* finds a function α such that $\alpha(v)$ contains s iff there exist a path ρ of the assertion graph and a trace σ of the system of the same length l such that $\rho[l] \in F$, $\rho[1] = v$, $\sigma[1] = s$ and $\sigma \models_{ant} \rho$. This computation is applied for every $F \in \mathcal{F}$ and it is nested in a second fixpoint computation: at every iteration the antecedent function is updated with α until a fixpoint is reached. At the end of the outer loop, $ant(v)$ contains s iff there exist a fair path ρ of the assertion graph and an infinite trace σ of the system such that $\sigma \models_{ant} \rho$.

3 GSTE vs. SMC

In this section, we clarify the relationship between GSTE and SMC. First, we show that AGs and BAs are equivalent. Then, we show that the GSTE algorithm is essentially a "partitioned" version of the SMC algorithm.

We now show that, given a BA B, one can easily find an AG A with the complementary language and vice versa. This means that, given a specification φ, one can choose either GSTE or SMC techniques to check φ, no matters whether φ is an AG or a BA. Moreover,

since BAs are nondeterministic (i.e., existential) automata, AGs are revealed to be their dual, which are universal automata.

The following four theorems establish the relationship between AGs and BAs: see Appendix for the proofs. First, the following two theorems show how to express AGs as BAs. Intuitively, the state space of an AG is multiplied by a counter: the value 0 of the counter corresponds to a finite path that satisfies the antecedent; the value 1 represents a point in which the antecedent is satisfied but the consequent is violated; the final value 2 corresponds to an accepting suffix that satisfies the antecedent.

Theorem 1 *Let $A = \langle G, ant, cons\rangle$ be an AG where $G = \langle V, V_I, E, \mathcal{F}\rangle$ and $\mathcal{F} = \{F\}$. Let B be the BA $\langle G', \mathcal{L}\rangle$, where $G' = \langle V', V'_I, E', \mathcal{F}'\rangle$ s.t.*

- $V' = V \times \{0, 1, 2\}$,
- $V'_I = V_I \times \{0, 1\}$,
- $E' = \{((v_1, k_1), (v_2, k_2)) \mid (v_1, v_2) \in E, k_2 \in \{0, 1\}$ *if* $k_1 = 0$, *and* $k_2 = 2$ *otherwise*$\}$,
- $\mathcal{F}' = \{F \times \{1, 2\}\}$,

$\mathcal{L}((v, k)) = ant(v)$ *if* $k \in \{0, 2\}$, *and* $\mathcal{L}((v, k)) = ant(v) \cap (S \backslash cons(v))$ *if* $k = 1$. *Then* $L_f(B) = S^* \backslash L_f(A)$.

Theorem 2 *Let $A = \langle G, ant, cons\rangle$ be an AG where $G = \langle V, V_I, E, \mathcal{F}\rangle$ and $\mathcal{F} = \{F_1, \ldots, F_n\}$. Let B be the BA $\langle G', \mathcal{L}\rangle$, where $G' = \langle V', V'_I, E', \mathcal{F}'\rangle$ s.t.*

- $V' = V \times \{0, 1, 2\}$,
- $V'_I = V_I \times \{0, 1\}$,
- $E' = \{((v_1, k_1), (v_2, k_2)) \mid (v_1, v_2) \in E, k_2 \in \{0, 1\}$ *if* $k_1 = 0$, *and* $k_2 = 2$ *otherwise*$\}$,
- $\mathcal{F}' = \{F_1 \times \{2\}, \ldots, F_n \times \{2\}\}$,

$\mathcal{L}((v, k)) = ant(v)$ *if* $k \in \{0, 2\}$, *and* $\mathcal{L}((v, k)) = ant(v) \cap (S \backslash cons(v))$ *if* $k = 1$. *Then* $L(B) = S^\omega \backslash L(A)$.

The following two theorems show how to express BAs as AGs.

Theorem 3 *Let $B = \langle G, \mathcal{L}\rangle$ be a BA. Let A be the AG $\langle G, ant, cons\rangle$, where $ant = \mathcal{L}$, $cons(v) = \emptyset$ for all $v \in V$. Then $L_f(B) = S^* \backslash L_f(A)$.*

Theorem 4 *Let $B = \langle G, \mathcal{L}\rangle$ be a BA. Let A be the AG $\langle G, ant, cons\rangle$, where $ant = \mathcal{L}$, $cons(v) = \emptyset$ for all $v \in V$. Then $L(B) = S^\omega \backslash L(A)$.*

We now compare the algorithms used by GSTE and SMC. In particular, we show that the former is essentially a "partitioned" version of the latter.

In Sect. 2, we saw how SMC solves the model checking problem for a BA B: it builds the product automaton P between M and B and it verifies that the language of P is empty. GSTE follows an analogous procedure for checking an AG A: it actually computes the product between M and B_{ant}, where B_{ant} is a BA with the same underlying graph G of A and the labeling function equal to ant. The only difference between SMC and GSTE is that the latter operates on partitioned subsets of the product state space. The partitioning is driven by the automaton state space and we refer to such partitioning as *property-driven partitioning*. The GSTE analog of a subset $Q \subseteq S_P$ is the partition $\{Q_v : v \in V\}$, where $Q_v = \{s : (s, v) \in S_P\}$. Indeed, every labeling function γ can be seen as a division of the model into sets of states, one for every vertex v of the graph, which is exactly the set $\gamma(v)$.

If $\gamma \subseteq ant$, then γ turns out to represent a set $S_\gamma \subseteq S_P$ of states in the product defined as follows: $S_\gamma = \{(s, v) | s \in \gamma(v)\}$.

One can see that the lines 3–9 of the algorithm in Fig. 4 computes the reachable states of S_P. In fact, we could rewrite lines 6–7 in terms of CTL formulas as $\alpha = \alpha \cup \mathbf{EY}[\alpha]$. Thus, at the end of the loop, $\alpha(v) = \{s | (s, v)$ is reachable in $S_P\}$. This computation is actually a partitioned version of the one of Fig. 2 with the difference that SMC applies the post-image only to the new states added in the previous iteration, while GSTE applies the post-image to the whole set of reached states.

In the case of fair semantics the computation of reachable states is preceded by a pruning of the product: *GSTE_fairstates* finds all vertices of S_P such that they are extensible to fair paths. To compare this procedure with EL, we rewrite the operations of *GSTE_fairstates* in terms of CTL formulas. At the line 4 of the algorithm in Fig. 5, *GSTE_fairstates* actually computes the preimage of $ant_{|_F}$ (seen as a set of states in S_P). So, we can rewrite this line as $\alpha = ant \cap \mathbf{EX}[(ant_{|_F})]$. Furthermore, the lines 6–7 are the same as $\alpha = \alpha \cup (ant \cap \mathbf{EX}[(\alpha)])$ so that one can see the loop of lines 5–8 as $\alpha = \mathbf{E}[(ant)\mathbf{U}(\alpha)]$. This reachability computation is nested in a second fixpoint computation, so that it becomes evident that *GSTE_fairstates* is a variant of the EL algorithm of Fig. 3.

4 SMC vs. property-driven partitioned SMC

In Sect. 3, we saw that GSTE is a partitioned version of SMC. We can also apply property-driven partitioning to standard SMC algorithms. In particular, there are two algorithms to be partitioned: *traversal* and *fairstates* (Figs. 2 and 3). We partitioned both of them, by using NuSMV as a platform. This choice is motivated by the fact that NuSMV implements symbolic model checking for LTL, its source is open, and its code is well-documented and easy to modify.

The "translated" algorithms are shown in Figs. 6 and 7. Both are based on backward reachability and respect the structure of NuSMV's implementation (e.g., the order of fair sets is irrelevant). The difference with the non-partitioned versions is that while *traversal* and *fairstates* operate on a single set of states in the product automaton, *partitioned_traversal* and *partitioned_fairstates* operate on an array of sets of states of the system (one set for every vertex of the BA). Thus, every variable in the algorithms of Figs. 6 and 7 can be considered as a labeling function. For every set $Y \subseteq S$ of states and labeling $\mathcal{L}$, we define the labeling function $par_{\mathcal{L}}(Y)$ such that: $par_{\mathcal{L}}(Y)(v) = Y \cap \mathcal{L}(v)$ for all $v \in V$. The initial states of the product are given by $par_{\mathcal{L}}(S_I)_{|_{V_I}}$. Given a fair set F of the BA, the correspondent set in the product is given by $par_{\mathcal{L}}(S)_{|_F}$. The backward image of a labeling function α is

Fig. 6

Algorithm *partitioned_traversal*(M, B)
1. $\alpha := par_{\mathcal{L}}(S)_{|_F}$
2. $\beta := \alpha$
3. **repeat**
4. $\quad \gamma := \mathbf{EX}[\beta]$
5. $\quad \beta = \gamma \backslash \alpha$
6. $\quad \alpha := \alpha \cup \gamma$
7. **until** $\beta = \emptyset$
8. **return** α

Fig. 7

Algorithm *partitioned_fairstates*(M, B)
1. $\alpha := \top$;
2. **repeat**
3. $\quad \alpha' := \alpha$;
4. $\quad \beta := \top$;
5. $\quad$ **for** $F \in \mathcal{F}$
6. $\quad\quad \beta := \beta \cap \mathbf{E}[\alpha \mathbf{U}(\alpha \cap par_{\mathcal{L}}(S)_{|_F})]$;
7. $\quad \alpha := \alpha \cap \beta$;
8. $\quad \alpha := \alpha \cap \mathbf{EX}[\alpha]$;
9. **until** $\alpha' = \alpha$
10. **return** α

given by

$$\mathbf{EX}[(\alpha)](v) = \bigcup_{v' \in E(v)} pre(\alpha(v')) \cap \mathcal{L}(v).$$

We investigated if property-driven partitioning is effective for symbolic model checking. In particular, we applied the technique to LTL model checking. In fact, it is well known that, given a formula φ expressed by an LTL formula, we can find a BA with the same language. The standard LTL symbolic model checkers translate the negation of the specification into a BA, they add the latter to the model and check for emptiness. The goal of our experiments was to compare the performance of partitioned and non-partitioned SMC algorithms. Thus, we did not try to optimize the algorithms implemented in NuSMV, but to apply to them property-driven partitioning. The question we wanted to answer is whether the reduction in BDD size more than compensates for the algorithmic overhead involved in handling a partitioned state-space. This also provides an indirect comparison between GSTE and standard SMC techniques.

To verify an LTL formula φ, NuSMV calls `ltl2smv`, which translates $\neg\varphi$ into a symbolically represented BA with fairness constraints $\mathcal{F}$. Then, the function $\mathbf{E}_{\mathcal{F}}\mathbf{G}[true]$ checks if the language of the product is empty. Since NuSMV does not apply any particular technique when φ is a safety formula [31], we enhanced the tool with the option `-safety`: when φ contains only the temporal connectives X, G, and V, it constructs a predicate F on the automaton states (representing accepting states for the complementary property) and calls the function $\mathbf{E}[true\mathbf{U}F]$. In the following, we refer to this procedure and to the standard NuSMV's procedure as "`NuSMV -safety`" and "`NuSMV`" respectively. We implemented the partitioned versions of both and we refer to latter ones as "`NuSMV -safety -partitioned`" and "`NuSMV -partitioned`" respectively. The BA is built automatically by `ltl2smv` in the case of non-partitioned algorithms while it is constructed by hand (in these experiments) in the case of partitioned algorithms.

When NuSMV builds the product between the property automaton and the system, it appends the symbolic variables of the property automaton at the bottom of the variable ordering. We added an option to NuSMV, "`-topencode`", in order to put such variables at the top. With this change, we obtain a symbolic version of property-driven partitioning: if indeed you have a BDD that corresponds to a subset Q of the product and you follow an assignment to the symbolic variables of the property automaton that corresponds to a vertex v, then the BDD node you obtain is exactly the partition Q_v of Q.

We ran our tests on three examples of SMV models (for the SMV code, visit `www.science.unitn.it/~stonetta/partitioning.html`). For every example, we chose two properties true in the model (one safety and one liveness property, see

Table 1 Satisfied properties

	Safety	Liveness
Dining	$G((p \wedge r \wedge X(r) \wedge XX(r) \wedge X^3(r)) \rightarrow X^4(e))$	$(\bigwedge_{1\leq i\leq N} GFr_i) \rightarrow (GFs)$
Mutex	$G((t_1 \wedge \bigwedge_{2\leq i\leq N} \neg t_i) \rightarrow Xc)$	$G(\bigwedge_{1\leq 1\leq N} t_i \rightarrow Fc_i)$
Life	$G(b \rightarrow Xc)$	$G((G!b) \rightarrow FG(d))$

Table 2 Failed properties

	Safety	Liveness
Dining	$G((p \wedge r \wedge X(r) \wedge XX(r) \wedge X^3(r)) \rightarrow X^4(\neg e))$	$(GFr_1) \rightarrow (GFe_1)$
Mutex	$G((t_1 \wedge \bigwedge_{2\leq i\leq N} \neg t_i) \rightarrow X\neg c)$	$F(t_1 \rightarrow G\neg c_1)$
Life	$G(b \rightarrow X\neg c)$	$F((G!b) \wedge GF(!d))$

Table 1) and two properties that failed (again one safety and one liveness property, see Table 2). The first example is a dining-philosophers protocol [15]. Concurrency is modeled with the interleaving semantics. Typically, a philosopher iterates through a sequence of four states: she thinks, tries to pick up the chopsticks, eats and, finally, she puts down the chopsticks. When a deadlock condition happens, a philosopher puts the chopsticks down. The safety property true in this example is the following: if a philosopher is thinking and both her chopsticks are free and she is scheduled for 4 steps in a row, then she will start eating. From this property, we deduce an analogous one which fails: with the same premises, after 4 steps the philosopher does not eat. The satisfied liveness property states that if every philosopher is scheduled infinitely often, then somebody eats infinitely often (at least one philosopher does not starve). In contrast, the following liveness property does not hold in the example: if a philosopher is scheduled infinitely often, then she eats infinitely often.

The second example is a mutual-exclusion protocol: N processes non-deterministically try to access the critical session. The access is controlled by the main module, which guarantees that a process does not wait forever. The true safety property says that, if a process is the only one that is waiting, then it accesses the critical session in one step. If we change this property by writing that the process does not access the critical session in one step, we obtain the safety property that fails. The satisfied liveness property asserts that, if a process is trying, sooner or later it will access the critical session. We chose the negation of this property as an example of liveness property that fails.

Finally, the third example is a variant of the game of life: at the beginning there is only one creature; every creature has a maximum life set to 100, but it can die non-deterministically in every moment; when the age is between 15 and 65, a creature can bear a child, which is born in the next step; at most N creatures can be born; when all the creatures are dead the game is reset. The true safety property states that, if a creature is bearing a child, then the number of born creatures increases; the failed property states that the number decreases. The true liveness property asserts the following: if no creature will be born anymore, then, after a certain point in the future (likely after a reset), the number of alive creatures will be equal to one forever. The negation of this property corresponds exactly to the liveness property which failed.

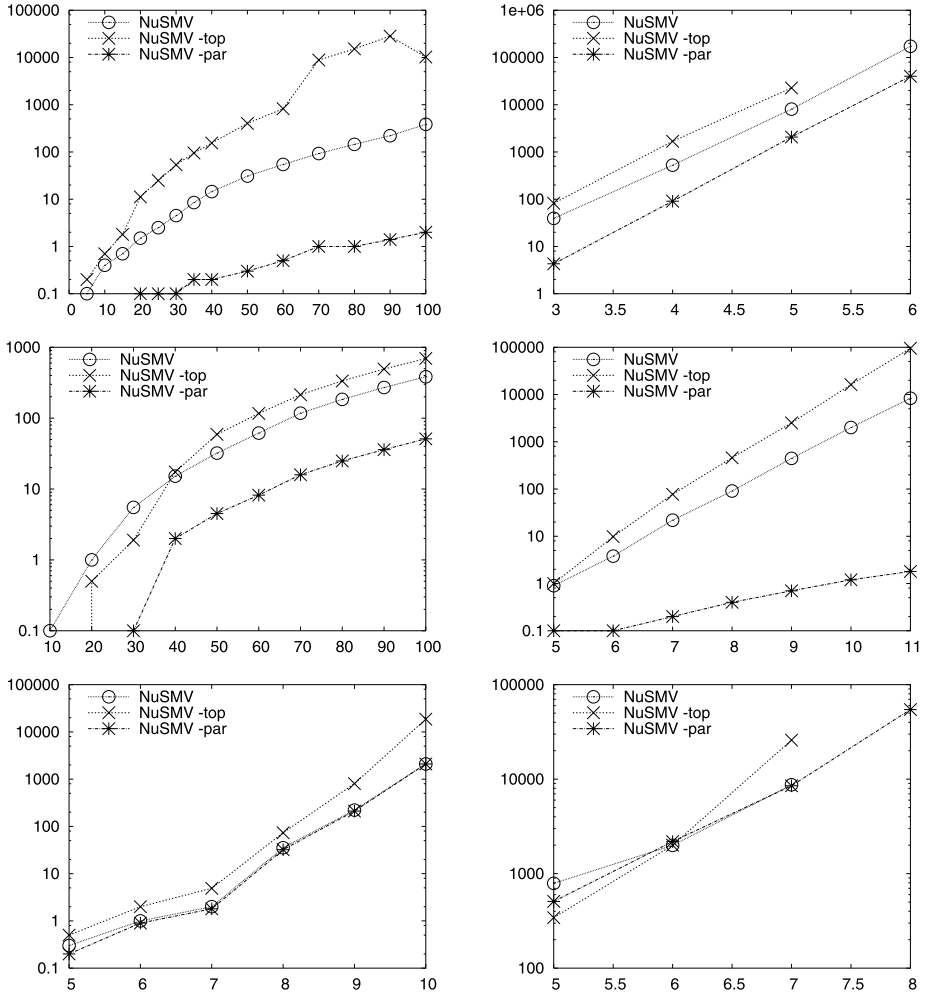


Fig. 8 Satisfied properties of Table 1. *X axis*: number of processes. *Y axis*: time. *Left column*: performances of "`NuSMV -safety`", "`NuSMV -safety -topencode`" and "`NuSMV -safety -partitioned`" on safety properties. *Right column*: performances of "`NuSMV`", "`NuSMV -topencode`" and "`NuSMV -partitioned`" on liveness properties. *1st row*: dining-philosophers example. *2nd row*: mutex example. *3rd row*: life example

We run NuSMV on the Rice Terascale Cluster (RTC),[3] a 1 TeraFLOP Linux cluster based on Intel Itanium 2 Processors. Timeout was set to 172800 seconds (two days). The results are shown in Figs. 8–11: Figs. 8 and 9 present the execution time in seconds, while Figs. 10 and 11 present the number of allocated BDD nodes; both quantities are plotted in log scale against the number N of processes in the model. Every example takes a row of plots. We plotted safety property in the first column and liveness properties in the second column. First compare the partitioned version with the non-partitioned one with regard to

[3] See `www.citi.rice.edu/rtc/`.

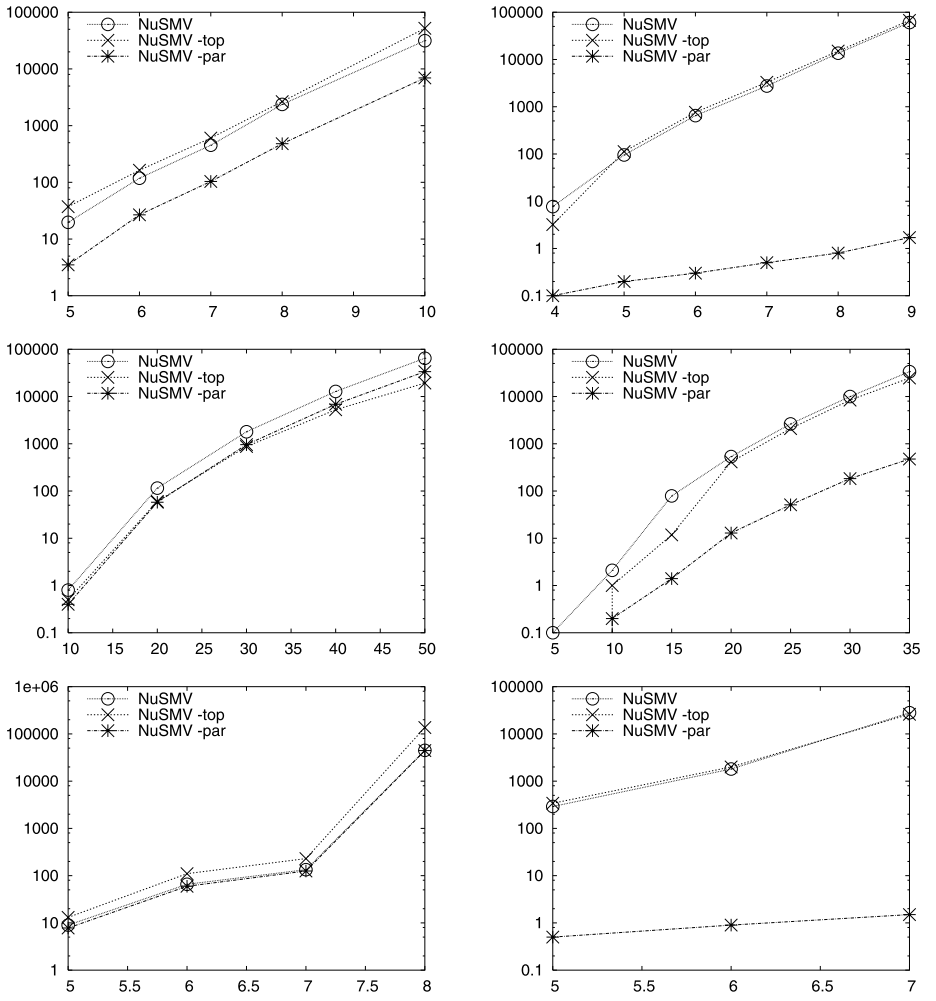


Fig. 9 Same pattern as in Fig. 8 but with the failed properties of Table 2

the verification time. As for satisfied properties (Fig. 8), we notice that, in the first two rows (dining philosophers and mutual exclusion), the former outperforms the latter. Moreover, in the case of the safety property for dining philosophers and the liveness property for mutual exclusion, the gap is exponential, i.e. the ratio between the two execution times grows exponentially with the size of the model. In the third row (life), NuSMV does not seem to get relevant benefit from the property-driven partitioning (even if you should notice that, in the last point of the liveness case, "NuSMV" runs out of time). Similarly, in the case of failed properties, the partitioned version always outperforms the non-partitioned one (see Fig. 9). Moreover, in the case of liveness properties, the improvement is exponential for all three examples.

In Figs. 10 and 11, we can compare the amount of memory required by the different versions of NuSMV. Again, partitioning generally reduces memory requirement, sometimes quite significantly.

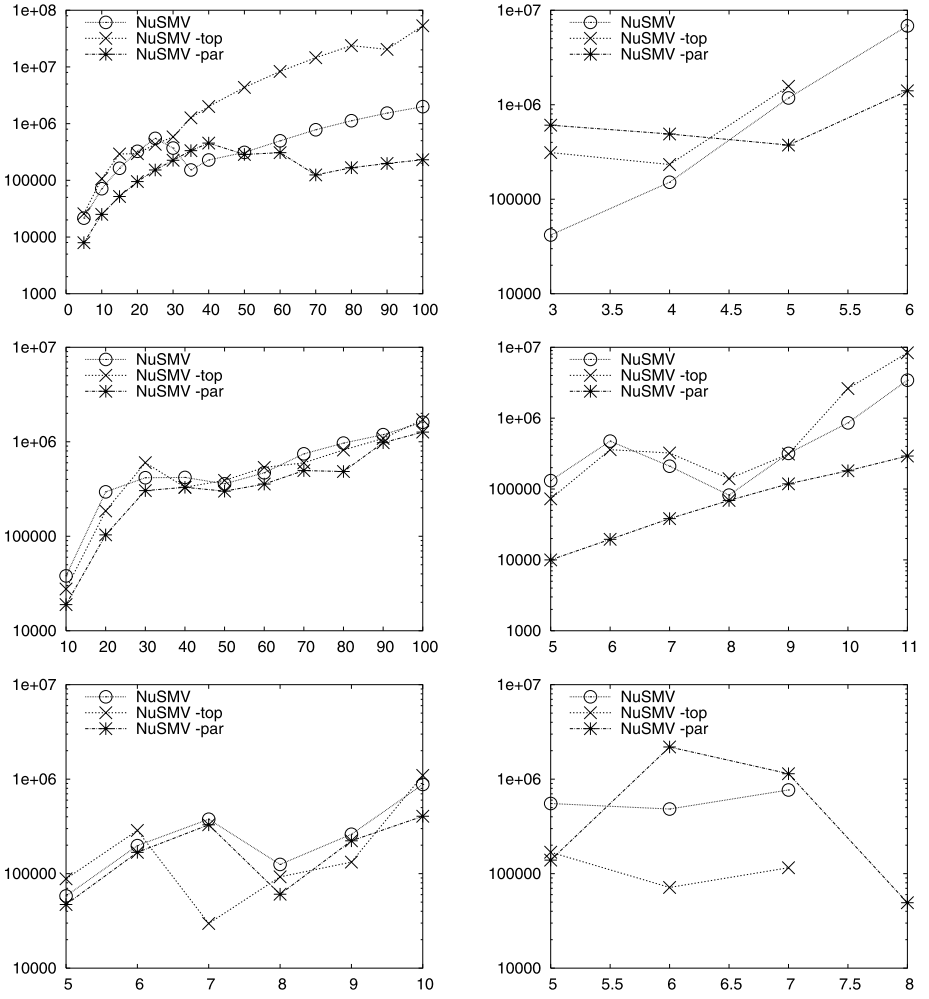


Fig. 10 Same pattern as in Fig. 8 but with the number of allocated BDD nodes on the Y axis

Finally, comparing the effect of "`-topencode`" on NuSMV, we notice that it usually worsens (as in Fig. 8) or it does not affect (as in Fig. 9) the execution time. As for space requirements, only in two cases (bottom right plots of Figs. 10 and 11) we have an evident improvement.

5 Conclusions

Our contributions in this work are two-fold. First, we elucidate the relationship between GSTE and SMC. We show that assertion graphs are simply universal automata, or, viewed dually, are nondeterministic automata for the complementary properties. Furthermore, GSTE algorithms are essentially a partitioned version of standard SMC algorithms, where the partitioning is static and is driven by the property under verification. Second, we ex-

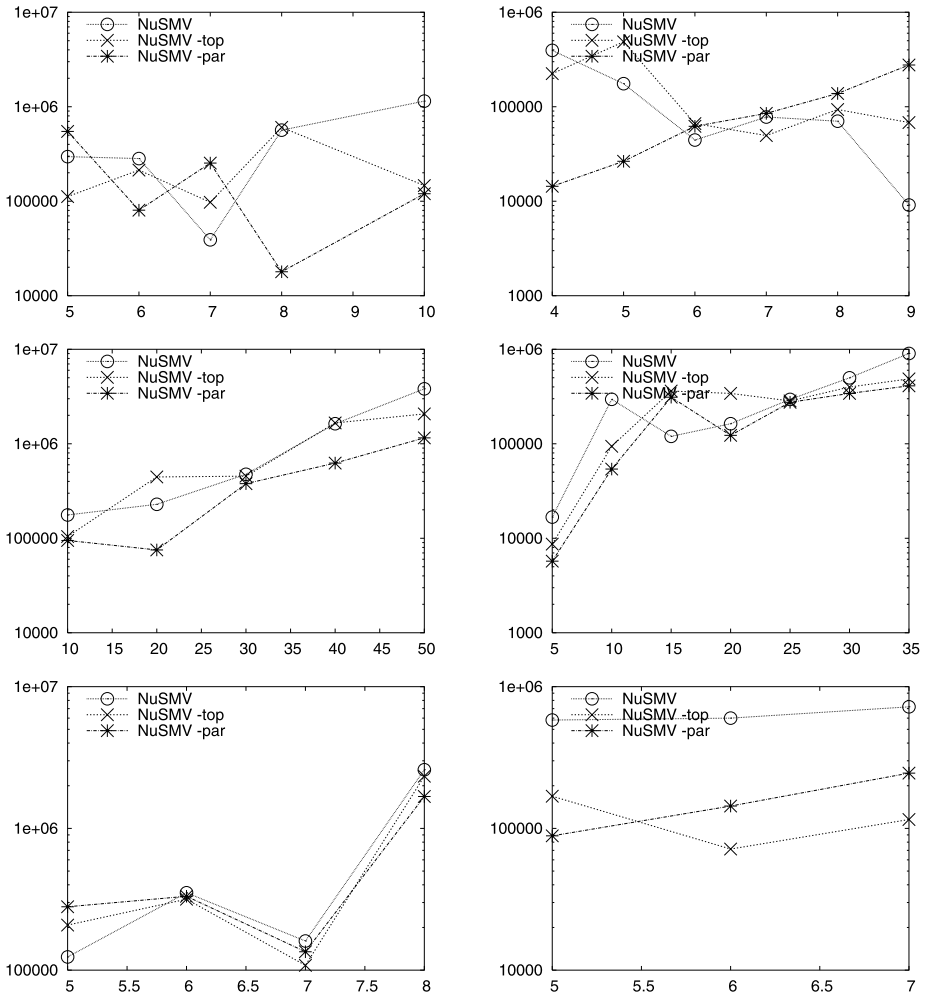


Fig. 11 Same pattern as in Fig. 10 but with the failed properties of Table 2

ported the technique of property-driven partitioning to SMC and showed its effectiveness in the framework of NuSMV.

This work opens us several directions for future work. First, we have to further investigate and understand the performance advantage of property-driven partitioning. Second, we need to combine the tool with an automated generator of explicit BAs for LTL formulas and evaluate property-driven partitioning for more complex LTL properties. Third, it requires revisiting the issue of translating LTL formulas to BAs. Previous translations have focused on making the BA smaller (cf. [14, 20, 21, 43]) or more deterministic [41]. The relative merit of the two approaches has to be investigated in the context of property-partitioned SMC. Fourth, it requires revisiting the issue of symbolic fair-cycle detection. Previous works have compared various variations of the EL algorithm, as well as non-EL algorithms, cf. [2, 18, 39]. This has to be re-evaluated for property-partitioned SMC. Finally, a major topic of research in the last few years has been that of property-driven abstraction

in model checking [11, 23]. The combination of this technique with property-driven partitioning is also worth of investigation, which could benefit from the study of abstraction in GSTE [48, 50].

Acknowledgements We are grateful to Amit Goel for highlighting the relation between property-driven partitioning and symbolic variable ordering, to Hardi Hungar for his comments on property-driven partitioning in the context of CTL model checking, and to Roberto Cavada, Alessandro Cimatti and Marco Roveri for their support with NuSMV. The fourth author would also like to thank Scott Hazelhurst for discussions about STE.

Appendix Proofs

Proof of Theorem 1 Suppose $\sigma \in S^*\backslash L_f(A)$. Then there exists $\rho \in L_f(G)$ s.t. $|\sigma| = |\rho| = l$, $\rho[l] \in F$ and $\sigma \not\models_A \rho$, i.e. $\sigma \models_{ant} \rho$ and $\sigma \not\models_{cons} \rho$. In particular, $\sigma[h] \in ant(\rho[h])$ for all $1 \leq h \leq l$ and there exists i, $1 \leq i \leq l$, s.t. $\sigma[i] \notin cons(\rho[i])$. Let $\rho' = (\rho[1], 0), \ldots, (\rho[l-1], 0), (\rho[l], 1)$ if $i = l$, $\rho' = (\rho[1], 0), \ldots, (\rho[i-1], 0), (\rho[i], 1), (\rho[i+1], 2), \ldots, (\rho[l], 2)$ otherwise. Thus, ρ' is a path of B, $\rho'[l] \in F \times \{1, 2\}$ and $\sigma \models_{\mathcal{L}} \rho'$, so that $\sigma \in L_f(B)$.

Suppose now $\sigma \in L_f(B)$. Then there exists $\rho' \in L_f(G')$ s.t. $|\sigma| = |\rho'| = l$, $\rho'(l) \in F \times \{1, 2\}$ and $\sigma \models_{\mathcal{L}} \rho$. Since $\rho'[1] \in V \times \{0, 1\}$ and $\rho'[l] \in V \times \{1, 2\}$, there must exist i, $1 \leq i \leq l$, s.t. $\rho'[i] \in V \times \{1\}$. If ρ is the projection of ρ' on the first component, we have $\sigma[h] \in ant(\rho[h])$ for all $1 \leq h \leq l$ and $\sigma[i] \notin cons(\rho[l])$. Thus, we have that $\sigma \models_{ant} \rho$ and $\sigma \not\models_{cons} \rho$, so that $\sigma \not\models_A \rho$. □

Proof of Theorem 2 Suppose $\sigma \in S^\omega\backslash L(A)$. Then there exists a fair path $\rho \in L(G)$ s.t. $\sigma \not\models_A \rho$, i.e. $\sigma \models_{ant} \rho$ and $\sigma \not\models_{cons} \rho$. In particular, $\sigma[h] \in ant(\rho[h])$ for all $h > 0$ and there exists i s.t. $\sigma[i] \notin cons(\rho[i])$. If $\rho' = (\rho[1], 0), \ldots, (\rho[i-1], 0), (\rho[i], 1), (\rho[i+1], 2), (\rho[i+2], 2), \ldots$, then ρ' is a path of B, ρ' visits infinitely often every $F_j \times \{2\} \in \mathcal{F}'$ and $\sigma \models_{\mathcal{L}} \rho'$, so that $\sigma \in L(B)$.

Suppose now $\sigma \in L(B)$. Then there exists a fair path $\rho' \in L(G')$ s.t. $\sigma \models_{\mathcal{L}} \rho$. Since ρ' starts from $V \times \{0, 1\}$ and visits $V \times \{2\}$, there exists i s.t. $\rho'(i) \in V \times \{1\}$. If ρ is the projection of ρ' on the first component, we have $\sigma[h] \in ant(\rho[h])$ for all $h > 0$ and $\sigma[i] \notin cons(\rho[i])$. Thus, we have that $\sigma \models_{ant} \rho$ and $\sigma \not\models_{cons} \rho$ so that $\sigma \not\models_A \rho$. □

Proof of Theorem 3 Suppose $\sigma \in S^*\backslash L_f(A)$. Then there exists $\rho \in L_f(G)$ s.t. $|\sigma| = |\rho| = l$, $\rho[l] \in F$ and $\sigma \not\models_A \rho$, i.e. $\sigma \models_{ant} \rho$ and $\sigma \not\models_{cons} \rho$. In particular, $\sigma \models_{\mathcal{L}} \rho$. Thus, $\sigma \in L_f(B)$.

Suppose now $\sigma \in L_f(B)$. Then there exists $\rho \in L_f(G)$ s.t. $|\sigma| = |\rho| = l$, $\rho[l] \in F$ and $\sigma \models_{\mathcal{L}} \rho$. Since $\sigma[1] \notin cons(\rho[1])$, $\sigma \not\models_{cons} \rho$. Thus, we have that $\sigma \models_{ant} \rho$ and $\sigma \not\models_{cons} \rho$ so that $\sigma \not\models_A \rho$. □

Proof of Theorem 4 Suppose $\sigma \in S^\omega\backslash L(A)$. Then there exists an initial fair path $\rho \in L(G)$ s.t. $\sigma \not\models_A \rho$, i.e. $\sigma \models_{ant} \rho$ and $\sigma \not\models_{cons} \rho$. In particular, $\sigma \models_{\mathcal{L}} \rho$. Thus, $\sigma \in L(B)$.

Suppose now $\sigma \in L(B)$. Then there exists an initial fair path $\rho \in L(G)$ s.t. $\sigma \models_{\mathcal{L}} \rho$. Since $\sigma[1] \notin cons(\rho[1])$, $\sigma \not\models_{cons} \rho$. Thus, we have that $\sigma \models_{ant} \rho$ and $\sigma \not\models_{cons} \rho$ so that $\sigma \not\models_A \rho$. □

References

1. Biere A, Clarke EM, Zhu Y (1999) Multiple state and single state tableaux for combining local and global model checking. In: Correct system design. Lecture notes in computer science, vol 1710. Springer, Berlin, pp 163–179
2. Bloem R, Gabow H, Somenzim F (2000) An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In: Proceedings of the 3rd international conference on formal methods in computer-aided design. Lecture notes in computer science, vol 1954. Springer, Berlin, pp 37–54
3. Burch J, Clarke E, Long D (1991) Symbolic model checking with partitioned transition relations. In: Proceedings of the international conference on very large scale integration. IFIP Transactions, vol A-1. North-Holland, Amsterdam, pp 49–58
4. Burch J, Clarke E, McMillan K, Dill D, Hwang L (1992) Symbolic model checking: 10^{20} states and beyond. Inf. Comput. 98(2):142–170
5. Cabodi G, Camurati P, Lavagno L, Quer S (1997) Disjunctive partitioning and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In: Proceedings of the 34th design automation conference. ACM, New York, pp 728–733
6. Cabodi G, Camurati P, Quer S (1996) Improved reachability analysis of large finite state machines. In: Proceedings of the international conference on computer-aided design. IEEE Computer Society, Los Alamitos, pp 354–360
7. Chou C-T (1999) The mathematical foundation of symbolic trajectory evaluation. In: Proceedings of the 11th international conference on computer-aided verification. Lecture notes in computer science, vol 1633. Springer, Berlin, pp 196–207
8. Cimatti A, Clarke E, Giunchiglia F, Roveri M (1999) NUSMV: a new symbolic model verifier. In: Proceedings of the 11th international conference on computer-aided verification. Lecture notes in computer science, vol 1633. Springer, Berlin, pp 495–499
9. Cimatti A, Roveri M, Bertoli P (2001) Searching powerset automata by combining explicit-state and symbolic model checking. In: Proceedings of the 7th international conference on tools and algorithms for the construction and analysis of systems. Lecture notes in computer science, vol 2031. Springer, Berlin, pp 313–327
10. Clarke E, Grumberg O, Hamaguchi K (1997) Another look at LTL model checking. Formal Methods Syst Des 10(1):47–71
11. Clarke E, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Proceedings of the 12th international conference on computer-aided verification. Lecture notes in computer science, vol 1855. Springer, Berlin, pp 154–169
12. Clarke E, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge
13. Courcoubetis C, Vardi M, Wolper P, Yannakakis M (1992) Memory-efficient algorithms for the verification of temporal properties. Formal Methods Syst Des 1(2/3):275–288
14. Daniele N, Giunchiglia F, Vardi M (1999) Improved automata generation for linear temporal logic. In: Proceedings of the 11th international conference on computer-aided verification. Lecture notes in computer science, vol 1633. Springer, Berlin, pp 249–260
15. Dijkstra E (1972) Hierarchical ordering of sequential processes, operating systems techniques. Academic, New York
16. Emerson E, Lei C (1986) Efficient model checking in fragments of the propositional μ-calculus. In: Proceedings of the symposium on logic in computer science. IEEE Computer Society, Los Alamitos, pp 267–278
17. Emerson E, Lei C-L (1985) Temporal model checking under generalized fairness constraints. In: Proceedings of the 18th international conference on system sciences. Western Periodicals Company, pp 277–288
18. Fisler K, Fraer R, Kamhi G, Vardi M, Yang Z (2001) Is there a best symbolic cycle-detection algorithm? In: Proceeding of the 7th international conference on tools and algorithms for the construction and analysis of systems. Lecture notes in computer science, vol 2031. Springer, Berlin, pp 420–434
19. Fraer R, Kamhi G, Ziv B, Vardi M, Fix L (2000) Prioritized traversal: efficient reachability analysis for verification and falsification. In: Proceeding of the 12th international conference on computer-aided verification. Lecture notes in computer science, vol 1855. Springer, Berlin, pp 389–402
20. Fritz C (2003) Constructing Büchi Automata from linear temporal logic using simulation relations for alternating Büchi automata. In: Proceedings of the 8th international conference on implementation and application of automata. Lecture notes in computer science, vol 2759. Springer, Berlin, pp 35–48
21. Gerth R, Peled D, Vardi M, Wolper P (1995) Simple on-the-fly automatic verification of linear temporal logic. In: Proceedings of the 15th international symposium on protocol specification, testing and verification, Warsaw, Poland. IFIP, vol. 38. Chapman & Hall, London, pp 3–18

22. Goel A, Bryant RE (2003) Set manipulation with Boolean functional vectors for symbolic reachability analysis. In: Proceedings of the 6th conference on design, automation and test in Europe, Munich, Germany. IEEE Computer Society, Los Alamitos, pp 10816–10821
23. Govindaraju S, Dill D (2000) Counterexample-guided choice of projections in approximate symbolic model checking. In: Proceedings of the international conference on computer-aided design. IEEE, New York, pp 115–119
24. Grumberg O, Heyman T, Schuster A (2003) A work-efficient distributed algorithm for reachability analysis. In: Proceedings of the 15th international conference on computer-aided verification. Lecture notes in computer science, vol 2725. Springer, Berlin, pp 54–66
25. Hardin R, Har'el Z, Kurshan R (1996) COSPAN. In: Proceedings 8th international conference on computer-aided verification. Lecture notes in computer science, vol 1102. Springer, Berlin, pp 423–427
26. Henzinger T, Kupferman O, Qadeer S (2003) From pre-historic to post-modern symbolic model checking. Formal Methods Syst Des 23(3)
27. Heyman T, Geist D, Grumberg O, Schuster A (2002) A scalable parallel algorithm for reachability analysis of very large circuits. Formal Methods Syst Des 21(3):317–338
28. Holzmann G (2003) The SPIN model checker: primer and reference manual. Addison-Wesley, Reading
29. Hu A, Casas J, Yang J (2003) Reasoning about GSTE assertion graphs. In: Proceedings of the conference on correct hardware design and verification methods. Lecture notes in computer science, vol 2860. Springer, Berlin, pp 170–184
30. Iyer S, Sahoo D, Stangier C, Narayan A, Jain J (2003) Improved Symbolic verification using partitioning techniques. In: Proceedings of the conference on correct hardware design and verification methods. Lecture notes in computer science, vol 2860. Springer, Berlin, pp 410–424
31. Kupferman O, Vardi M (2001) Model checking of safety properties.. Formal Methods Syst Des 19(3): 291–314
32. Kupferman O, Vardi M (2001) On bounded specifications. In: Proceedings of the 9th international conference on logic for programming, artificial intelligence and reasoning. Lecture notes in computer science, vol 2250. Springer, Berlin, pp 24–38
33. Kupferman O, Vardi M, Wolper P (2000) Model checking of safety properties. J. ACM 47(2):312–360
34. Kurshan R (1994) Computer aided verification of coordinating processes. Princeton University Press, Princeton
35. Manna Z, Pnueli A (1987) Specification and verification of concurrent programs by ∀-automata. In: Proceedings of the 14th symposium on principles of programming. ACM, New York, pp 1–2
36. McMillan KL (1996) A conjunctively decomposed boolean representation for symbolic model checking. In: Proceedings of the 8th international conference on computer aided verification. New Brunswick, NJ, USA, pp 13–25
37. Narayan A, Isles A, Jain J, Brayton R, Sangiovanni-Vincentelli A (1997) Reachability analysis using partitioned-ROBDDs. In: Proceedings of the international conference on computer-aided design. IEEE Computer Society, Los Alamitos, pp 388–393
38. Narayan A, Jain J, Fujita M, Sangiovanni-Vincentelli A (1996) Partitioned ROBDDs-a compact, canonical and efficiently manipulable representation for Boolean functions. In: Proceedings of the international conference on computer-aided design. IEEE Computer Society, Los Alamitos, pp 547–554
39. Ravi K, Bloem R, Somenzi F (2000) A comparative study of symbolic algorithms for the computation of fair cycles. In: Proceedings of the 3rd international conference on formal methods in computer-aided design. Lecture notes in computer science, vol 1954. Springer, Berlin, pp 143–160
40. Sebastiani R, Singerman E, Tonetta S, Vardi MY (2004) GSTE is partitioned model checking. In: Proceedings of the 15th international conference on computer-aided verification. Lecture notes in computer science, vol 3114. Springer, Berlin, pp 229–241
41. Sebastiani R, Tonetta S (2003) "More Deterministic" vs. "smaller" Büchi automata for efficient ltl model checking. In: Proceedings of the conference on correct hardware design and verification methods. Lecture notes in computer science, vol 2860. Springer, Berlin, pp 126–140
42. Seger C-J, Bryant R (1995) Formal verification by symbolic evaluation of partially-ordered trajectories. Formal Methods Syst Des 6(2):147–189
43. Somenzi F, Bloem R (2000) Efficient Büchi automata from LTL formulae. In: Proceedings of the 12th international conference on computer-aided verification. Lecture notes in computer science, vol 1855. Springer, Berlin, pp 247–263
44. Vardi M, Wolper P (1986) An automata-theoretic approach to automatic program verification. In: Proceedings of the 1st symposium on logic in computer science. IEEE Computer Society, Los Alamitos, pp 332–344
45. Vardi M, Wolper P (1994) Reasoning about infinite computations. Inf Comput 115(1):1–37
46. Wang C, Bloem R, Hachtel G, Ravi K, Somenzi F (2001) Divide and compose: SCC refinement for language emptiness. In: Proceedings of 12th international conference on concurrency theory. Lecture notes in computer science, vol 2154. Springer, Berlin, pp 456–471

47. Wang C, Hachtel G (2002) Sharp disjunctive decomposition for language emptiness checking. In: Proceedings of the 4th international conference on formal methods in computer-aided design. Lecture notes in computer science, vol 2517. Springer, Berlin, pp 106–122
48. Yang J, Goel A (2002) GSTE through a case study. In: Proceedings of the international conference on computer-aided design. ACM, Los Alamitos, pp 534–541
49. Yang J, Seger C-J (2000) Generalized symbolic trajectory evaluation. Technical report, Intel SCL
50. Yang J, Seger C-J (2002) Generalized symbolic trajectory evaluation—abstraction in action. In: Proceedings of the 4th international conference on formal methods in computer-aided design. Lecture notes in computer science, vol 2517. Springer, Berlin, pp 70–87
51. Yang J, Seger C-JH (2003) Introduction to generalized symbolic trajectory evaluation. IEEE Trans Very Large Scale Integration Syst 11(3)