Structural Symbolic CTL Model Checking of Asynchronous Systems*

Gianfranco Ciardo and Radu Siminiceanu

Department of Computer Science, College of William and Mary
Williamsburg, VA 23187, USA
{ciardo, radu}@cs.wm.edu

Abstract. In previous work, we showed how structural information can be used to efficiently generate the state-space of asynchronous systems. Here, we apply these ideas to symbolic CTL model checking. Thanks to a Kronecker encoding of the transition relation, we detect and exploit event locality and apply better fixed-point iteration strategies, resulting in orders-of-magnitude reductions for both execution times and memory consumption in comparison to well-established tools such as NuSMV.

1 Introduction

Verifying the correctness of a system, either by proving that it refines a specification or by determining that it satisfies certain properties, is an important step in system design. *Model checking* is concerned with the tasks of representing a system with an automaton, usually finite-state, and then showing that the initial state of this automaton satisfies a *temporal logic* statement [13].

Model checking has gained increasing attention since the development of techniques based on *binary decision diagrams (BDDs)* [4]. *Symbolic model checking* [6] is known to be effective for *computation tree logic (CTL)* [12], as it allows for the efficient storage and manipulation of the large sets of states corresponding to CTL formulae. However, practical limitations still exist. First, memory and time requirements might be excessive when tackling real systems. This is especially true since the size (in number of nodes) of the BDD encoding the set of states corresponding to a CTL formula is usually much larger *during* the fixed-point iterations than upon convergence. This has spurred work on distributed/parallel algorithms for BDD manipulation and on verification techniques that use only a fraction of the BDD nodes that would be required in principle [3, 19].

Second, symbolic model checking has been quite successful for hardware verification but software, in particular distributed software, has so far been considered beyond reach. This is because the state space of software is much larger, but also because of the widely-held belief that symbolic techniques work well only in synchronous settings. We attempt to dispel this myth by showing that symbolic model checking based on the model *structure* copes well with asynchronous be-

* Work supported in part by the National Aeronautics and Space Administration under grants NAG-1-2168 and NAG-1-02095 and by the National Science Foundation under grants CCR-0219745 and ACI-0203971.

W.A. Hunt, Jr. and F. Somenzi (Eds.): CAV 2003, LNCS 2725, pp. 40–53, 2003.

havior and even benefits from it. Furthermore, the techniques we introduce excel at reducing the *peak number of nodes* in the fixed-point iterations.

The present contribution is based on our earlier work in symbolic state-space generation using *multivalued decision diagrams (MDDs)*, *Kronecker encoding* of the next state function [17, 8], and the *saturation* algorithm [9]. This background is summarized in Section 2, which also discusses how to exploit the model structure for MDD manipulation. Section 3 contains our main contribution: improved computation of the basic CTL operators using structural model information. Section 4 gives memory and runtime results for our algorithms implemented in SMART [7] and compares them with NuSMV [11].

2 Exploiting the Structure of Asynchronous Models

We consider globally-asynchronous locally-synchronous systems specified by a tuple $(\widehat{\mathcal{S}}, \mathcal{S}^{init}, \mathcal{E}, \mathcal{N})$, where the *potential state space* $\widehat{\mathcal{S}}$ is given by the product $\mathcal{S}_K \times \cdots \times \mathcal{S}_1$ of the K *local state spaces* of K *submodels*, i.e., a generic (global) state is $\mathbf{i} = (i_K, ..., i_1)$; $\mathcal{S}^{init} \subseteq \widehat{\mathcal{S}}$ is the set of *initial states*; $\mathcal{E}$ is a set of *(asynchronous) events*; the *next-state function* $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ is *disjunctively partitioned* [14] according to $\mathcal{E}$, i.e., $\mathcal{N} = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha$, where $\mathcal{N}_\alpha(\mathbf{i})$ is the set of states that can be reached when event α *fires* in state $\mathbf{i}$; we say that α is *disabled* in $\mathbf{i}$ if $\mathcal{N}_\alpha(\mathbf{i}) = \emptyset$.

With high-level models such as Petri nets or pseudo-code, the sets $\mathcal{S}_k$, for $K \geq k \geq 1$, might not be known a priori. Their derivation alongside the construction of the *(actual) state space* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$, defined by $\mathcal{S} = \mathcal{S}^{init} \cup \mathcal{N}(\mathcal{S}^{init}) \cup \mathcal{N}^2(\mathcal{S}^{init}) \cup \cdots = \mathcal{N}^*(\mathcal{S}^{init})$, where $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}(\mathbf{i})$, is an interesting problem in itself [10]. Here, we assume that each $\mathcal{S}_k$ is known and of finite size n_k and map its elements to $\{0, ..., n_k - 1\}$ for notational simplicity and efficiency.

Symbolic model checking manages subsets of $\widehat{\mathcal{S}}$ and relations over $\widehat{\mathcal{S}}$. In the binary case, these are simply subsets of $\{0,1\}^K$ and of $\{0,1\}^{2K}$, respectively, and are encoded as BDDs. Our structural approach instead uses MDDs to store sets and (boolean) sums of Kronecker matrix products to store relations. The use of MDDs has been proposed before [15], but their implementation through BDDs made them little more than a "user interface". In [17], we showed instead that implementing MDDs directly may increase "locality", thus the efficiency of state-space generation, if paired with our Kronecker encoding of $\mathcal{N}$. We use *quasi-reduced ordered* MDDs, directed acyclic edge-labeled multi-graphs where:

- Nodes are organized into $K + 1$ *levels*. We write $\langle k|p \rangle$ to denote a generic node, where k is the level and p is a unique index for a node at that level.
- Level K contains only a single *non-terminal* node $\langle K|r \rangle$, the *root*, whereas levels $K-1$ through 1 contain one or more non-terminal nodes.
- Level 0 consists of the two *terminal* nodes, $\langle 0|0 \rangle$ and $\langle 0|1 \rangle$.
- A non-terminal node $\langle k|p \rangle$ has n_k arcs, labeled from 0 to $n_k - 1$, pointing to nodes at level $k-1$. If the arc labeled i_k points to node $\langle k-1|q \rangle$, we write $\langle k|p \rangle[i_k] = q$. *Duplicate* nodes are not allowed but, unlike the *(strictly)*

reduced ordered decision diagrams of [15], *redundant* nodes where all arcs point to the same node are allowed (both versions are canonical [16]).

Let $\mathcal{A}(\langle k|p\rangle)$ be the set of tuples $(i_K, ..., i_{k+1})$ labeling paths from $\langle K|r\rangle$ to node $\langle k|p\rangle$, and $\mathcal{B}(\langle k|p\rangle)$ the set of tuples $(i_k, ..., i_1)$ labeling paths from $\langle k|p\rangle$ to $\langle 0|1\rangle$. In particular, $\mathcal{B}(\langle K|r\rangle)$ and $\mathcal{A}(\langle 0|1\rangle)$ specify the states encoded by the MDD.

A more drastic departure from traditional symbolic approaches is our encoding of $\mathcal{N}$ [17], inspired by the representation of the transition rate matrix for a continuous-time Markov chain by means of a (real) sum of Kronecker products [5, 18]. This requires a *Kronecker-consistent* decomposition of the model into submodels, i.e., there must exist functions $\mathcal{N}_{\alpha,k} : \mathcal{S}_k \to 2^{\mathcal{S}_k}$, for $\alpha \in \mathcal{E}$ and $K \geq k \geq 1$, such that, for any $\mathbf{i} \in \widehat{\mathcal{S}}$, $\mathcal{N}_\alpha(\mathbf{i}) = \mathcal{N}_{\alpha,K}(i_K) \times \cdots \times \mathcal{N}_{\alpha,1}(i_1)$.

$\mathcal{N}_{\alpha,k}(i_k)$ represents the set of local states locally reachable (i.e., for submodel k in isolation) from local state i_k when α fires. In particular, α is disabled in any global state whose k^{th} component i_k satisfies $\mathcal{N}_{\alpha,k}(i_k) = \emptyset$. This consistency requirement is quite natural for asynchronous systems. Indeed, it is always satisfied by formalisms such as Petri nets, for which any partition of the P places of the net into $K \leq P$ subsets is consistent. We define the boolean incidence matrices $\mathbf{W}_{\alpha,k} \in \{0,1\}^{\mathcal{S}_k \times \mathcal{S}_k}$ so that $\mathbf{W}_{\alpha,k}[i_k, j_k] = 1$ iff $j_k \in \mathcal{N}_{\alpha,k}(i_k)$. Then, $\mathbf{W} = \bigvee_{\alpha\in\mathcal{E}} \bigotimes_{K\geq k\geq 1} \mathbf{W}_{\alpha,k}$ encodes $\mathcal{N}$, i.e., $\mathbf{W}[\mathbf{i}, \mathbf{j}] = 1$ iff $\mathbf{j} \in \mathcal{N}(\mathbf{i})$, where $\otimes$ denotes the Kronecker product of matrices, and the mixed-base value $\sum_{k=1}^{K} i_k \prod_{l=1}^{k-1} n_l$ of $\mathbf{i}$ is used when indexing $\mathbf{W}$.

2.1 Locality, In-Place-Updates, and Saturation

One important advantage of our Kronecker encoding is its ability to evidence the *locality* of events inherently present in most asynchronous systems. We say that an event α is *independent* of level k if $\mathbf{W}_{\alpha,k}$ is the identity matrix; this means that the k^{th} local state does not affect the enabling of α, nor is modified by the firing of α. We then define $Top(\alpha)$ and $Bot(\alpha)$ to be the maximum and minimum levels on which α depends. Since an event must be able to modify at least some local state to be meaningful, we can assume that these levels are always well defined, i.e., $K \geq Top(\alpha) \geq Bot(\alpha) \geq 1$.

One advantage of our encoding is that, for practical asynchronous systems, most $\mathbf{W}_{\alpha,k}$ are the identity matrix (thus do not need to be stored explicitly) while the rest usually have very few nonzero entries per row (thus can be stored with sparse data structures). This is much more compact than the BDD or MDD storage of $\mathcal{N}$. In a BDD representation of $\mathcal{N}_\alpha$, for example, an edge skipping levels k and k' (the k^{th} components of the "from" and "to" states, respectively) means that, after α fires, the k^{th} component can be either 0 or 1, regardless of whether it was 0 or 1 before the firing. The more common behavior is instead the one where 0 remains 0 and 1 remains 1, the default in our Kronecker encoding.

In addition to reducing the memory requirements to encode $\mathcal{N}$, the Kronecker encoding allows us to exploit locality to reduce the execution time when generating the state space. In [17], we performed an iteration of the form

repeat
  for each $\alpha \in \mathcal{E}$ do $\boxed{\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{N}_\alpha(\mathcal{S})}$
until $\mathcal{S}$ does not change

with $\mathcal{S}$ initialized to $\mathcal{S}^{init}$. If $Top(\alpha) = k$, $Bot(\alpha) = l$, and $\mathbf{i} \in \mathcal{S}$, then, for any $\mathbf{j} \in \mathcal{N}_\alpha(\mathbf{i})$ we have $\mathbf{j} = (i_K, ..., i_{k+1}, j_k, ..., j_l, i_{l-1}, ...i_1)$. Thus, we descend from the root of the MDD encoding the current $\mathcal{S}$ and, only when encountering a node $\langle k|p\rangle$ we call the recursive function $Fire(\alpha, \langle k|p\rangle)$ to compute the resulting node at the same level k using the information encoded by $\mathbf{W}_{\alpha,k}$; furthermore, after processing a node $\langle l|q\rangle$, with $l = Bot(\alpha)$, the recursive *Fire* calls stop.

In [8], we gained further efficiency by performing *in-place updates* of (some) MDD nodes. This is based on the observation that, for any other $\mathbf{i}' \in \mathcal{S}$ whose last k components coincide with those of $\mathbf{i}$ and whose first $K-k$ components $(i'_K, ..., i'_{k+1})$ lead to the same node $\langle k|p\rangle$ as $(i_K, ..., i_{k+1})$, we can immediately conclude that $\mathbf{j}' = (i'_K, ..., i'_{k+1}, j_k, ..., j_l, i_{l-1}, ...i_1)$ is also reachable. Thus, we performed an iteration of the form (let $\mathcal{E}_k = \{\alpha : Top(\alpha) = k\}$)

repeat
  for $k = 1$ to K do
    for each node $\langle k|p\rangle$ do
      for each $\alpha \in \mathcal{E}_k$ do $\boxed{\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{A}(\langle k|p\rangle) \times Fire(\alpha, \langle k|p\rangle)}$
until $\mathcal{S}$ does not change

where the "$\mathcal{A}(\langle k|p\rangle)\times$" operation comes at no cost, since it is implied by starting the firing of α "in the middle of the MDD" and directly updating node $\langle k|p\rangle$.

The memory and time savings due to in-place updates are compounded to those due to locality. Especially when studying asynchronous systems with "tall" MDDs (large K), this results in orders-of-magnitude improvements with respect to traditional symbolic approaches. However, even greater savings are achieved by *saturation* [9], a new iteration control strategy made possible by the use of structural model information. A node $\langle k|p\rangle$ is *saturated* if it is a fixed point with respect to firing any event that is independent of all levels above k:

$$\forall l, k \geq l \geq 1,\ \forall \alpha \in \mathcal{E}_l,\ \mathcal{A}(\langle k|p\rangle)\times\mathcal{B}(\langle k|p\rangle) \supseteq \mathcal{N}_\alpha(\ \mathcal{A}(\langle k|p\rangle)\times\mathcal{B}(\langle k|p\rangle)\ ).$$

With saturation, the traditional global fixed-point iteration for the overall MDD disappears. Instead, we start saturating the node at level 1 (assuming $|\mathcal{S}^{init}| = 1$, the initial MDD contains one node per level), move up in the MDD saturating nodes, and end the process when we have saturated the root. To saturate a node $\langle k|p\rangle$, we exhaustively fire each event $\alpha \in \mathcal{E}_k$ in it, using in-place updates at level k. Each required $Fire(\alpha, \langle k|p\rangle)$ call may create nodes at lower levels, which are recursively saturated before completing the *Fire* call itself (see Fig. 1).

Saturation has numerous advantages over traditional methods, resulting in enormous memory and time savings. Once $\langle k|p\rangle$ is saturated, we never fire an event $\alpha \in \mathcal{E}_k$ in it again. Only saturated nodes appear in the unique table and operation caches. Finally, most of these nodes will still be present in the final MDD (non-saturated nodes are *guaranteed* not to be part of it). In fact, the peak and final number of nodes differ by a mere constant in some models [9].

$$\mathbf{W}_{a,3} = \mathbf{I} \quad \mathbf{W}_{b,3} = \mathbf{I} \quad \mathbf{W}_{c,3} = \mathbf{I} \quad \mathbf{W}_{d,3} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \quad \mathbf{W}_{e,3} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

$$\mathbf{W}_{a,2} = \mathbf{I} \quad \mathbf{W}_{b,2} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \mathbf{W}_{c,2} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{W}_{d,2} = \mathbf{I} \quad \mathbf{W}_{e,2} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{W}_{a,1} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad \mathbf{W}_{b,1} = \mathbf{I} \quad \mathbf{W}_{c,1} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \mathbf{W}_{d,1} = \mathbf{I} \quad \mathbf{W}_{e,1} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

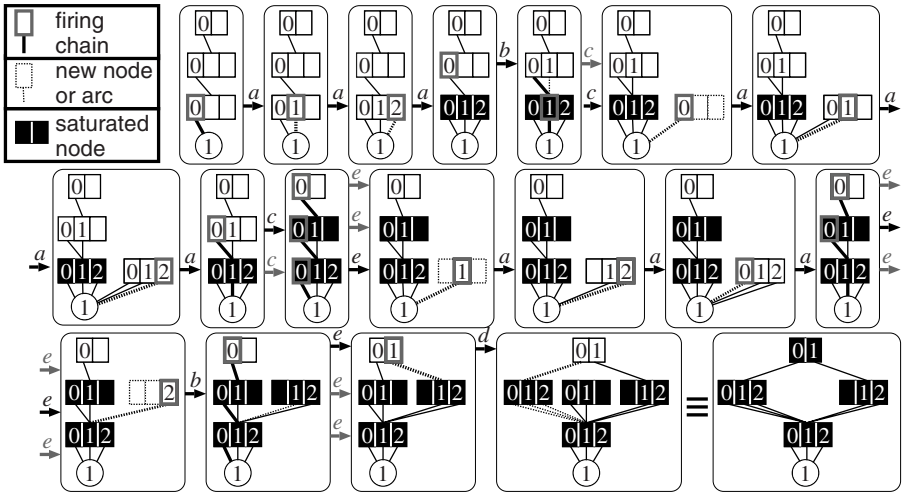

Fig. 1: Encoding of $\mathcal{N}$ and generation of $\mathcal{S}$ using saturation: $K = 3$, $\mathcal{E} = \{a, b, c, d, e\}$. Arrows between frames indicate the event being fired (a darker shade is used for the event label on the "active" level in the firing chain). The firing sequence is: a (3 times), b, c (at level 1), a (interrupting c, 3 times to saturate a new node), c (resumed, at level 2), e (at level 1) a (interrupting e, 3 times to saturate a new node) e (resumed, at level 2), b (interrupting e), e (resumed, at level 3), and finally d (the union of 0 1 2 and 0 1 2 at level 2, i.e., 0 1 2, is saturated by definition). There is at most one unsaturated node per level, and the one at the lowest level is being saturated.

3 Structural-Based CTL Model Checking

After having summarized the distinguishing features of the data structures and algorithms we employ for state-space generation, we now consider how to apply them to symbolic model checking. CTL [12] is widely used due to its simple yet expressive syntax and to the existence of efficient algorithms for its analysis [6]. In CTL, operators occur in pairs: the path quantifier, either A (on all paths) or E (there exists a path), is followed by the tense operator, one of X (next), F (future, or finally), G (globally, or generally), and U (until). Of the eight possible pairings, only a *generator* (sub)set needs to be implemented in a model checker, as the remaining operators can be expressed in terms of those in the set [13]. $\{EX, EU, EG\}$ is such a set, but the following discusses also EF for clarity.

3.1 The *EX* Operator

Semantics: $\mathbf{i}^0 \models EXp$ iff $\exists \mathbf{i}^1 \in \mathcal{N}(\mathbf{i}^0)$ s.t. $\mathbf{i}^1 \models p$. ("$\models$" means "satisfies")

In our notation, EX corresponds to the inverse function of $\mathcal{N}$, the previous-state function, $\mathcal{N}^{-1}$. With our Kronecker matrix encoding, the inverse of $\mathcal{N}_\alpha$ is simply obtained by transposing the incidence matrices $\mathbf{W}_{\alpha,k}$ in the Kronecker product, thus $\mathcal{N}^{-1}$ is encoded as $\bigvee_{\alpha\in\mathcal{E}} \bigotimes_{K\geq k\geq 1} \mathbf{W}^T_{\alpha,k}$.

To compute the set of states where EXp is satisfied, we can follow the same idea used to fire events in an MDD node during our state-space generation: given the set $\mathcal{P}$ of states satisfying formula p, we can accumulate the effect of "firing backward" each event by taking advantage of locality and in-place updates. This results in an efficient calculation of EX. Computing its reflexive and transitive closure, that is, the backward reachability operator EF, is a much more difficult challenge, which we consider next.

3.2 The *EF* Operator

Semantics: $\mathbf{i}^0 \models EFp$ iff $\exists n \geq 0, \exists \mathbf{i}^1 \in \mathcal{N}(\mathbf{i}^0), ..., \exists \mathbf{i}^n \in \mathcal{N}(\mathbf{i}^{n-1})$ s.t. $\mathbf{i}^n \models p$.

In our approach, the construction of the set of states satisfying EFp is analogous to the saturation algorithm for state-space generation, with two differences. Besides using the transposed incidence matrices $\mathbf{W}^T_{\alpha,k}$, the execution starts with the set $\mathcal{P}$, not a single state. These differences do not affect the applicability of saturation, which retains all its substantial time and memory benefits.

3.3 The *EU* Operator

Semantics: $\mathbf{i}^0 \models E[p\,U\,q]$ iff $\exists n \geq 0, \exists \mathbf{i}^1 \in \mathcal{N}(\mathbf{i}^0), ..., \exists \mathbf{i}^n \in \mathcal{N}(\mathbf{i}^{n-1})$ s.t. $\mathbf{i}^n \models q$ and $\mathbf{i}^m \models p$ for all $m < n$. (in particular, $\mathbf{i} \models q$ implies $\mathbf{i} \models E[p\,U\,q]$)

The traditional computation of the set of states satisfying $E[p\,U\,q]$ uses a least fixed point algorithm. Starting with the set $\mathcal{Q}$ of states satisfying q, it iteratively adds all the states that reach them on paths where property p holds (see Algorithm *EUtrad* in Fig. 2). The number of iterations to reach the fixed point is $\max_{\mathbf{i}\in\widehat{\mathcal{S}}} \left(\min\left\{n \mid \exists\, \mathbf{i}^0 \in \mathcal{Q} \wedge \forall\, 0<m\leq n, \exists\, \mathbf{i}^m \in \mathcal{N}^{-1}(\mathbf{i}^{m-1}) \cap \mathcal{P} \wedge \mathbf{i} = \mathbf{i}^n\right\}\right)$.

EUtrad(in $\mathcal{P}, \mathcal{Q}$: set of *state*) : set of *state*
1. declare $\mathcal{X}, \mathcal{Y}$: set of *state*;
2. $\mathcal{X} \leftarrow \mathcal{Q}$; • *initialize $\mathcal{X}$ with all states in $\mathcal{Q}$*
3. repeat
4. $\quad \mathcal{Y} \leftarrow \mathcal{X}$;
5. $\quad \mathcal{X} \leftarrow \mathcal{X} \cup (\mathcal{N}^{-1}(\mathcal{X}) \cap \mathcal{P})$; • *add predecessors of states in $\mathcal{X}$ that are in $\mathcal{P}$*
6. until $\mathcal{Y} = \mathcal{X}$;
7. return $\mathcal{X}$;

Fig. 2: Traditional algorithm to compute the set of states satisfying $E[p\,U\,q]$.

Applying Saturation to *EU*. As the main contribution of this paper, we propose a new approach to computing *EU* based on saturation. The challenge in applying saturation arises from the need to "filter out" states not in $\mathcal{P}$ (line 5 of Algorithm *EUtrad*): as soon as a new predecessor of the working set $\mathcal{X}$ is obtained, it must be intersected with $\mathcal{P}$. Failure to do so can result in paths to $\mathcal{Q}$ that stray, even temporarily, out of $\mathcal{P}$. However, saturation works in a highly localized manner, adding states out of breadth-first-search (BFS) order. Performing an expensive intersection after each firing would add enormous overhead, since our firings are very lightweight operations. To cope with this problem, we propose a "partial" saturation that is applied to a subset of events for which no filtering is needed. These are the events whose firing is *guaranteed* to preserve the validity of the formula p. For the remaining events, BFS with filtration must be used. The resulting global fixed point iteration interleaves these two phases (see Fig. 3). The following classification of events is analogous to, but different from, the *visible* vs. *invisible* one proposed for partial order reduction [1].

Definition 1 In a discrete state model $(\widehat{\mathcal{S}}, \mathcal{S}^{init}, \mathcal{E}, \mathcal{N})$, an event α is *dead* with respect to a set of states $\mathcal{X}$ if there is no state in $\mathcal{X}$ from which its firing leads to a state in $\mathcal{X}$, i.e., $\mathcal{N}_\alpha^{-1}(\mathcal{X}) \cap \mathcal{X} = \emptyset$ (this includes the case where α is always disabled in $\mathcal{X}$); it is *safe* if it is not dead and its firing cannot lead from a state not in $\mathcal{X}$ to a state in $\mathcal{X}$, i.e., $\emptyset \subset \mathcal{N}_\alpha^{-1}(\mathcal{X}) \subseteq \mathcal{X}$; it is *unsafe* otherwise, i.e., $\mathcal{N}_\alpha^{-1}(\mathcal{X}) \setminus \mathcal{X} \neq \emptyset \wedge \mathcal{N}_\alpha^{-1}(\mathcal{X}) \cap \mathcal{X} \neq \emptyset$. □

Given a formula $E[p\,U\,q]$, we first classify the safety of events through static analysis. Then, each *EU* fixed point iteration consists of two backward steps: BFS on unsafe events followed by saturation on safe events. Since saturation is in turn a fixed point computation, the resulting algorithm computes a nested fixed point. Note that the operators used in both steps are monotonic (the working set $\mathcal{X}$ is increasing), a condition for applying saturation and in-place updates.

Note 1 Dead events can be ignored altogether by our *EUsat* algorithm, since the working set $\mathcal{X}$ is always a subset of $\mathcal{P} \cup \mathcal{Q}$.

Note 2 The *Saturate* procedure in line 10 of *EUsat* is analogous to the one we use for *EF*, except that it is restricted to a subset $\mathcal{E}_S$ of events.

Note 3 *ClassifyEvents* has the same time complexity as one *EX* step and is called only once prior to the fixed point iterations.

Note 4 To simplify the description of *EUsat*, we call *ClassifyEvents* with the filter $\mathcal{P} \cup \mathcal{Q}$, i.e., $\mathcal{E}_S = \{\alpha : \emptyset \subset \mathcal{N}_\alpha^{-1}(\mathcal{P} \cup \mathcal{Q}) \subseteq \mathcal{P} \cup \mathcal{Q}\}$. With a slightly more complex initialization in *EUsat*, we could use instead the smaller filter $\mathcal{P}$, i.e., $\mathcal{E}_S = \{\alpha : \emptyset \subset \mathcal{N}_\alpha^{-1}(\mathcal{P}) \subseteq \mathcal{P}\}$. In practice, both sets of events could be computed. Then, if one is a strict superset of the other, it should be used, since the larger $\mathcal{E}_S$ is, the more *EUsat* behaves like our efficient *EF* saturation; otherwise, some heuristic must be used to choose between the two.

```
ClassifyEvents(in 𝒳 : set of state, out ℰ_U, ℰ_S : set of event)
1. ℰ_S ← ∅;   ℰ_U ← ∅;                    • initialize safe and unsafe sets of events
2. for each event α ∈ ℰ                    • determine safe and unsafe events, the rest are dead
3.     if (∅ ⊂ 𝒩_α^{-1}(𝒳) ⊆ 𝒳)
4.         ℰ_S ← ℰ_S ∪ {α};                                  • safe event w.r.t. 𝒳
5.     else if (𝒩_α^{-1}(𝒳) ∩ 𝒳 ≠ ∅)
6.         ℰ_U ← ℰ_U ∪ {α};                                  • unsafe event w.r.t. 𝒳
```

```
EUsat(in 𝒫, 𝒬 : set of state) : set of state
1. declare 𝒳, 𝒴 : set of state;
2. declare ℰ_U, ℰ_S : set of event;
3. ClassifyEvents(𝒫 ∪ 𝒬, ℰ_U, ℰ_S);
4. 𝒳 ← 𝒬;
5. Saturate(𝒳, ℰ_S);      • initialize 𝒳 with all states at unsafe distance 0 from 𝒬
6. repeat
7.     𝒴 ← 𝒳;
8.     𝒳 ← 𝒳 ∪ (𝒩_U^{-1}(𝒳) ∩ (𝒫 ∪ 𝒬)); • perform one unsafe backward BFS step
9.     if 𝒳 ≠ 𝒴 then
10.        Saturate(𝒳, ℰ_S);          • perform one safe backward saturation step
11. until 𝒴 = 𝒳;
12. return 𝒳;
```

Fig. 3: Saturation-based algorithm to compute the set of states satisfying $E[p\,U\,q]$.

Note 5 The number of *EUsat* iterations is 1 plus the "unsafe distance from $\mathcal{P}$ to $\mathcal{Q}$", $\max_{\mathbf{i}\in\mathcal{P}}(\min\{n|\exists\mathbf{i}^0\in\mathcal{R}_S^*(\mathcal{Q})\wedge\forall 0<m\leq n,\exists\mathbf{i}^m\in\mathcal{R}_S^*(\mathcal{R}_U(\mathbf{i}^{m-1})\cap\mathcal{P})\wedge\mathbf{i}=\mathbf{i}^n\})$, where $\mathcal{R}_S(\mathcal{X})=\bigcup_{\alpha\in\mathcal{E}_S}\mathcal{N}_\alpha^{-1}(\mathcal{X})$ and $\mathcal{R}_U(\mathcal{X})=\bigcup_{\alpha\in\mathcal{E}_U}\mathcal{N}_\alpha^{-1}(\mathcal{X})$ are the sets of "safe predecessors" and "unsafe predecessors" of $\mathcal{X}$, respectively.

Lemma 1 Iteration d of *EUsat* finds all states $\mathbf{i}$ at unsafe distance d from $\mathcal{Q}$.

Proof. By induction on d. Base: $d=0\Rightarrow\mathbf{i}\in\mathcal{R}_S^*(\mathcal{Q})$ which is a subset of $\mathcal{X}$ (lines 4,5). Inductive step: suppose all states at unsafe distance $m\leq d$ are added to $\mathcal{X}$ in the m^{th} iteration. By definition, a state $\mathbf{i}$ at unsafe distance $d+1$ satisfies: $\exists\mathbf{i}^0\in\mathcal{R}_S^*(\mathcal{Q})\wedge\forall 0<m\leq d+1,\exists\mathbf{j}^m\in\mathcal{R}_U(\mathbf{i}^{m-1})\cap\mathcal{P},\exists\mathbf{i}^m\in\mathcal{R}_S^*(\mathbf{j}^m)$, and $\mathbf{i}=\mathbf{i}^{d+1}$. Then, $\mathbf{i}^m$ and $\mathbf{j}^m$ are at unsafe distance m. By the induction hypothesis, they are added to $\mathcal{X}$ in iteration m. In particular, $\mathbf{i}^d$ is a new state found in iteration d. This implies that the algorithm must execute another iteration, which finds $\mathbf{j}^{d+1}$ as an unsafe predecessor of $\mathbf{i}^d$ (line 8). Since $\mathbf{i}$ is either $\mathbf{j}^{d+1}$ or can reach it through safe events alone, it is added to $\mathcal{X}$ (line 10). □

Theorem 1 Algorithm *EUsat* returns the set $\mathcal{X}$ of states satisfying $E[p\,U\,q]$.

Proof. It is immediate to see that *EUsat* terminates, since its working set is a monotonically increasing subset of $\widehat{\mathcal{S}}$, which is finite. Let $\mathcal{Y}$ be the set of states satisfying $E[p\,U\,q]$. We have (i) $\mathcal{Q}\subseteq\mathcal{X}$ (line 4) (ii) every state in $\mathcal{X}$ can reach a state in $\mathcal{Q}$ through a path in $\mathcal{X}$, and (iii) $\mathcal{X}\subseteq\mathcal{P}\cup\mathcal{Q}$ (lines 8,10). This implies

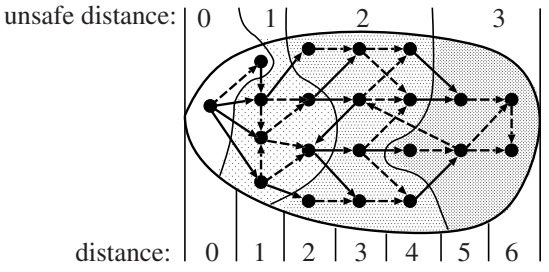

Fig. 4: Comparing BFS and saturation order: distance vs. unsafe distance.

$\mathcal{X} \subseteq \mathcal{Y}$. Since any state in $\mathcal{Y}$ is at some finite unsafe distance d from $\mathcal{Q}$, by Lemma 1 we conclude that $\mathcal{Y} \subseteq \mathcal{X}$. The two set inclusions imply $\mathcal{X} = \mathcal{Y}$. □

Figure 4 illustrates the way our exploration differs from BFS. Solid and dashed arcs represent unsafe and safe transitions, respectively. The shaded areas encircle the explored regions after each iteration of *EUsat*, four in this case. *EUtrad* would instead require seven iterations to explore the entire graph (states are aligned vertically according to their BFS depth).

Note 6 Our approach exhibits "graceful degradation". In the best case, all events are safe, and *EUsat* performs just one saturation step and stops. This happens for example when $p \vee q \equiv true$, which includes the special case $p \equiv true$. As $E[true\ U\ q] \equiv EFq$, we simply perform backward reachability from $\mathcal{Q}$ using saturation on the entire set of events. In the worst case, all events are unsafe, and *EUsat* performs the same steps as *EUtrad*. But even then, locality and our Kronecker encoding can still substantially improve the efficiency of the algorithm.

3.4 The *EG* Operator

Semantics: $\mathbf{i}^0 \models EGp$ iff $\forall n > 0\ , \exists \mathbf{i}^n \in \mathcal{N}(\mathbf{i}^{n-1})$ s.t. $\mathbf{i}^n \models p$.

In graph terms, consider the reachability subgraph obtained by restricting the transition relation to states in $\mathcal{P}$. Then, EGp holds in any state belonging to, or reaching, a nontrivial strongly connected component (SCC) of this subgraph.

Algorithm *EGtrad* in Fig. 5 shows the traditional greatest fixed point iteration. It initializes the working set $\mathcal{X}$ with all states in $\mathcal{P}$ and gradually eliminates states that have no successor in $\mathcal{X}$ until only the SCCs of $\mathcal{P}$ and their incoming paths along states in $\mathcal{P}$ are left. The number of iterations equals the maximum length of any path over $\mathcal{P}$ that does *not* lead to such an SCC.

Applying Saturation to *EG*. *EGtrad* is a greatest fixed point, so to speed it up we must *eliminate* unwanted states faster. The criterion for a state $\mathbf{i}$ is a *conjunction*: $\mathbf{i}$ should be eliminated if *all* its successors are not in $\mathcal{P}$. Since it considers a single event at a time and makes local decisions that must be globally correct, it would appear that saturation cannot be used to improve *EGtrad*.

```
EGtrad(in P : set of state) : set of state
1. declare X, Y : set of state;
2. X ← P;
3. repeat
4.     Y ← X;
5.     X ← N^{-1}(X) ∩ P;
6. until Y = X;
7. return X;
```

```
EGsat(in P : set of state) : set of state
1. declare X, Y, C, T : set of state;
2. C ← EUsat(P, {i ∈ P : i ∈ N(i)});
3. T ← ∅;
4. while ∃i ∈ P \ (C ∪ T) do
5.     X ← EUsat(P \ C, {i});
6.     Y ← ESsat(P \ C, {i});
7.     if |X ∩ Y| > 1 then
8.         C ← C ∪ X;
9.     else
10.        T ← T ∪ {i};
11. return C;
```

Fig. 5: Traditional and saturation-based EG algorithms.

However, Fig. 5 shows an algorithm for EG which, like [2, 20], enumerates the SCCs by finding forward and backward reachable sets from a state. However, it uses saturation, instead of breadth-first search. In line 2, Algorithm *EGsat* disposes of selfloop states in $\mathcal{P}$ and of the states reaching them through paths in $\mathcal{P}$ (selfloops can be found by performing EX using a modified set of matrices $\mathbf{W}_{\alpha,k}$ where off-diagonal entries are set to zero). Then, it chooses a single state $\mathbf{i} \in \mathcal{P}$ and builds the backward and forward reachable sets from $\mathbf{i}$ restricted to $\mathcal{P}$, using *EUsat* and *ESsat* (ES is the dual in the past of EU; it differs from *EUsat* only in that it does not transpose the matrices $\mathbf{W}_{\alpha,k}$). If $\mathcal{X}$ and $\mathcal{Y}$ have more than just $\mathbf{i}$ in common, $\mathbf{i}$ belongs to a nontrivial SCC and all of $\mathcal{X}$ is part of our answer $\mathcal{C}$. Otherwise, we add $\mathbf{i}$ to the set $\mathcal{T}$ of trivial SCCs ($\mathbf{i}$ might nevertheless reach a nontrivial SCC, in $\mathcal{Y}$, but we have no easy way to tell). The process ends when $\mathcal{P}$ has been partitioned into $\mathcal{C}$, containing nontrivial SCCs and states reaching them over $\mathcal{P}$, and $\mathcal{T}$, containing trivial SCCs.

EGsat is more efficient than our *EGtrad* only in special cases. An example is when the *EUsat* and *ESsat* calls in *EGsat* find each next state on a long path of trivial SCCs through a single lightweight firing, while *EGtrad* always attempts firing each event at each iteration. In the worst case, however, *EGsat* can be much worse than not only *EGtrad*, but even an explicit approach. For this reason, the next section discusses only *EGtrad*, which is guaranteed to benefit from locality and the Kronecker encoding.

4 Results

We implemented our algorithms in SMART [7] and compared them with NuSMV (version 2.1.2), on a 2.2 GHz Pentium IV Linux workstation with 1GB of RAM. Our examples are chosen from the world of distributed systems and protocols. Each system is modeled in the SMART and NuSMV input languages. We verified that the two models are equivalent, by checking that they have the same sets of potential and reachable states and the same transition relation.

We briefly describe the chosen models and their characteristics. Detailed descriptions can be found in [7]. The randomized asynchronous leader election protocol solves the problem of designating a unique leader among N participants by sending messages along a unidirectional ring. The dining philosophers and the round robin protocol models solve a specific type of mutual exclusion problem among N processes. The slotted ring models a communication protocol in a network of N nodes. The flexible manufacturing system model describes a factory with three production units where N parts of each of three different types move around on pallets (for compatibility with NuSMV, we had to change immediate events in the original SMART model [7] into timed ones). This is the only model where the number of levels in the MDD is fixed, not depending on N (of course, the size of the local state spaces $\mathcal{S}_k$, depends instead on N). All of these models are characterized by loose connectivity between components, i.e., they are examples of globally-asynchronous locally-synchronous systems. We used the best known variable ordering for SMART and NuSMV (they coincide in all models except round robin, where, for best performance, NuSMV uses the reverse of the one for SMART). The time to build the encoding of $\mathcal{N}$ is not included in the table; while this time is negligible for our Kronecker encoding, it can be quite substantial for NuSMV, at times exceeding the reported runtimes.

Table 4 shows the state-space size, runtime (sec), and peak memory consumption (MB) for the five models, counting MDD nodes plus Kronecker matrices in SMART, and BDD nodes in NuSMV. There are three sets of columns: state-space generation (analogous to EF), EU, and EG. See [7] for the meaning of the atomic propositions. In NuSMV, it is possible to evaluate EU expressions without explicitly building the state space first; however, this substantially increases the runtime, so that it almost equals the sum of the state-space generation and EU entries. The same holds for EG. In SMART the state-space construction is always executed in advance, hence the memory consumption includes the MDD for the state space. We show the largest parameter for which NuSMV can build the state space in the penultimate row of each model, while the last row shows the largest parameter for which SMART can evaluate the EU and EG.

Overall, SMART outperforms NuSMV time- and memory-wise. Saturation excels at state-space generation, with improvements exceeding 100,000 in time and 1,000 in memory. Indeed, SMART can scale N even more than shown, e.g., 10 for the leader election, 10,000 for philosophers, 200 for round robin, and 150 for FMS. For EU, we can see the effect of the data structures and of the algorithm separately, since we report for both *EUtrad* and *EUsat*. When comparing the two, the latter reaches a fixed point in fewer iterations (recall Fig. 4) and uses less memory. While each *EUsat* iteration is more complex, it also operates on smaller MDDs, one of the benefits of saturation. The performance gain is more evident in large models, where *EUtrad* runs out of memory before completing the task and is up to 20 times slower. The comparison between our *EUtrad*, *EGtrad* and NuSMV highlights instead the differences between data structures. SMART is still faster and uses much less memory, suggesting that the Kronecker representation for the transition relation is much more efficient than the $2K$-level BDD representation.

N	$\lvert\mathcal{S}\rvert$	State-space generation				EU query										EG query					
		NuSMV		SMART		NuSMV				SMART						NuSMV				SMART	
						after SS		alone		*EUtrad*			*EUsat*			after SS		alone		*EGtrad*	
		time	mem	time	mem	time	mem	time	mem	iter	time	mem	iter	time	mem	time	mem	time	mem	time	mem
Leader: $K = 2N$, $\lvert\mathcal{E}\rvert = N^2 + 13N$						$E[(pref_1 = 0)\,U\,(status_0 = leader)]$										$EG(status_0 \neq leader)$					
3	8.49×10^{2}	0.1	2	0.04	<.5	0.1	3	18.3	12	43	0.02	<.5	22	0.02	<.5	0.2	4	0.7	4	0.02	<.5
4	1.15×10^{4}	2.1	10	0.27	<.5	2.3	11	8104.7	371	62	0.36	1	38	0.27	1	232.8	12	1189.1	235	0.11	2
5	1.50×10^{5}	56.0	29	1.49	1	52.0	33	—	—	81	3.74	7	52	3.09	7	18023.6	104	—	—	0.44	9
6	1.89×10^{6}	1063.7	295	7.35	3	—	—	—	—	101	46.90	30	66	35.67	28	—	—	—	—	1.64	38
7	2.39×10^{7}	—	—	40.64	7	—	—	—	—	121	690.85	116	85	416.85	101	—	—	—	—	7.15	128
Philosophers: $K = \lceil N/2 \rceil$, $\lvert\mathcal{E}\rvert = 4N$						$E[(phil_1 \neq eat)\,U\,(phil_0 = eat)]$										$EG(phil_0 \neq eat)$ (starvation)					
20	3.46×10^{12}	0.8	6	0.02	<.5	0.1	7	0.8	6	40	0.03	<.5	4	0.02	<.5	0.1	8	1.1	6	0.01	<.5
50	2.23×10^{31}	36.0	46	0.07	<.5	1.2	46	39.7	46	100	0.17	1	4	0.06	1	0.9	46	132.3	50	0.02	1
100	4.96×10^{62}	1134.8	316	0.15	<.5	7.9	316	1121.8	316	200	0.67	3	4	0.14	3	9.0	316	2525.3	358	0.05	3
500	3.03×10^{313}	—	—	1.01	1	—	—	—	—	1000	19.09	78	4	0.77	60	—	—	—	—	0.28	58
Slotted ring: $K = N$, $\lvert\mathcal{E}\rvert = 8N$						$E[(slot_1 \neq bf)\,U\,(slot_0 = ag)]$										$EG(slot_0 \neq hg)$					
5	5.38×10^{5}	0.1	1	<.005	<.5	0.0	1	0.0	1	33	0.01	<.5	9	<.005	<.5	<.005	1	<.005	<.5	<.005	<.5
10	8.29×10^{9}	3.1	10	0.05	<.5	0.2	10	0.4	3	63	0.01	<.5	9	0.01	<.5	0.6	10	0.1	1	0.01	<.5
15	1.46×10^{15}	1503.9	15	0.17	<.5	1.8	15	2.0	10	93	0.37	1	9	0.02	<.5	4.7	15	0.2	2	0.01	<.5
100	3.03×10^{105}	—	—	39.70	16	—	—	—	—	603	—	—	9	1.60	62	—	—	—	—	0.62	62
Round robin: $K = N + 1$, $\lvert\mathcal{E}\rvert = 6N$						$E[(p_1 \neq load)\,U\,(p_0 = send)]$										$EG(true)$ (find all cycles)					
5	3.60×10^{2}	0.1	1	<.005	<.5	0.0	1	0.2	1	19	<.005	<.5	6	<.005	<.5	0.0	1	0.1	1	<.005	<.5
10	2.30×10^{5}	68.2	11	0.01	<.5	0.2	11	85.0	11	39	0.01	<.5	11	0.01	<.5	0.3	11	78.5	13	<.005	<.5
15	1.10×10^{6}	4201.5	40	0.02	<.5	0.6	40	4922.7	40	59	0.03	<.5	16	0.01	<.5	1.2	40	4739.5	44	0.01	<.5
100	2.85×10^{32}	—	—	1.98	1	—	—	—	—	399	13.32	32	101	4.67	19	—	—	—	—	1.29	20
Flexible manuf. sys.: $K = 19$, $\lvert\mathcal{E}\rvert = 20$						$E[(M_1 > 0)\,U\,(P_1s = P_2s = P_3s = N)]$										$EG\neg(P_1s = P_2s = P_3s = N)$					
2	3.44×10^{3}	128.3	17	0.01	<.5	0.2	17	318.1	43	31	0.04	<.5	6	0.01	<.5	0.2	17	128.9	18	<.005	<.5
3	4.86×10^{4}	4107.5	127	0.02	<.5	1.0	127	—	—	46	0.16	<.5	8	0.02	<.5	1.0	127	—	—	0.01	<.5
25	8.54×10^{13}	—	—	17.98	<.5	—	—	—	—	376	—	—	52	1010.85	293	—	—	—	—	50.38	251

Table 1: Experimental results: SMART vs. NuSMV.

5 Conclusion and Future Work

We showed how, by exploiting the structure of a discrete-state model, one can recognize event locality, encode it using a boolean Kronecker matrix representation of the next-state function, and greatly improve the efficiency of symbolic CTL model checking, i.e., the computation of the sets of states satisfying an EX, EF, EU, or EG formula.

Furthermore, we showed that the *saturation* algorithm we initially proposed for state-space generation can be adapted to efficiently find the states satisfying an EU expression, by automatically classifying the *safety* of the model events. The resulting *EUsat* algorithm is at least as fast, and in many cases much faster than, our already improved EU computation. *EUsat* can also be used as the key procedure to compute the set of states satisfying an EG formula. However, this approach enumerates the SCCs in the model, thus it can be pathologically poor.

In the future, we will investigate how to further improve the EG computation, for example by exploring how saturation may be used to obtain the SCC hull without enumeration. We will also extend our work to Fair-CTL, since our event classification idea should remain applicable. Finally, since the asynchronous systems we target are often studied using explicit partial-order reduction techniques, we intend to perform a thorough comparison with tools such as SPIN.

References

[1] R. Alur et al. Partial-order reduction in symbolic state space exploration. In *Proc. CAV*, pages 340–351. Springer, 1997.

[2] R. Bloem, H. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In *Proc. FMCAD*, pages 37–54. Springer, 2000.

[3] R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Proc. DAC*, pages 29–34. ACM Press, 2000.

[4] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comp. Surv.*, 24(3):393–318, 1992.

[5] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, 2000.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, pages 428–439, 4–7 1990.

[7] G. Ciardo et al. SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual. Available at http://www.cs.wm.edu/∼ciardo/SMART/.

[8] G. Ciardo, G. Luettgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *Proc. ICATPN*, LNCS 1825, pages 103–122. Springer, June 2000.

[9] G. Ciardo, G. Luettgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342. Springer, Apr. 2001.

[10] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proc. TACAS*, LNCS 2619, pages 379–393. Springer, Apr. 2003.

[11] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proc. CAV*, LNCS 1633, pages 495–499. Springer, 1999.
[12] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer, 1981.
[13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
[14] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *Proc. Int. Conference on VLSI*, pages 49–58. IFIP Transactions, North-Holland, Aug. 1991.
[15] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
[16] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. ICCD*, pages 220–223. IEEE Comp. Soc. Press, Sept. 1990.
[17] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. ICATPN*, LNCS 1639, pages 6–25, June 1999.
[18] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. SIGMETRICS*, pages 147–153, May 1985.
[19] K. Ravi and F. Somenzi. Efficient fixpoint computation for invariant checking. In *Proc. ICCD*, pages 467–474. IEEE Comp. Soc. Press, Oct. 1999.
[20] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components. In *Proc. ICCAD*, pages 37–40. ACM Press, 1999.