GSTE is Partitioned Model Checking

Roberto Sebastiani[1] ⋆, Eli Singerman[2], Stefano Tonetta[1], and Moshe Y. Vardi[3] ⋆⋆

[1] DIT, Università di Trento, {rseba,stonetta}@dit.unitn.it
[2] Intel Israel Design Center, eli.singerman@intel.com
[3] Dept. of Computer Science, Rice University, vardi@cs.rice.edu

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.

1 Introduction

Verifying whether an ω-regular property is satisfied by a finite-state system is a core problem in Model Checking (MC) [23, 31]. 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 [30, 23]. 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 [10]. 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) [3] 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) [13].

Another approach to formal verification is that of Symbolic Trajectory Evaluation (STE) [28], 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

⋆ Sponsored 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.
⋆⋆ 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.

operator [5]. Recently, GSTE [33] 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. Recent work on GSTE, e.g., in [34], 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 [21], 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 do not deal with abstraction, which is an orthogonal issue.) We first fill in details not given in [21] to show that assertion graphs are essentially *universal* ω-automata [24], 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. 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 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 a BDD representing a subset P of the product state space, GSTE operates on an array of BDDs, 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 [6] 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 results in smaller BDDs. The reduced BDD size 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 large transition relations.) Static partitioning techniques, which require an analysis of the state space, have been discussed, e.g., in [25]. Dynamic partitioning techniques, which are driven by heuristics to reduce BDD size, have been discussed, e.g., in [4]. Partitioning has been used in [19] to develop a distributed approach to SMC.

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 represent explicitly (e.g. in SPIN [20]) or symbolically (in NuSMV [6] or in Cadence SMV `www-cad.eecs.berkeley.edu/~kenmcmil/smv/`). 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. Other hybrid approaches have been described in [1, 7, 18], but ours is the first work to evaluate a hybrid approach in the context of general model checking.

The paper begins with an overview of the basic notions of SMC [9] and GSTE [33] in Section 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 Section 3, first, we prove that AGs and BAs are equivalent; then, we analyze the checking algorithms of GSTE and show that it is a partitioned version of standard SMC algorithms. In Section 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 Section 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 [9] and GSTE [33]. 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 $Y, Z, ..$ 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.

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 [33]). 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 systems states. (In practice, labels are given as predicates on system states; a predicate describes the sets 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 $\mathcal{F} = \{F_1, ..., F_n\}$, 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. We define $L_f(G)$ as the set of all finite initial paths of G and $L(G)$ as the set of all fair initial paths.

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 $E^-(v) = \{v' \in V \mid (v', v) \in E\}$. (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 trace 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 trace is accepted by a BA iff it satisfies the labeling function along at least one 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) traces of M accepted by B:

Definition 3.
- finite semantics: *if $\mathcal{F} = \{F\}$, $L_f(B) = \{\sigma \in L_f(M) \mid$ 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: *$L(B) = \{\sigma \in L(M) \mid$ there exists a fair path $\rho \in L(G)$ with $\sigma \models_{\mathcal{L}} \rho\}$.*

Since BAs have the complementary language of the specification, the model checking problem consists in verifying whether $L_f(B) = \emptyset$, in the case of finite semantics, $L(B) = \emptyset$, 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 trace iff, along all paths, either the trace does not satisfy the antecedent or if 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 function. Given a trace 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) traces of M accepted by A:

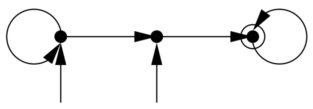

Fig. 1. FG example

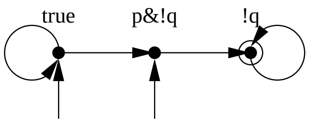

Fig. 2. BA example

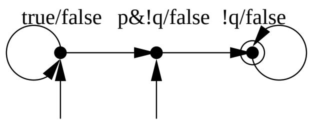

Fig. 3. AG example

Definition 4.

- finite semantics: *if* $\mathcal{F} = \{F\}$, $L_f(A) = \{\sigma \in L_f(M) \mid$ *for all finite path* $\rho \in L_f(G)$, *if* $|\sigma| = |\rho| = l$ *and* $\rho[l] \in F$, *then* $\sigma \models_A \rho\}$*;*
- fair semantics: $L(A) = \{\sigma \in L(M) \mid$ *for all fair path* $\rho \in L(G)$, $\sigma \models_A \rho\}$.

The model checking problem for AGs consists in 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. 1. The vertices are represented by points, the edges by arrows. An arrow without starting vertex point 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 Fig. 2 and 3. 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. $\square$

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 check 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 $V_P = \{(s,v) \mid s \in M, v \in V, s \in \mathcal{L}(v)\}$, $I_P = \{(s,v) \in V_P \mid s \in S_I, v \in V_I\}$, $E_P = \{((s,v),(s',v')) \mid (s,v) \in V_P, (s',v') \in V_P, (s,s') \in T, (v,v') \in E\}$, $\mathcal{F}_P = \{F_{P1}, ..., F_{Pn}\}$ 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 $L_f(P) = L_f(B)$. 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. 4.

Similarly, in the case of fair semantics, it is easy to see that $L(P) = L(B)$. 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. 5 [13]. SMC tools typically implement a variant of this doubly-nested fixpoint computation.

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

Fig. 4.

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

Fig. 5.

```
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 $v \in V$ $\alpha(v) :=$
8.      $\alpha'(v) \cup \bigcup_{v' \in E^-(v)} post(\alpha'(v')) \cap ant(v)$
9.  until $\alpha' = \alpha$
10. if fair semantics
11.   then return $\alpha \subseteq cons$
12.   else return $\alpha_{|F} \subseteq cons$
```

Fig. 6.

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

Fig. 7.

2.3 GSTE algorithms

The algorithm used by GSTE to check the AG in the different semantics is described in Fig. 6. The function $GSTE\_fairstates$ of line 2 is called only in the case of fair semantics and it is described in Fig. 7. $GSTE\_fairstates$ restricts the antecedent function to the states of the system that are extensible to fair paths. In the lines 3-9 of Fig. 6, 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 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 6-10), $GSTE\_fairstates$ 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 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.[4] First, the following two theorems show how to express AGs as BAs.

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 $L_f(B) = L_f(M) \backslash L_f(A)$*

Theorem 2. *Let $A = \langle G, ant, cons\rangle$ be an AG where $G = \langle V, V_I, E, \mathcal{F}\rangle$ and $\mathcal{F} = \{F_1, ..., F_n\}$. 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$\}$, $\mathcal{F}' = \{F_1 \times \{2\}, ..., F_n \times \{2\}\}$, $\mathcal{L}((v,k)) = ant(v)$ if $k \in \{0,2\}$, and $\mathcal{L}((v,k)) = ant(v) \cap (S \backslash cons(v))$ if $k = 1$. Then $L(B) = L(M) \backslash L(A)$*

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

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

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

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 Section 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

[4] A longer version of the paper, containing the proofs of the theorems and a more extensive bibliography , can be downloaded at `www.science.unitn.it/~stonetta/partitioning.html`.

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. 6 computes the reachable states of S_P. In fact, we could rewrite lines 6-8 in terms of CTL formulas as $\alpha = \alpha \cup \mathbf{EY[}\alpha\mathbf{]}$. 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. 4 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 lines 4-5 of the algorithm in Fig. 7, $GSTE\_fairstates$ actually computes the preimage of $ant_{|F}$ (seen as a set of states in S_P). So, we can rewrite these lines as $\alpha = \mathbf{EX[}(ant_{|F})\mathbf{]}$. Furthermore, the lines 7-9 are the same as $\alpha = \alpha \cup (ant \cap \mathbf{EX[}(\alpha)\mathbf{]})$ so that one can see the loop of lines 6-10 as $\alpha = \mathbf{E[}(ant)\mathbf{U}(\alpha)\mathbf{]}$. 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. 5.

4 SMC vs. property-driven partitioned SMC

In Section 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 $fairstates$ (Fig. 4 and 5). We partitioned both of them, by using NuSMV as 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 is Fig. 8 and Fig. 9. 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 Fig. 8 and 9 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 given by $\mathbf{EX[}(\alpha)\mathbf{]}(v) = \bigcup_{v' \in E(v)} pre(\alpha(v')) \cap \mathcal{L}(v)$.

We investigated if property-driven partitioning is effective for symbolic model checking. In particular, we applied such technique to LTL model checking. In fact, it is well

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

Fig. 8.

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

Fig. 9.

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 provides also 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 $\mathbf{E}_{\mathcal{F}}\mathbf{G[}true\mathbf{]}$ checks if the language of the product is empty. Since NuSMV does not apply any particular technique when φ is a safety formula [22], 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 $\mathbf{E[}true\mathbf{U}F\mathbf{]}$. 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.

We run our tests on three examples of SMV models (for the SMV code, we refer the reader to `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 Tab. 1) and two properties that failed (again one safety and one liveness property, see Tab. 2). The first example is a dining-philosophers protocol [12]. 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 philoso-

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

Table 1. Satisfied properties.

	Safety	Liveness
Dining	$G((p \wedge r \wedge X(r) \wedge XX(r) \wedge XXX(r)) \rightarrow XXXX(\neg e))$	$(GFr_1) \rightarrow (GFe_1)$
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 2. Failed properties.

pher is thinking and both her chopsticks are free and she is scheduled for 4 four 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.

We run NuSMV on the Rice Terascale Cluster (RTC), a 1 TeraFLOP Linux cluster based on Intel Itanium 2 Processors. A timeout has been fixed to 172800 sec. (2 days). The results are shown in Fig. 10 and 11. The execution time has been plotted in log scale against the number N of processes in the model. Every example takes a column of plots. On the first row, we have the safety properties and on the second one the liveness properties. Comparing the partitioned version with the non-partitioned one in the case of satisfied properties (Fig. 10), we notice that, in the first two columns (dining philosophers and mutual exclusion), the former outperforms the latter. Moreover, in the

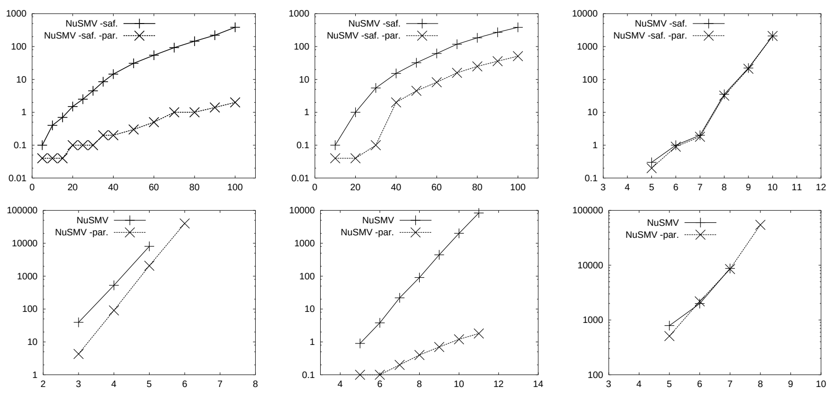

Fig. 10. Satisfied properties of Tab. 1. X axis: number of processes. Y axis: time. First row: performances of `''NuSMV -safety -partitioned''` and `''NuSMV -safety''` on safety properties. Second row: performances of `''NuSMV -partitioned''` and `''NuSMV''` on liveness properties. 1st column: dining-philosophers. 2nd column: mutex. 3rd column: life.

case of the safety property for dining philosophers and the liveness property for mutual exclusion, the gap is exponential, i.e. the difference between the two execution times grows exponentially with the size of the model. In the third column (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 outperforms always the non-partitioned one (see Fig. 11). Moreover, in the case of liveness properties, the improvement is exponential for all the three examples.

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

This opens us several directions for future work. First, 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. Second, it requires revisiting the issue of translating LTL formulas to BAs. Previous translations have focused on making the BA smaller (cf. [16, 11, 29, 15]) or more deterministic [27]. The relative merit of the two approaches has to be investigated in the context of property-partitioned SMC. Third, 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,

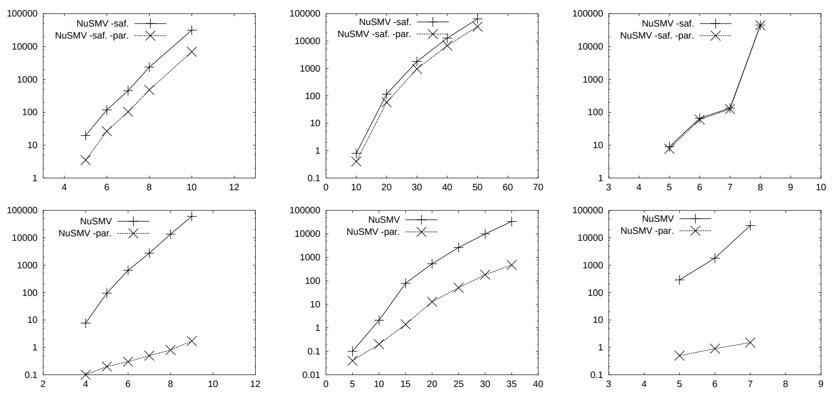

Fig. 11. Same pattern as in Fig. 10 but with the failed properties of Tab. 2.

cf. [2, 26, 14]. 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, [8, 17]. The combination of this technique with property-driven partitioning is also worth of investigation, which could benefit from the study of abstraction in GSTE [34, 32].

References

1. A. Biere, E. M. Clarke, and Y. Zhu. Multiple State and Single State Tableaux for Combining Local and Global Model Checking. In *Correct System Design*, pages 163–179, 1999.
2. R. Bloem, H.N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In *Formal Methods in Computer Aided Design*, 2000.
3. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Mhecking: 10^{20} States and Beyond. *Information and Computation*, 98(2), 1992.
4. G. Cabodi, P. Camurati, L. Lavagno, and S. Quer. Disjunctive Partitioning and Partial Iterative Squaring: An Effective Approach for Symbolic Traversal of Large Circuits. In *Design Automation Conf.*, 1997.
5. C.-T. Chou. The Mathematical Foundation of Symbolic Trajectory Evaluation. In *Computer-Aided Verification*. Springer, 1999.
6. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In *Proc. of the 11th Conf. on Computer-Aided Verification*, 1999.
7. A. Cimatti, M. Roveri, and P. Bertoli. Searching Powerset Automata by Combining Explicit-State and Symbolic Model Checking. In *TACAS'01*, 2001.
8. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, 2000.
9. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3), 1992.
11. N. Daniele, F. Guinchiglia, and M.Y. Vardi. Improved automata generation for linear temporal logic. In *Computer Aided Verification, Proc. 11th International Conf.*, 1999.

12. E.W. Dijksta. *Hierarchical ordering of sequential processes, Operating systems techniques*. Academic Press, 1972.
13. E.A. Emerson and C.L. Lei. Efficient Model Checking in Fragments of the Propositional μ–Calculus. In *Symp. on Logic in Computer Science*, 1986.
14. K. Fisler, R. Fraer, G. Kamhi, M.Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Tools and algorithms for the construction and analysis of systems*, 2001.
15. C. Fritz. Constructing Büchi Automata from Linear Temporal Logic Using Simulation Relations for Alternating Bchi Automata. In *Implementation and Application of Automata*, 2003.
16. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification*, 1995.
17. S.G. Govindaraju and D.L. Dill. Counterexample-guided Choice of Projections in Approximate Symbolic Model Checking. In *Proc. of ICCAD 2000*, 2000.
18. T. A. Henzinger, O. Kupferman, and S. Qadeer. From Pre-Historic to Post-Modern Symbolic Model Checking. *Form. Methods Syst. Des.*, 23(3), 2003.
19. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. A Scalable Parallel Algorithm for Reachability Analysis of Very Large Circuits. In *Formal Methods in System Design*, 2002.
20. G.J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2003.
21. A.J. Hu, J. Casas, and J. Yang. Reasoning about GSTE Assertion Graphs. In *Correct Hardware Design and Verification Methods*. Springer, 2003.
22. O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal methods in System Design*, 19(3), 2001.
23. R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.
24. Z. Manna and A. Pnueli. Specification and Verification of Concurrent Programs by $\forall$-automata. In *Proc. 14th ACM Symp. on Principles of Programming*, 1987.
25. A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned ROBDDs-a Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions. In *Inter. Conf. on Computer-aided design*, 1996.
26. K. Ravi, R. Bloem, and F. Somenzi. A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles. In *Formal Methods in Computer-Aided Design*, 2000.
27. R. Sebastiani and S. Tonetta. "More Deterministic" vs. "Smaller" Buechi Automata for Efficient LTL Model Checking. In *Correct Hardware Design and Verification Methods*, 2003.
28. C.-J.H. Seger and R.E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in System Design: An Inter. Journal*, 6(2), 1995.
29. F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In *Proc CAV'00*, volume 1855 of *LNCS*. Springer, 2000.
30. M.Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proc. 1st Symp. on Logic in Computer Science*, 1986.
31. M.Y. Vardi and P. Wolper. Reasoning about Infinite Computations. *Information and Computation*, 115(1), 1994.
32. J. Yang and A. Goel. GSTE through a Case Study. In *Proc. of the 2002 IEEE/ACM Inter. Conf. on Computer-Aided Design*. ACM Press, 2002.
33. J. Yang and C.-J.H. Seger. Generalized Symbolic Trajectory Evaluation, 2000. Intel SCL Technical Report, under revision for Journal Publication.
34. J. Yang and C.-J.H. Seger. Generalized Symbolic Trajectory Evaluation - Abstraction in Action. In *Formal Methods in Computer-Aided Design*, 2002.