

# Symbolic CTL Model Checking of Asynchronous Systems Using Constrained Saturation<sup>\*</sup>

Yang Zhao and Gianfranco Ciardo

Department of Computer Science and Engineering  
University of California, Riverside  
{zhaoy,ciardo}@cs.ucr.edu

**Abstract.** The saturation state-space generation algorithm has demonstrated clear improvements over state-of-the-art symbolic methods for asynchronous systems. This work is motivated by efficiently applying saturation to CTL model checking. First, we introduce a new “constrained saturation” algorithm which constrains state exploration to a set of states satisfying given properties. This algorithm avoids the expensive after-the-fact intersection operations and retains the advantages of saturation, namely, exploiting event locality and benefiting from recursive local fix-point computations. Then, we employ constrained saturation to build the set of states satisfying EU and EG properties for asynchronous systems. The new algorithm can achieve orders-of-magnitude reduction in runtime and memory consumption with respect to methods based on breath-first search, and even with a previously-proposed hybrid approach that alternates between “safe” saturation and “unsafe” breadth-first searches. Furthermore, the new approach is fully general, as it does not require the next-state function to be expressible in Kronecker form. We conclude this paper with a discussion of some possible future work, such as building the set of states belonging to strongly connected components.

## 1 Introduction

CTL model checking is an important state-of-the-art approach in formal verification. Paired with the use of BDDs [2], which provide a time and space efficient data structure to perform operations such as union, intersection, and relational product over sets of states, symbolic model checking [13] is one of the most successful techniques to verify industrial hardware and embedded software systems.

Most current symbolic model checkers, such as NuSMV [12], employ methods based on breath-first search (BFS). The saturation algorithm [7] employs a very different philosophy, recursively computing “local fixpoints”. A series of publications has proven the clear advantages of saturation for state-space generation over traditional symbolic approaches [6, 8, 9, 14], while extending its applicability to increasingly general settings. However, our previous attempts to apply saturation to CTL model checking have been only partially successful [10].

---

<sup>\*</sup> Work supported in part by the National Science Foundation under grant CCF-0848463.

This paper addresses CTL model checking for asynchronous systems by proposing an extended *constrained saturation* algorithm. This algorithm constrains the saturation-based state-space exploration to a given set of states without explicitly executing the expensive intersection operations normally required to implement CTL operators. Furthermore, unlike the original approach [10] where the next-state function had to satisfy a Kronecker expression, the proposed algorithm is fully general, as it employs a disjunctive-then-conjunctive encoding that exploits the common characteristics of asynchronous systems. Constrained saturation can be used to compute the set of states satisfying an EU formula as well as to efficiently compute the *backward reachability relation*, which we in turn use for a new algorithm to compute the set of states satisfying an EG formula.

The remainder of this paper is organized as follows. Section 2 introduces the relevant background on MDDs and the saturation algorithm. Section 3 introduces constrained saturation and new EU computation algorithm. Section 4 proposes a new EG computation algorithm based on the backward reachability relation. We conclude this paper and outline future work in the last section.

## 2 Preliminaries

Consider a discrete-state model  $(\mathcal{S}, \mathcal{S}_{init}, \mathcal{E}, \mathcal{N})$  where the state space  $\mathcal{S}$  is given by the product  $\mathcal{S}_L \times \dots \times \mathcal{S}_1$  of local state spaces of  $L$  submodels, that is, each (global) state  $\mathbf{i}$  is a tuple  $(i_L, \dots, i_1)$ , where  $i_k \in \mathcal{S}_k$ , for  $L \geq k \geq 1$ ; the set of initial states is  $\mathcal{S}_{init} \subseteq \mathcal{S}$ ; the set of (asynchronous) events is  $\mathcal{E}$ ; the next-state function  $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$  is described in disjunctively partitioned form as  $\mathcal{N} = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha$ , where  $\mathcal{N}_\alpha(\mathbf{i})$  is the set of states that can be reached in one step when  $\alpha$  occurs, or fires, in state  $\mathbf{i}$ . We say that  $\alpha$  is *enabled* in state  $\mathbf{i}$  if  $\mathcal{N}_\alpha(\mathbf{i}) \neq \emptyset$ . Correspondingly,  $\mathcal{N}^{-1}$  and  $\mathcal{N}_\alpha^{-1}$  denote the inverse next-state functions, e.g.,  $\mathcal{N}_\alpha^{-1}(\mathbf{i})$  is the set of states that can reach  $\mathbf{i}$  in one step by firing event  $\alpha$ .

For high-level models, the generation of the state space  $\mathcal{S}$  is an important and interesting problem in itself. This is particularly true for models such as Petri nets, where the sets  $\mathcal{S}_k$  might not be known a priori. However, using the saturation algorithm, the state spaces of complex models can usually be generated in acceptable runtime, so we now assume that state-space generation is a preprocessing step that has been already performed prior to model checking. Consequently, the sets  $\mathcal{S}_k$ , their sizes  $n_k$ , and the reachable state space  $\mathcal{S}_{rch} \subseteq \mathcal{S}$  are assumed known in the following discussion, and we let  $\mathcal{S}_k = \{0, \dots, n_k - 1\}$ , without loss of generality. More details about our state-space generation algorithm are reviewed in Section 2.3.

### 2.1 Symbolic encoding of sets of states

Binary decision diagrams (BDDs) [2] are the most widely used data structure to store and manipulate sets of states. Instead of BDDs, we employ *multi-way decision diagrams* (MDDs) to encode sets of states. MDDs extend BDDs by allowing integer-valued variables, so that the choices for nodes at level  $k$  naturally correspond to the local states of submodel  $k$ .

**Definition 1.** A *quasi-reduced* MDD is an acyclic directed edge-labeled graph where:

- Each node  $a$  belongs to a level in  $\{L, \dots, 1, 0\}$ , denoted by  $a.lvl$ .
- There is a single root node.
- The only terminal nodes are  $\mathbf{0}$  and  $\mathbf{1}$ , and are at level 0.
- A nonterminal node  $a$  at level  $k$ , with  $L \geq k \geq 1$ , has  $n_k$  outgoing edges labeled with a different integer in  $\{0, \dots, n_k - 1\}$ . The node pointed by the edge labeled with  $i_k$  is denoted  $a[i_k]$ , and, if not  $\mathbf{0}$ , it must be at level  $k - 1$ .

The set encoded by MDD node  $a$  at level  $k > 1$  is then recursively defined as

$$\mathcal{B}(a) = \bigcup_{i_k \in \mathcal{S}_k} \{i_k\} \times \mathcal{B}(a[i_k]).$$

with terminal cases  $\mathcal{B}(\mathbf{0}) = \emptyset$  and  $\mathcal{B}(a) = \{i_1 \in \mathcal{S}_1 : a[i_1] = \mathbf{1}\}$  if  $a.lvl = 1$ .

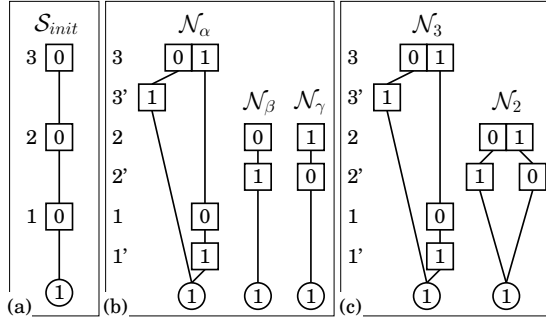
## 2.2 Symbolic encoding of the next-state functions

In a traditional symbolic approach, the next-state function  $\mathcal{N}$  is encoded using a  $2L$ -level BDD or MDD, with levels ordered as  $L, L', \dots, 1, 1', 0$ , where levels  $L$  through 1, corresponding to the “from” states, and levels  $L'$  through  $1'$ , corresponding to the “to” states, are *interleaved*. Given a next-state function  $\mathcal{N}$  and a set of states  $\mathcal{X}$ , the image computation or *relational product* builds the set  $\mathcal{N}(\mathcal{X}) = \{\mathbf{j} : \exists \mathbf{i} \in \mathcal{X}, (\mathbf{i}, \mathbf{j}) \in \mathcal{N}\}$ .

A commonly employed technique is to conjunctively or disjunctively partition the next-state function encoded by a monolithic MDD into several MDDs [3, 11]. For asynchronous systems, it is natural to disjunctively store each next-state function  $\mathcal{N}_\alpha$ , for each event  $\alpha \in \mathcal{E}$ , so that the overall next-state function  $\mathcal{N} = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha$  is actually never stored as a single MDD.

*Locality* is a fundamental property enjoyed by most asynchronous systems: an event  $\alpha$  is *independent* of the  $k^{\text{th}}$  submodel (or, briefly, of  $k$ ) if its enabling does not depend on  $i_k$  and its firing does not change the value of  $i_k$ . A level  $k$  belongs to the *support* set of event  $\alpha$ , denoted  $\text{supp}(\alpha)$ , if  $\alpha$  is not independent of  $k$ . We define  $\text{Top}(\alpha)$  and  $\text{Bot}(\alpha)$  to be the maximum and minimum levels in  $\text{supp}(\alpha)$ , and  $\mathcal{E}_k$  to be the set of events  $\{\alpha \in \mathcal{E} : \text{Top}(\alpha) = k\}$ . Also, we let  $\mathcal{N}_k$  be the next-state function corresponding to all events in  $\mathcal{E}_k$ , i.e.,  $\mathcal{N}_k = \bigcup_{\text{Top}(\alpha)=k} \mathcal{N}_\alpha$ .

In previous work [9], the locality of events is indicated by the presence of identity matrices in a *Kronecker* description of the model’s next-state function. However, while the existence of a Kronecker description is not a restriction for some formalisms (e.g., ordinary Petri nets, even if extended with inhibitor and reset arcs), it does impose constraints on the decomposition granularity in others (e.g., Petri nets with arbitrary marking-dependent arc cardinalities or transition guards). [11] proposes a new encoding for the transition relation based on a disjunctive-conjunctive partition and a *fully-identity* reduction rule for MDDs. [14] compares several possible MDD reduction rules and finally adopts the *quasi-identity-fully* (*QIF*) reduction rule for MDDs encoding next-state functions. This



**Fig. 1.** An example of set and next-state function encoding using MDDs.

encoding is successfully applied to saturation and allows for complex variable dependencies in asynchronous systems. A  $2L$ -level MDD encoding next-state function  $\mathcal{N}_\alpha$  with this QIF reduction rule satisfies the following:

- The root node is at level  $Top(\alpha)$ , an unprimed level.
- (Interleaving) Level  $k$  is immediately followed by level  $k'$ , for  $L \geq k \geq 1$ .
- (Quasi-reduced rule) If  $k \in supp(\alpha)$ , level  $k$  is present on every path in the MDD encoding  $\mathcal{N}_\alpha$ .
- (Identity-reduced rule) If  $k \in supp(\alpha)$ , level  $k'$  can be present on a path in the MDD encoding  $\mathcal{N}_\alpha$ , or it can be absent. In the latter case, the meaning of an edge  $a[i_k] = b$ , with  $a.lvl = k > b.lvl$  is that a *singular* node  $c$  at level  $k'$  has been skipped, i.e., a node with  $c[i_k] = b$  and  $c[j_k] = \mathbf{0}$  for  $j_k \in \mathcal{S}_k \setminus \{i_k\}$ .
- (Fully-reduced rule) If  $k \notin supp(\alpha)$ , level  $k$  is absent on every path in the MDD encoding  $\mathcal{N}_\alpha$ . The meaning of an edge  $a[i_l] = b$ , with  $a.lvl > k > b.lvl$  is that a *redundant* node  $c$  at level  $k$  has been skipped, i.e., a node with  $c[i_k] = b$  for every  $i_k \in \mathcal{S}_k$ .
- (Fully-identity-reduced pattern) This is really a combination of the two preceding cases. The interpretation of an edge skipping levels  $k$  and  $k'$  (or of the root being at a level below  $k'$ ) is that  $\alpha$  is independent of  $i_k$ , i.e., event  $\alpha$  is enabled when the  $k^{\text{th}}$  local state is  $i_k$ , for any  $i_k \in \mathcal{S}_k$ , and that, if  $\alpha$  fires, the  $k^{\text{th}}$  local state remains equal to  $i_k$  in the new state.

Figure 1(a) shows a 3-level MDD encoding an initial set of states containing a single state,  $\mathcal{S}_{init} = \{000\}$ . Figure 1(b) shows three MDDs encoding next-state functions for events  $\alpha$ ,  $\beta$ , and  $\gamma$ , respectively. Figure 1(c) shows the next-state functions merged by levels,  $\mathcal{N}_3 = \mathcal{N}_\alpha$ ,  $\mathcal{N}_2 = \mathcal{N}_\beta \cup \mathcal{N}_\gamma$ , and  $\mathcal{N}_1 = \emptyset$  (not shown). Note, for example, that  $\mathcal{N}_2$  does not depend on level 1 and that the node at level  $3'$  for the 1-child of the root of  $\mathcal{N}_3$  is skipped due to the identity-reduced rule.

### 2.3 State space generation

All symbolic approaches to state-space generation use some variant of symbolic image computation. The simplest approach is a breadth-first iteration, directly

implementing the definition of the state space  $\mathcal{S}_{rch}$  as the fixpoint of the expression  $\mathcal{S}_{init} \cup \mathcal{N}(\mathcal{S}_{init}) \cup \mathcal{N}^2(\mathcal{S}_{init}) \cup \mathcal{N}^3(\mathcal{S}_{init}) \cup \dots$ .

Locality and disjunctive partition of the next-state function form instead the basis is for the saturation algorithm. The key idea is to fire events node-wise, bottom-up, and exhaustively, instead of level-wise and exactly once per iteration. A node  $a$  is *saturated* if it is a fixpoint with respect to firing any event that is independent of all levels above  $k$ :

$$\forall h, k \geq h \geq 1, \forall \alpha \in \mathcal{E}_h, \mathcal{S}_L \times \dots \times \mathcal{S}_{k+1} \times \mathcal{B}(a) \supseteq \mathcal{N}_\alpha(\mathcal{S}_L \times \dots \times \mathcal{S}_{k+1} \times \mathcal{B}(a))$$

Starting from the MDD encoding the initial states, the nodes in the MDD are saturated in order, from the bottom level to the top level. Of course, one of the advantages of saturation is that, given the QIF reduction, the application of  $\mathcal{N}_\alpha$  needs to start at level  $Top(\alpha)$ , and not all the way up, at the root of entire MDD. Every node is saturated before being checked in the unique table, including nodes newly created when firing an event on a node at a higher level.

## 2.4 CTL model checking

CTL is a widely used temporal logic to describe the system properties because of its simple yet expressive syntax. All CTL properties are state properties and can be checked by manipulating sets of states. It is well-known that  $\{\text{EX}, \text{EU}, \text{EG}\}$  is a complete set of operators for CTL, that is, it can be used to express any other CTL operator (for example, the EF operator is a special case of EU, since  $\text{EF}p \equiv \text{EtrueUp}$ ). The  $\text{EX}p$  operator can be easily computed as the relational product  $\mathcal{N}^{-1}(\mathcal{P})$ , where  $\mathcal{P}$  is the set of states satisfying property  $p$ . Building the set of states satisfying  $\text{EF}p$  is instead essentially the same process as state-space generation, the only differences being that we start from  $\mathcal{P}$  instead of  $\mathcal{S}_{init}$  and that we go backwards instead of forwards, thus we use  $\mathcal{N}^{-1}$  (or  $\mathcal{N}_\alpha^{-1}$ , or  $\mathcal{N}_k^{-1}$ , as appropriate) instead of  $\mathcal{N}$ .

The traditional algorithm to obtain the set of states satisfying  $\text{EpU}q$  computes a least fixpoint (see *EUtrad* in Figure 2; all sets of states and relations over states in our pseudocode are encoded using MDDs, of course). Starting from  $\mathcal{Q}$ , the set of states satisfying  $q$ , it computes the intersection of the preimage of the explored states,  $\mathcal{X}$ , with the states in  $\mathcal{P}$ . The newly computed states are added to the explored states, for the next iteration. The number of iterations is thus equal to the longest path of states in  $\mathcal{P} \setminus \mathcal{Q}$  reaching a state in  $\mathcal{Q}$ .

A saturation-based EU computation algorithm was instead proposed in [10] (see *EUsat* in Figure 2). First, it partitions the set  $\mathcal{E}$  of events into safe,  $\mathcal{E}_S$ , and unsafe,  $\mathcal{E}_U = \mathcal{E} \setminus \mathcal{E}_S$ , where  $\alpha \in \mathcal{E}$  is safe iff  $\mathcal{N}_\alpha^{-1}(\mathcal{P} \cup \mathcal{Q}) \subseteq \mathcal{P}$ , i.e., it is such that, following its firing backwards, we can only find states in  $\mathcal{P}$  (alternatively, we can restrict all sets by intersecting them with the reachable states  $\mathcal{S}_{rch}$  in the above test). The algorithm iteratively (1) saturates the MDD encoding the set of explored states using only safe events, then (2) fires each unsafe event once using  $\mathcal{N}_U^{-1} = \bigcup_{\alpha \in \mathcal{E}_U} \mathcal{N}_\alpha^{-1}$  in breadth-first fashion, then (3) intersects the result with  $\mathcal{P}$ , and finally (4) adds the result to the working set  $\mathcal{X}$ .

$EUtrad(in \mathcal{P}, \mathcal{Q})$ : set of state 1 declare $\mathcal{X}$ : set of states 2 $\mathcal{X} \leftarrow \mathcal{Q}$ ; 3 repeat 4 $\mathcal{X} \leftarrow \mathcal{X} \cup (\mathcal{N}^{-1}(\mathcal{X}) \cap \mathcal{P})$ ; 5 until $\mathcal{X}$ does not change; 6 return $\mathcal{X}$ ;	$EUsat(in \mathcal{P}, \mathcal{Q})$ : set of state 1 declare $\mathcal{X}, \mathcal{Y}$ : set of state; 2 declare $\mathcal{E}_U, \mathcal{E}_S$ : set of event; 3 $ClassifyEvents(\mathcal{P} \cup \mathcal{Q}, \mathcal{E}_U, \mathcal{E}_S)$ 4 $\mathcal{X} \leftarrow \mathcal{Q}$ ; 5 $Saturate(\mathcal{X}, \mathcal{E}_S)$ 6 repeat 7 $\mathcal{Y} \leftarrow \mathcal{X}$ ; 8 $\mathcal{X} \leftarrow \mathcal{X} \cup (\mathcal{N}_U^{-1}(\mathcal{X}) \cap (\mathcal{P} \cup \mathcal{Q}))$ 9   if $\mathcal{X} \neq \mathcal{Y}$ then 10 $Saturate(\mathcal{X}, \mathcal{E}_S)$ 11 until $\mathcal{X} = \mathcal{Y}$ ; 12 return $\mathcal{X}$ ;
$EGtrad(in \mathcal{P})$ : set of state 1 declare $\mathcal{X}$ : set of states 2 $\mathcal{X} \leftarrow \mathcal{P}$ ; 3 repeat 4 $\mathcal{X} \leftarrow \mathcal{X} \cap \mathcal{N}^{-1}(\mathcal{X})$ ; 5 until $\mathcal{X}$ does not change; 6 return $\mathcal{X}$ ;	

**Fig. 2.** Traditional and saturation-based CTL model checking algorithms.

The traditional EG algorithm (see  $EGtrad$  in Figure 2) computes a greatest fixpoint by iteratively eliminating states without successors in the working set  $\mathcal{X}$ . [10] also attempts to compute set of states satisfying  $EGp$  using forward and backward EU saturation from a single state in  $\mathcal{P}$ . However, this approach is more efficient than the traditional algorithm only in very special cases.

### 3 Constrained saturation for the EU operator

The set of states satisfying  $EpUq$  is a least fixpoint, where the saturation algorithm could be efficiently employed. However, the challenge arises from the need to “filter out” the states not in  $\mathcal{P}$  before exploring their predecessors. Failure to do so can result in paths which temporarily go out of  $\mathcal{P}$ , so that the result may include some states not satisfying  $EpUq$ . The saturation algorithm does not find states in breadth-first-search order, as the process of saturating a node often consists of firing a series of events. Performing an expensive intersection operation *after each firing* would be enormously less time and memory efficient.

The advantage of Algorithm  $EUsat$  [10] over  $EUtrad$  depends on the structure of the model. If there are no safe events with respect to a given property  $p$ ,  $EUsat$  degrades to the simple breadth-first exploration of  $EUtrad$ . To overcome this difficulty, we propose two approaches, both aimed at exploring only states in  $\mathcal{P}$  without requiring an expensive intersection operation after each firing.

**1. Saturation with constrained next-state functions.** For each  $\mathcal{N}_k^{-1}$ , we build a *constrained inverse next-state function*  $\mathcal{N}_{k, \mathcal{P}}^{-1}$  such that

$$\mathbf{j} \in \mathcal{N}_{k, \mathcal{P}}^{-1}(\mathbf{i}) \iff (\mathbf{j} \in \mathcal{P}) \wedge \mathbf{j} \in \mathcal{N}_k^{-1}(\mathbf{i}).$$

Algorithm  $ConsNSF$  in Figure 3 builds the MDD representation of  $\mathcal{N}_{\alpha, \mathcal{P}}^{-1}$ .

**2. Constrained saturation.** This is the main contribution of our paper. We do not explicitly constrain the next-state functions, but perform instead a “check-and-fire” step when computing the constrained preimage (function *ConsRelProd* in the pseudocode of Figure 3), based on the following observation:

$$\mathcal{B}(t) = \text{RelProd}(s, r) \cap \mathcal{B}(a) \iff \mathcal{B}(t[i']) = \bigcup_{i \in \mathcal{S}_l} \text{RelProd}(s[i], r[i][i']) \cap \mathcal{B}(a[i']), \quad (1)$$

where  $t$  and  $s$  are  $L$ -level MDDs encoding sets of states,  $l = s.lvl$ , and  $r$  is a  $2L$ -level encoding a next-state function. This can be considered as a form of *ITE* operator [2], widely used in BDD operations, but extended from boolean to integer variables. The overall process of EU computation based on constrained saturation is then shown in Figure 3. The key differences from the saturation algorithm in [9] are marked with a “✱”.

The idea of the first approach is straightforward: all constrained next-state functions  $\mathcal{N}_{\alpha, \mathcal{P}}^{-1}$  are forced to be safe by definition. According to the saturation-based EU computation algorithm in Figure 2, the result is obtained in a single iteration (a single call to *Saturate*). The downside of this approach is a possible decrease in locality. A property  $p$  is *dependent* on level  $k$  if the value of  $i_k$  affects the satisfiability of  $p$ , i.e., if the (fully-reduced) encoding of  $p$  has nodes at level  $k$ . After constraining a next-state function  $\mathcal{N}_\alpha$  with  $p$ , the levels on which  $p$  depends become part of the support, thus,  $\text{Top}(\mathcal{N}_{\alpha, \mathcal{P}}^{-1}) = \max\{\text{Top}(\mathcal{P}), \text{Top}(\mathcal{N}_\alpha^{-1})\}$ . If  $\mathcal{P}$  depends on level  $L$ , all the constrained next-state functions belong to  $\mathcal{E}_L^{-1}$  and the saturation algorithm degrades to BFS, losing its advantages.

The second approach, constrained saturation, does not modify the transition relation explicitly, but constrains the state exploration “on-the-fly” following the “check-and-fire” policy. This policy guarantees that the state exploration is constrained to set  $\mathcal{P}$ . At the same time, it retains the advantages of saturation due to exploiting event locality and employing recursive local fixpoint computations. In the pseudocode shown in the right portion of Figure 3, the “check-and-fire” policy can be summarized into two cases:

1. If  $p[i] = \mathbf{0}$ ,  $s[i]$  is kept unchanged without adding new states (line 4 in function *ConsSaturate*).
2. When computing the relational product, check whether the newly generated local state is included in  $p$  on each level (line 7 in function *ConsSaturate* and line 5 in function *ConsRelProd*). If instead  $p[i'] = \mathbf{0}$  in formula (1), the relational product stops the recursive execution and returns  $\mathbf{0}$ .

Another tradeoff affecting efficiency is how to select the set of states  $\mathcal{P}$  when checking *EpUq*. In high-level models,  $\mathcal{P}$  is often associated with an atomic property, e.g., “place  $a$  of the Petri Net is empty” or a “localized” property dependent on just a few levels. There are then two reasonable choices to define  $\mathcal{P}$ :

- $\mathcal{P} = \mathcal{P}_{pot}$ : include all states in the potential state space  $\mathcal{S}$  that satisfy the given property, even if they are not reachable (recall that the potential state space is finite because the bound for each local state space  $\mathcal{S}_k$  is known).

<pre> mdd EUsatConsNSF(mdd P, mdd Q) 1 foreach <math>\alpha \in \mathcal{E}</math> do <math>\mathcal{N}_{\alpha, P}^{-1} \leftarrow \text{ConsNSF}(P, \mathcal{N}_{\alpha}^{-1})</math>; 2 mdd <math>s \leftarrow \text{Saturate}(Q)</math>; 3 <math>s \leftarrow \text{intersection}(s, \mathcal{S}_{rch})</math>; 4 return <math>s</math>; </pre>
<pre> mdd ConsNSF(mdd a, mdd r) 1 if <math>a = \mathbf{1}</math> and <math>r = \mathbf{1}</math> then return <math>\mathbf{1}</math>; 2 if <math>\text{InCache}_{\text{ConsNSF}}(a, r, t)</math> then return <math>t</math>; 3 mdd <math>t \leftarrow \mathbf{0}</math>; level <math>lr \leftarrow r.lvl</math>; level <math>la \leftarrow a.lvl</math>; 4 if <math>lr &lt; la</math> then 5   foreach <math>i \in \mathcal{S}_{la}</math> s.t. <math>a[i] \neq \mathbf{0}</math> do <math>t[i][i] \leftarrow \text{ConsNSF}(a[i], r)</math>; 6 else if <math>lr &gt; la</math> then 7   foreach <math>i, i' \in \mathcal{S}_{lr}</math> s.t. <math>r[i][i'] \neq \mathbf{0}</math> do <math>t[i][i'] \leftarrow \text{ConsNSF}(a, r[i][i'])</math>; 8 else 9   foreach <math>i, i' \in \mathcal{S}_{lr}</math> s.t. <math>r[i][i'] \neq \mathbf{0}</math> and <math>a[i'] \neq \mathbf{0}</math> do <math>t[i][i'] \leftarrow \text{ConsNSF}(a[i'], r[i][i'])</math>; 10 CacheAdd<math>_{\text{ConsNSF}}(a, r, t)</math>; 11 return <math>t</math>; </pre>
<pre> mdd EUconsSat(mdd a, mdd b) • <math>a</math>: the constraint; <math>b</math>: the set being saturated 1 mdd <math>s \leftarrow \text{ConsSaturate}(a, b)</math>; 2 <math>s \leftarrow \text{intersection}(s, \mathcal{S}_{rch})</math>; 3 return <math>s</math>; </pre>
<pre> mdd ConsSaturate(mdd a, mdd s) • <math>a</math>: the constraint; <math>s</math>: the set being saturated 1 if <math>\text{InCache}_{\text{ConsSaturate}}(a, s, t)</math> then return <math>t</math>; 2 level <math>l \leftarrow s.lvl</math>; mdd <math>t \leftarrow \text{NewNode}(l)</math>; mdd <math>r \leftarrow \mathcal{N}_l^{-1}</math>; 3 foreach <math>i \in \mathcal{S}_l</math> s.t. <math>s[i] \neq \mathbf{0}</math> do 4* if <math>a[i'] \neq \mathbf{0}</math> then <math>t[i] \leftarrow \text{ConsSaturate}(a[i], s[i])</math>; else <math>t[i] \leftarrow s[i]</math>; 5 repeat 6   foreach <math>i, i' \in \mathcal{S}_l</math> s.t. <math>r[i][i'] \neq \mathbf{0}</math> do 7* if <math>a[i'] \neq \mathbf{0}</math> then 8   mdd <math>u \leftarrow \text{ConsRelProd}(a[i'], t[i], r[i][i'])</math>; <math>t[i'] \leftarrow \text{Or}(t[i'], u)</math>; 9 until <math>t</math> does not change; 10 <math>t \leftarrow \text{UniqueTablePut}(t)</math>; CacheAdd<math>_{\text{ConsSaturate}}(a, s, t)</math>; 11 return <math>t</math>; </pre>
<pre> mdd ConsRelProd(mdd a, mdd s, mdd r) 1 if <math>s = \mathbf{1}</math> and <math>r = \mathbf{1}</math> then return <math>\mathbf{1}</math>; 2 if <math>\text{InCache}_{\text{ConsRelProd}}(a, s, r, t)</math> then return <math>t</math>; 3 level <math>l \leftarrow s.lvl</math>; mdd <math>t \leftarrow \mathbf{0}</math>; 4 foreach <math>i, i' \in \mathcal{S}_l</math> s.t. <math>r[i][i'] \neq \mathbf{0}</math> do 5* if <math>a[i'] \neq \mathbf{0}</math> then 6* mdd <math>u \leftarrow \text{ConsRelProd}(a[i'], s[i], r[i][i'])</math>; 7* if <math>u \neq \mathbf{0}</math> then 8* if <math>t = \mathbf{0}</math> then <math>t \leftarrow \text{NewNode}(l)</math>; 9* <math>t[i'] \leftarrow \text{Or}(t[i'], u)</math>; 10 <math>t \leftarrow \text{ConsSaturate}(a, \text{UniqueTablePut}(t))</math>; CacheAdd<math>_{\text{ConsRelProd}}(a, s, r, t)</math>; 11 return <math>t</math>; </pre>

**Fig. 3.** EU computation: saturation using a constrained next-state function ( $EU\text{satConsNSF}$ ) and constrained saturation ( $EU\text{consSat}$ ).

- $\mathcal{P} = \mathcal{P}_{rch}$ : include in  $\mathcal{P}$  only the *reachable* states that satisfy the given property,  $\mathcal{P}_{rch} = \mathcal{P}_{pot} \cap \mathcal{S}_{rch}$ .

We are normally only interested in reachable states and, of course, backward state exploration from unreachable states can only lead to more unreachable states; all these unreachable states can be filtered out *after* saturation, without affecting the correctness of the result (unlike the discussion at the beginning of this section, pertaining to filtering out states not in  $\mathcal{P}$ ). Exploration including the unreachable states might result in greater time and memory requirements, in which case using  $\mathcal{P}_{rch}$  is preferable for algorithmic efficiency. On the other hand,  $\mathcal{P}_{pot}$  is often dependent on very few levels, while, for most models,  $\mathcal{P}_{rch}$  is a strict subset of  $\mathcal{S}$ , thus depends on many levels, and this increases the complexity of algorithm, especially for the first approach. In the ideal case, we can constrain the state exploration to  $\mathcal{P}_{rch}$  with an acceptable overhead.

The experimental results in Section 5 demonstrate that constrained saturation using  $\mathcal{P}_{rch}$  tends to perform much better than saturation with constrained next-state functions in both runtime and memory consumption. We select it as our main method to compute the EU operator, as well as the reachability relation, which we introduce in the next section.

## 4 Reachability relation and the EG operator

The EGp property describes the existence of a path in  $\mathcal{P}$  from a state leading to a nontrivial strongly-connected component (SCC), where  $p$  holds in all states along the path and in the SCC. In this section, we propose a new EGp computation algorithm based on the reachability relation, built using constrained saturation.

The following defines the (backward) *reachability relation* of a set of states  $\mathcal{X}$  within  $\mathcal{P}$ , denoted with  $(\mathcal{N}_{\mathcal{X}, \mathcal{P}}^{-1})^+$ .

**Definition 2.** *Given a state  $\mathbf{i} \in \mathcal{X}$ ,  $\mathbf{j} \in (\mathcal{N}_{\mathcal{X}, \mathcal{P}}^{-1})^+(\mathbf{i})$  iff there exists a nontrivial (i.e., positive length) forward path  $\pi$  from  $\mathbf{j}$  to  $\mathbf{i}$  and all states in  $\pi$  belong to  $\mathcal{P}$ .*

If  $\mathbf{j} \in (\mathcal{N}_{\mathcal{X}, \mathcal{P}}^{-1})^+(\mathbf{i})$ , we know that  $\mathbf{j}$  is in  $\mathcal{P}$ . Since it is not always necessary to compute the reachability relation for all  $\mathbf{i} \in \mathcal{S}$ , we can build the reachability relation starting only from states in  $\mathcal{X}$ , to reduce time and memory consumption.

**Claim 1:** If  $\mathbf{j} \in (\mathcal{N}_{\mathcal{S}, \mathcal{P}}^{-1})^+(\mathbf{i})$ , then  $\exists \mathbf{i}' \in \mathcal{N}^{-1}(\mathbf{i}) \cap \mathcal{P}$  s.t.  $\mathbf{j} \in \text{ConsSaturate}(\mathcal{P}, \{\mathbf{i}'\})$ .

This claim comes from the definition of constrained saturation and derives a way of building the reachability relation efficiently. Starting from the MDD encoding  $\mathcal{N}^{-1}$ , appropriately restricted to a set of states (e.g.,  $\mathcal{P}$ ), we compute the constrained saturation for states encoded at the primed levels. Analogous to constrained saturation, this process can be performed bottom-up recursively on each level.

**Claim 2:** EGp holds in state  $\mathbf{j}$  iff  $\exists \mathbf{i} \in \mathcal{P}$  s.t.  $\mathbf{i} \in (\mathcal{N}_{\mathcal{P}, \mathcal{P}}^{-1})^+(\mathbf{i})$  and  $\mathbf{j} \in (\mathcal{N}_{\mathcal{P}, \mathcal{P}}^{-1})^+(\mathbf{i})$ .

From this claim, we can obtain an algorithm to compute the set of states satisfying EGp. Given a  $2L$ -level MDD encoding the reachability relation, it is

easy to obtain the set of states  $\mathcal{S}_{scc} = \{\mathbf{i} : \mathbf{i} \in (\mathcal{N}_{\mathcal{P}, \mathcal{P}}^{-1})^+(\mathbf{i})\}$ . These states belong to SCCs where property  $\mathcal{P}$  holds continuously. Then, the result of EGp can be obtained computing  $RelProd(\mathcal{S}_{scc}, (\mathcal{N}_{\mathcal{P}, \mathcal{P}}^{-1})^+)$ .

Building the reachability relation is a time and memory intensive task, constituting the bottleneck for our new EG algorithm. On the other hand, the reachability relation contains more information than the basic EG property and has further applications. We discuss one of them: EG computation under a weak fairness constraint. Fairness is widely used in formal specification of protocols; in particular, weak fairness specifies that there is an infinite execution on which some states, say in  $\mathcal{F}$ , appear infinitely often. The difficulty lies in that the fact that executions in SCCs which do not contain states in  $\mathcal{F}$  must be eliminated to guarantee the fairness, and the traditional EG algorithm cannot handle this problem. However, this extension is easy in our framework, as discussed next.

**Claim 3:** EGp under weak a fairness constraint  $\mathcal{F}$  holds in state  $\mathbf{j}$  iff  $\exists \mathbf{i} \in \mathcal{F}$  s.t.  $\mathbf{i} \in (\mathcal{N}_{\mathcal{F} \cap \mathcal{P}, \mathcal{P}}^{-1})^+(\mathbf{i})$  and  $\mathbf{j} \in (\mathcal{N}_{\mathcal{F} \cap \mathcal{P}, \mathcal{P}}^{-1})^+(\mathbf{i})$ .

Since  $\mathbf{i} \in \mathcal{F}$ , the SCCs containing such a state satisfy the fairness constraint. We only need to build reachability relation on these states. An interesting point is that many fewer state pairs are in  $(\mathcal{N}_{\mathcal{F} \cap \mathcal{P}, \mathcal{P}}^{-1})^+$  than in  $(\mathcal{N}_{\mathcal{P}, \mathcal{P}}^{-1})^+$ . Although, in symbolic approaches, fewer states do not always lead to smaller MDDs, thus lower time and memory requirements, it is often the case in our framework that considering fairness will reduce the runtime, which is quite the opposite than in traditional approaches.

## 5 Experimental results

We implemented the proposed approach in SMART [5] and report on experiments run on an Intel Xeon 3.0Ghz workstation with 3GB RAM under SuSE Linux 9.1. Detailed descriptions of the models we use in the experiments can be found in the SMART User Manual [4]. The state space size for each model is controlled by a parameter  $N$ . For comparison, we study each model in both SMART and NuSMV version 2.4.3 [1].

### 5.1 Results for the EU computation

Table 1 shows the results for each EU query. The runtime (seconds), final (mem-f) and peak memory (mem-p) consumption (Kbytes) required by NuSMV, the old version of SMART [10], and our new approach are shown in the corresponding columns, for each model. We compare the following five approaches:

- *BFS*: the traditional EU algorithm implemented in SMART
- *ConNSFSat- $\mathcal{P}_{pot}$* : Saturation using constrained next-state functions, where the next-state functions are constrained using  $\mathcal{P}_{pot}$
- *ConNSFSat- $\mathcal{P}_{rch}$* : Saturation using constrained next-state functions, where the next-state functions are constrained using  $\mathcal{P}_{rch}$ .

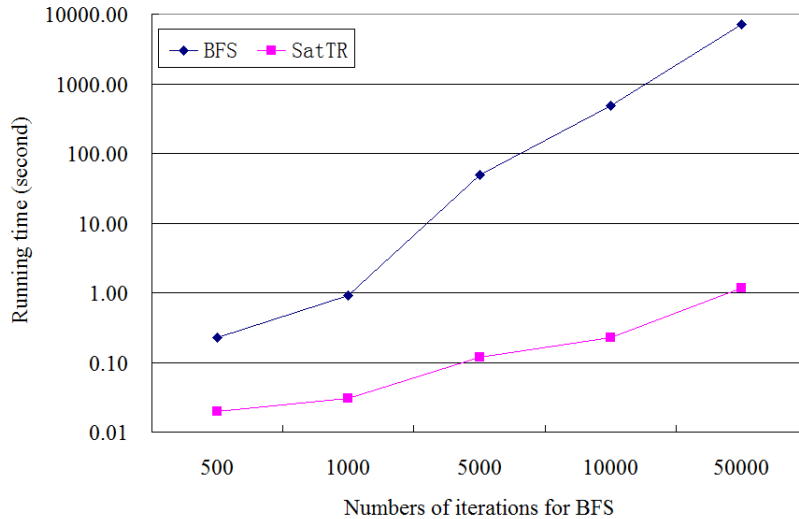


Fig. 4. EG computation based on BFS v.s. reachability relation.

- $ConSat-\mathcal{P}_{pot}$ : Constrained saturation with  $\mathcal{P}_{pot}$ .
- $ConSat-\mathcal{P}_{rch}$ : Constrained saturation with  $\mathcal{P}_{rch}$ .

Our main method, constrained saturation using  $\mathcal{P}_{rch}$ , outperforms (sometimes by orders-of-magnitude) NuSMV and other methods in both time and memory. In comparison with NuSMV, the saturation-based methods excel because of the local fixpoint iteration scheme. The improvement of our new work over our old approach [10] can be attributed to both the MDD encoding of the next-state function and the more advanced saturation schemes.

Overall,  $ConSat$  requires less runtime as well as memory than  $ConNSFSat$ , because the constrained next-state functions often impose overhead on relational product operations. The difference between the results of  $ConNSFSat-\mathcal{P}_{pot}$  and  $ConNSFSat-\mathcal{P}_{rch}$  shows the tradeoff discussed at the end of Section 3.  $ConSat$  constrained with  $\mathcal{P}_{rch}$  is more advantageous than with  $\mathcal{P}_{pot}$  because  $ConSat$  is not sensitive to the complexity of the constraint set due to our lightweight “check-and-fire” policy. The reduction in state exploration becomes the dominant factor for efficiency.

## 5.2 Results for the EG computation

Table 2 compares the results of NuSMV, BFS (SMART-BFS) and the method in Section 4 (SMART-RchRel) for EG computation with or without fairness constraints. For BFS, the number of iterations is listed in column *itr*.

Without fairness, traditional BFS (in NuSMV or SMART-BFS) is often orders-of-magnitude faster than our algorithm based on the reachability relation. This

Model	EU query																			
	NuSMV [1]			OldSmart [10]			SMART													
	BFS			$ConsNSFSat-P_{pot}$			$ConsNSFSat-P_{rch}$			$ConsSat-P_{pot}$			$ConsSat-P_{rch}$							
sec	KB(f)	KB(p)	sec	KB(f)	KB(p)	sec	KB(f)	KB(p)	sec	KB(f)	KB(p)	sec	KB(f)	KB(p)	sec	KB(f)	KB(p)			
leader	$E(pref_1 = 0)U(status_0 = leader)$																			
5	44.1	62,057	9.8	163	6,998	1.6	3,112	6,539	28.8	632	809	1.1	1,351	2,375	17.6	544	776	1.2	602	728
6	304.8	180,988	296.1	463	40,429	8.3	12,175	24,080	2,022.5	1,283	2,328	19.8	3,153	8,046	1,032.9	1,473	2,284	14.9	1,054	1,337
7	1,791.5	666,779	-	-	-	39.8	44,551	91,466	-	-	-	347.3	8,726	23,332	-	-	-	114.9	2,078	2,616
8	-	-	-	-	-	371.5	171,867	277,649	-	-	-	-	-	-	-	-	-	4,058.2	4,110	5,139
phil.	$E(phil_1 \neq eat)U(phil_0 = eat)$																			
50	1,644.2	75,282	< 0.1	73	287	4.7	3,248	4,586	< 0.1	464	465	0.2	1,117	1,591	0.2	436	436	< 0.1	435	435
100	-	-	0.2	147	872	60.6	7,141	16,446	0.2	861	864	0.8	3,119	4,907	0.4	817	843	0.3	816	841
500	-	-	2.9	740	16,082	-	-	-	0.3	3,837	3,870	160.7	52,447	87,791	0.7	3,856	3,885	0.4	3,850	3,876
1,000	-	-	4.1	1,036	30,707	-	-	-	0.6	5,262	5,269	1,228.2	100,516	164,137	4.8	7,589	7,697	0.8	5,253	5,271
robin	$E(p_1 \neq load)U(p_0 = send)$																			
10	29.2	70,583	5.1	186	4,447	0.1	494	574	< 0.1	92	92	< 0.1	204	204	< 0.1	90	90	< 0.1	64	64
20	-	-	-	-	-	1.4	3,088	3,738	< 0.1	283	317	< 0.1	598	659	0.1	312	312	< 0.1	166	166
100	-	-	-	-	-	-	-	-	5.7	14,272	14,274	636.1	17,398	30,833	3.6	15,899	15,907	< 0.1	4,269	4,298
200	-	-	-	-	-	-	-	-	163.1	105,787	105,788	-	-	-	111.6	119,988	119,988	0.4	28,781	28,790
fms	$E(M_1 > 0)U(P_1s = P_2s = P_3s = N)$																			
10	578.9	181,353	0.1	27	628	5.2	35,979	35,980	0.4	407	477	43.2	993	2,965	< 0.1	257	288	< 0.1	256	286
25	-	-	1.6	155	8,223	-	-	-	31.5	638	1,362	-	-	-	1.8	981	1,107	1.9	977	1,104
50	-	-	24.0	812	75,884	-	-	-	2,566.3	1,449	6,972	-	-	-	39.1	1,247	6,018	40.5	1,246	6,006
100	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1,128.0	4,302	40,588	1,200.9	4,299	40,504
queens	$E(p[N-1][N] = 1)U(p[N-1][N] = 0)$																			
10	15.3	77,861	-	-	-	1.1	9,744	9,744	< 0.1	2,728	2,728	< 0.1	1,899	1,899	0.2	3,532	3,532	< 0.1	1,841	1,841
11	84.0	327,907	-	-	-	7.2	41,252	41,252	0.5	12,445	12,445	0.1	7,289	7,312	1.6	15,877	15,877	< 0.1	7,043	7,062
12	595.1	1,355,128	-	-	-	237.2	186,360	186,360	2.3	52,764	52,765	0.7	80,785	80,859	9.1	75,347	75,347	< 0.1	29,714	29,763
13	-	-	-	-	-	-	-	-	10.9	236,501	236,506	6.1	166,711	166,837	86.9	337,265	337,265	< 0.1	133,121	133,121

Table 1. Results for the EU computation.

Model	EG query									Fairness				
	NuSMV			SMART-BFS			SMART-RchRel			NuSMV		SMART-RchRel		
	sec	KB(f)	itr	sec	KB(f)	KB(p)	sec	KB(f)	KB(p)	sec	KB	sec	KB(f)	KB(p)
leader	EG( $status_0 \neq leader$ )									$pref_0 = 1$				
3	0.2	9,308	14	0.1	139	196	4.9	934	1,115	0.6	11,428	0.02	266	268
4	3.0	50,187	18	< 0.1	400	436	791.6	5,999	7,225	11.2	49,655	207.4	5,271	6,394
phil.	EG( $phil_0 \neq eat$ )									$phil_0 = has\_left\_fork$				
10	0.1	7,193	4	< 0.1	95	95	0.1	170	170	0.2	8,447	< 0.1	113	113
50	3.0	50,187	4	< 0.1	220	241	0.2	682	682	1,244.5	75,274	< 0.1	393	399
100	-	-	4	0.1	444	562	1.1	1,180	1,191	-	-	0.1	704	705
robin	EG( $true$ )									$p1 = Ask$				
10	2.3	70,581	1	< 0.1	86	86	0.1	437	437	73.5	73,145	< 0.1	222	222
50	-	-	1	0.1	1,263	1,263	4.8	15,676	15,676	-	-	0.3	1,902	1,902
100	-	-	1	1.0	7,688	7,688	53.9	100,719	102,941	-	-	1.5	9,317	9,317
fms	EG( $\neg(P_1s = P_2s = P_3s = N)$ )									$P_1s = N$				
5	0.8	18,474	1	< 0.1	61	135	1.9	1,022	1,024	2.5	35,238	0.27	419	475
10	16.5	60,559	1	< 0.1	128	220	1,062.4	4,338	6,231	191.8	62,188	77.7	607	1,050
kanban	EG( $P_1out > 0 \vee P_2out > 0 \vee P_3out > 0 \vee P_4out > 0$ )									$P_1out = N$				
8	2.1	42,925	1	< 0.1	279	415	1,131.1	1,949	2,714	2.2	43,511	6.2	1,303	1,486
10	4.4	58,693	1	< 0.1	529	930	-	-	-	4.6	58,705	27.0	2,507	2,939

**Table 2.** Results for the EG computation.

result is not surprising due to the time and memory complexity of building the reachability relation, even if this is done using saturation.

Another experiment is provided to show the merit of our algorithm. We build a simple model with a long path from an SCC where  $EGp$  holds to a terminal state, with  $p$  holding on every state on this path. We parameterize the length of the path to control the number of iterations which a traditional EG algorithm will require to reach the fixpoint. For different numbers of iterations, from 500 to 50,000, we compare the runtimes (in seconds) of traditional (BFS) search and our algorithm (SatTR) in Figure 4. As the number of iterations grows, the runtime of our algorithm grows much slower than that of the traditional algorithm, due to the efficient state exploration scheme in constrained saturation.

If we consider fairness, as discussed in Section 4, the time and memory complexity of building the reachability relation is often reduced, while that of the traditional algorithm in NuSMV increases. The advantage of our algorithm is observable in this case, especially for some complex models which are not manageable in NuSMV.

## 6 Conclusion and future work

In this paper, we focus on symbolic CTL model checking based on the idea of the saturation algorithm. To constrain state exploration in a given set of states, we present a constrained saturation algorithm. The “check-and-fire” policy filters out the states not in the given set when saturating MDD nodes recursively. For the EG operator, we first symbolically build the reachability relation, using constrained saturation, then compute the set of states satisfying  $EGp$ . We discuss desirable properties of our new EU and EG algorithms and analyze a set of experimental results.

Constrained saturation enables building the reachability relation for some complex systems. The application of the reachability relation can be further extended to SCC construction, a basic problem in emptiness checking for Büchi automata. Another future work is to reduce the cost of building the reachability relation. For SCC enumeration,  $\mathcal{X}$  in  $R_{\mathcal{X},\mathcal{S}}^+$  can be refined to reduce the computation complexity.

## References

1. NuSMV: a new symbolic model checker. Available at <http://nusmv.iirst.itc.it/>.
2. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.
3. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *Int. Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, Aug. 1991. IFIP Transactions, North-Holland.
4. G. Ciardo et al. SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual. Available at <http://www.cs.ucr.edu/~ciardo/SMART/>.
5. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578–608, 2006.
6. G. Ciardo, G. Lüttgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31:63–100, 2007.
7. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In T. Margaria and W. Yi, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2031, pages 328–342, Genova, Italy, Apr. 2001. Springer-Verlag.
8. G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In H. Garavel and J. Hatcliff, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2619, pages 379–393, Warsaw, Poland, Apr. 2003. Springer-Verlag.
9. G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state space exploration. *Software Tools for Technology Transfer*, 8(1):4–25, Feb. 2006.
10. G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In W. Hunt, Jr. and F. Somenzi, editors, *Computer Aided Verification (CAV'03)*, LNCS 2725, pages 40–53, Boulder, CO, USA, July 2003. Springer-Verlag.
11. G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In D. Borriore and W. Paul, editors, *Proc. CHARME*, LNCS 3725, pages 146–161, Saarbrücken, Germany, Oct. 2005. Springer-Verlag.
12. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of LNCS, Copenhagen, Denmark, July 2002. Springer.
13. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992. CMU-CS-92-131.

14. M. Wan and G. Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In M. Nielsen et al., editors, *Proc. 35th Int. Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 5404, pages 582–594, Špindlerův Mlýn, Czech Republic, Feb. 2009. Springer-Verlag.