

Decision-diagram-based Techniques for Bounded Reachability Checking of Asynchronous Systems.*

Andy Jinqing Yu¹, Gianfranco Ciardo¹, Gerald Lüttgen²

¹ Department of Computer Science and Engineering, University of California, Riverside, CA 92521, USA

e-mail: {jqyu, ciardo}@cs.ucr.edu

² Department of Computer Science, University of York, York YO10 5DD, U.K. e-mail: luetttgen@cs.york.ac.uk

The date of receipt and acceptance will be inserted by the editor

Abstract Bounded reachability analysis and bounded model checking are widely believed to perform poorly when using decision diagrams instead of SAT procedures. Recent research suggests this to be untrue with regards to synchronous systems and, in particular, digital circuits. This article shows that the belief is also a myth for asynchronous systems, such as models specified by Petri nets. We propose several *Bounded Saturation* approaches to compute bounded state spaces using decision diagrams. These approaches are based on the established Saturation algorithm, which benefits from a non-standard search strategy that is very different from breadth-first search, but employ different flavors of decision diagrams: Multi-valued Decision Diagrams, Edge-valued Decision Diagrams, and Algebraic Decision Diagrams. We apply our approaches to studying deadlock as a safety property. Our extensive benchmarking shows that our algorithms often, but not always, compare favorably against two SAT-based approaches that are advocated in the literature.

1 Introduction

Bounded model checking (BMC) is a well-established technique for reasoning about reactive systems [3]. Unlike conventional model checking based on explicit or symbolic representations of state spaces [17], bounded model checking takes a system, a bound B , and a safety property ϕ , unwinds the system's transition relation B

times, and derives a propositional formula that is satisfiable if and only if there exists a path through the system of length at most B that demonstrates the violation of ϕ . Due to the impressive technology advances in *SAT solving* (see, e.g., [30]), such satisfiability problems can often be decided efficiently.

BDDs vs. SAT. BMC is an incomplete verification technique unless the bound exceeds the state-space diameter, or unless it is combined with additional checks [27, 29, 34]. However, as faults involve relatively short counterexamples in practice, BMC has proved itself an efficient debugging aid and verification method: bounded model checkers are nowadays used to debug and verify *digital circuits* [16], *Petri nets* [21, 31], and *software* [24, 32]. Several studies have found such model checkers beneficial in industrial settings, especially when compared to symbolic model checkers using decision diagrams [18].

It is widely believed that SAT methods are key to the performance of bounded model checkers. Recent research by Cabodi et al. [6], however, counters this suggestion. Their work proposes enhancements to standard techniques based on *Binary Decision Diagrams* (BDDs), making BDD-based BMC competitive with SAT-based approaches. Their results were obtained in the context of debugging synchronous systems and digital circuits, for which BDDs are known to work well. It has remained an open question whether the aforementioned belief is also a myth with regards to asynchronous systems that are governed by interleaving semantics, such as distributed algorithms expressed in Petri nets.

Contribution. Our aim is to prove that decision diagrams are competitive with SAT solvers for the bounded model checking of *asynchronous* systems. To this end, we propose several new approaches for bounded reachability checking using decision diagrams based on *Saturation* [9], an established symbolic algorithm for generating the state spaces defined by asynchronous systems.

* Research supported by the NSF under grants CNS-0501747 and CNS-0501748 and by the EPSRC under grant GR/S86211/01. An extended abstract of this article appeared in the proceedings of the 13th Intl. Conf. on *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS), LNCS 4424, pp. 648–663, 2007. Springer.

By taking into account event locality and interleaving semantics and by using a different iteration strategy for computing fixpoints, Saturation is often orders of magnitude more efficient than advanced breadth-first search (BFS) algorithms implemented in popular model checkers [14]. In particular, Saturation’s search strategy is designed to compute “sub-fixpoints” on decision-diagram nodes while traversing the decision diagram in a bottom-up fashion, thereby exploring states at greater distances earlier than standard BFS-based approaches do.

The difficulty in adapting our Saturation algorithm to bounded reachability checking lies in its non-standard search strategy which is completely different from BFS. We present several solutions using Multi-valued Decision Diagrams (MDDs) [25], Edge-valued Decision Diagrams (EDDs, called EV⁺MDDs in [12]), and Algebraic Decision Diagrams (ADDs [2], also called MTBDDs [15]).

In the EDD- and ADD-based approaches, we store not only the reachable states but also the distance of each state from the initial state(s). EDDs extend EV-BDDs [26] just as MDDs extend BDDs. Each state stored in such a decision diagram corresponds to a path from the diagram’s root to its terminal node, whereas the distance of the state from the initial state(s) is the sum of the weights of the edges along that path. The resulting EDD-based *Bounded Saturation* algorithm comes in two variants. The first one computes all reachable states at distance no more than a user-provided bound B . The second one finds additional states at distance greater than B but at most $K \cdot B$, where K is the number of the levels in the EDD. Just as ordinary BFS, both can find minimal-length counterexamples. However, the second variant is usually more efficient in terms of runtime and memory, even though it discovers more states. Such behavior, while counterintuitive at first, is not uncommon for decision diagrams.

The ADD-based Bounded Saturation approach stores the distance explicitly in terminal nodes and bounds the forward traversal when the distance stored reaches B . It therefore finds exactly all the states at distance up to the bound B . We also consider MDD-based Bounded Saturation, presenting approaches that remove the need to store distance information within decision diagrams. We employ BFS-style iterations in each Saturation step and bound forward traversal by limiting the number of iterations. The MDD-based algorithm also comes in two variants; they differ in the way the symbolic forward traversal is bounded.

Experiments and results. We evaluate our Bounded Saturation algorithms against two SAT-based algorithms for bounded reachability checking which have been devised, respectively, by Heljanko [21] and by Ogata, Tsuchiya, and Kikuno [31]. Both are aimed at finding deadlocks in asynchronous systems specified by Petri nets. We implemented our algorithms in the Petri-net verification tool SMART [8], and ran them on the suite of examples used in

both [21] and [31], which was first proposed by Corbett in [19], as well as on models taken from the SMART release. The static variable ordering used in our algorithms was computed via a heuristic [33].

Our experiments show that Bounded Saturation performs better or on par with competing SAT-based algorithms, and is less efficient in only few cases. Thus, it is a myth that decision diagrams are uncompetitive with respect to SAT solvers for BMC. Just as the roles of bounded and unbounded model checking are complementary, so is the use of SAT solvers and decision diagrams.

Organization. The next section provides background on decision-diagram-based reachability analysis, including the different flavors of decision diagrams we employ, and on our Saturation algorithm. It also introduces a running example that is used throughout this article to illustrate different concepts and algorithms. Sec. 3 presents our various Saturation-based approaches to bounded reachability checking, which are then carefully analyzed and compared to established SAT-based approaches via extensive benchmarking in Sec. 4. Finally, Sec. 5 discusses related work, while our conclusions and suggestions for future work are presented in Sec. 6.

2 Background

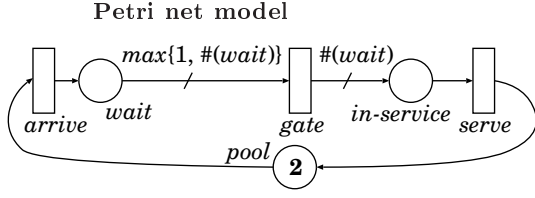
We consider a discrete-state model $\mathcal{M} = (\widehat{\mathcal{S}}, \mathcal{S}^{init}, \mathcal{R})$, where $\widehat{\mathcal{S}}$ is a (finite) set of states, $\mathcal{S}^{init} \subseteq \widehat{\mathcal{S}}$ are the initial states, and $\mathcal{R} \subseteq \widehat{\mathcal{S}} \times \widehat{\mathcal{S}}$ is a transition relation. We assume the *global* model state to be a tuple (x_K, \dots, x_1) of K *local state* variables, where $x_l \in \mathcal{S}_l = \{0, 1, \dots, n_l - 1\}$, for $K \geq l \geq 1$ and $n_l > 0$, is the l^{th} *local state* variable. Thus, $\widehat{\mathcal{S}} = \mathcal{S}_K \times \dots \times \mathcal{S}_1$, and we write $\mathcal{R}(\mathbf{i}[K], \dots, \mathbf{i}[1], \mathbf{j}[K], \dots, \mathbf{j}[1])$ or simply $\mathcal{R}(\mathbf{i}, \mathbf{j})$ if the model can move from *current state* \mathbf{i} to *next state* \mathbf{j} in one step.

Most symbolic approaches encode x_l in b_l boolean variables, where b_l is either n_l or $\lceil \log n_l \rceil$ (called *one-hot* and *binary* encoding, respectively), and a set of states via a BDD with $\sum_{K \geq l \geq 1} b_l$ levels. *Ordered Multi-valued Decision Diagrams* (MDDs) [25] instead map x_l to level l , whose nodes have n_l outgoing edges. MDDs can be implemented directly, as is done in our tool SMART [8], or as an interface to BDDs [20].

The computation of a model’s reachable state space consists of building the smallest set of states $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ satisfying $\mathcal{S} \supseteq \mathcal{S}^{init}$ and $\mathcal{S} \supseteq \text{Image}(\mathcal{S}, \mathcal{R})$, where the *image computation* function

$$\text{Image}(\mathcal{X}, \mathcal{R}) = \{\mathbf{j} : \exists \mathbf{i} \in \mathcal{X}, \mathcal{R}(\mathbf{i}, \mathbf{j})\}$$

describes the successors to the set \mathcal{X} of states. In BMC, only a portion of this state space must be examined, namely the set of states within some given distance bound B from \mathcal{S}^{init} .



Guarded command language model

Initial state: $p = 2 \wedge w = 0 \wedge i = 0$;
 $\mathcal{D}_a : p \geq 1 \rightarrow \{p' = p - 1 \wedge w' = w + 1 \wedge i' = i\}$;
 $\mathcal{D}_s : i \geq 1 \rightarrow \{p' = p + 1 \wedge w' = w \wedge i' = i - 1\}$;
 $\mathcal{D}_g : w \geq 1 \rightarrow \{p' = p \wedge w' = 0 \wedge i' = i + w\}$;

Figure 1: A limited-arrival gated-service model is described using a Petri net with marking-dependent arc cardinalities (left) and guarded command language (right), respectively.

2.1 Symbolic techniques for asynchronous models

A BFS-based approach, as used for example by NuSMV [14], computes the bounded state space with a simple image computation iteration. Set $\mathcal{X}^{[0]}$ is initialized to \mathcal{S}^{init} and, after d iterations, set $\mathcal{X}^{[d]}$ contains the states at distance up to d from \mathcal{S}^{init} . With MDDs, $\mathcal{X}^{[d]}$ is encoded as a K -level MDD and \mathcal{R} as a $2K$ -level MDD whose current and next state variables are normally interleaved for efficiency. The transition relation is often conjunctively partitioned into a set of *conjuncts* or disjunctively into a set of *disjuncts* [5], and is stored as a set of MDDs with shared nodes, instead of a single monolithic MDD. Heuristically, such partitions are known to be effective for synchronous and asynchronous systems, respectively.

2.2 Disjunctive-conjunctive partitioning and chaining

Our work focuses on the important class of systems exhibiting *globally-asynchronous locally-synchronous* behavior, and assumes that a given high-level model specifies a set \mathcal{E} of asynchronous events, where each event $\alpha \in \mathcal{E}$ is further specified as a set of small synchronous components \mathcal{D}_α . We then write the transition relation as $\mathcal{R} \equiv \bigvee_{\alpha \in \mathcal{E}} \mathcal{D}_\alpha$, and conjunctively partition each disjunct \mathcal{D}_α into conjuncts $\mathcal{C}_{\alpha,r}$ that represent the synchronous components of α , thus expressing \mathcal{R} as

$$\mathcal{R} = \bigvee_{\alpha \in \mathcal{E}} \mathcal{D}_\alpha = \bigvee_{\alpha \in \mathcal{E}} \left(\bigwedge_r \mathcal{C}_{\alpha,r} \right).$$

These high-level models may be specified using Petri nets or a guarded command language. Such a language consists of a set of commands of the form

$$\text{guard} \rightarrow \text{assignment}_1 \parallel \text{assignment}_2 \parallel \dots \parallel \text{assignment}_m,$$

whose meaning is that m parallel atomic assignments are executed concurrently, whenever the boolean predicate *guard* evaluates to *true*. The assignments are asynchronous events and, for each command, the corresponding parallel assignments are its synchronous components. Similarly, for a Petri net, the transitions are the asynchronous events, and the firing of a transition synchronously updates all input and output places connected to it. We use extended Petri nets as the input formalism in SMART, which augment ordinary nets by *inhibitor arcs* and *marking-dependent arc cardinalities* [7, 37].

2.3 Running example

Fig. 1 shows a Petri net and its equivalent guarded command language expression, which models a gated-service queue with a limited pool of customers. New arrivals wait at the gate until it is opened, and then all waiting customers enter the service queue. Customers return to the pool after service. Each state of the model corresponds to a possible value of the integer variable vector (p, w, i) , where p stands for *pool* (the number of customers in the pool), w for *wait* (the number of customers waiting at the gate), and i for *in-service* (the number of customers in the service queue). Assuming a pool of two customers, the model has an initial state of $(2, 0, 0)$, one immediate successor state $(1, 1, 0)$, and six reachable states: $\mathcal{S} = \{(2, 0, 0), (1, 1, 0), (0, 2, 0), (1, 0, 1), (0, 0, 2), (0, 1, 1)\}$.

2.4 Event locality

In asynchronous models, the execution of each event usually modifies or depends on just a small subset of state variables. In the running example, event *gate* \mathcal{D}_g depends only on variable w , and modifies only variables w and i . Given an event α , we define the sets of variables $\mathcal{V}_M(\alpha)$ and $\mathcal{V}_D(\alpha)$ that can be modified by α or can disable α , respectively:

$$\begin{aligned} \mathcal{V}_M(\alpha) &= \{x_l : \exists \mathbf{i}, \mathbf{j} \in \widehat{\mathcal{S}}, \mathcal{D}_\alpha(\mathbf{i}, \mathbf{j}) \wedge \mathbf{i}[l] \neq \mathbf{j}[l]\}; \\ \mathcal{V}_D(\alpha) &= \{x_l : \exists \mathbf{i}, \mathbf{i}' \in \widehat{\mathcal{S}}, \forall k \neq l, \mathbf{i}[k] = \mathbf{i}'[k] \wedge \\ &\quad \exists \mathbf{j} \in \widehat{\mathcal{S}}, \mathcal{D}_\alpha(\mathbf{i}, \mathbf{j}) \wedge \nexists \mathbf{j}' \in \widehat{\mathcal{S}}, \mathcal{D}_\alpha(\mathbf{i}', \mathbf{j}')\}. \end{aligned}$$

Further defining

$$\begin{aligned} \text{Top}(\alpha) &= \max\{l : x_l \in \mathcal{V}_M(\alpha) \cup \mathcal{V}_D(\alpha)\} \text{ and} \\ \text{Bot}(\alpha) &= \min\{l : x_l \in \mathcal{V}_M(\alpha) \cup \mathcal{V}_D(\alpha)\}, \end{aligned}$$

we can then partition the event set \mathcal{E} according to the value of *Top* into the subsets $\mathcal{E}_l = \{\alpha : \text{Top}(\alpha) = l\}$, for $K \geq l \geq 1$. In [10] we observed that a chaining order [35], where these subsets are applied to an MDD in bottom-up fashion, results in good speed-ups with respect to a strict BFS symbolic state-space generation. The bounded version of this chaining heuristic is shown in Fig. 3 and discussed in Sec. 3.

By exploiting *event locality*, we can store \mathcal{D}_α in an MDD over just the current and next state variables with

index k , for $Top(\alpha) \geq k \geq Bot(\alpha)$; variables outside this range undergo an *identity transformation* when computing the result of firing α , i.e., remain unchanged.

2.5 Saturation-based fixpoint computation

The *Saturation* algorithm to compute the reachable state space of an asynchronous system was originally proposed in [9] for models in *Kronecker-product* form; it has since been extended to general models [13] and applied to shortest path computations and CTL model checking [12]. Saturation has been shown to reduce runtime and memory requirements by several orders of magnitude with respect to BFS-based algorithms, when applied to asynchronous systems [9,13].

Saturation may best be understood as a dynamic programming approach to the symbolic reachability problem of asynchronous systems. It recursively computes “sub-fixpoints” on decision-diagram nodes in a bottom-up fashion, i.e., from level 1 of a decision diagram up to the root node at level K , by firing events α with $Top(\alpha) = l$ on nodes at level l . A node at level l is called *saturated*, once the “sub-fixpoint” on it is reached, i.e., no more firings of events α with $Top(\alpha) = l$ leads to the discovery of new (sub-)states. The entire reachable state space is explored when the root node is saturated. Hence, Saturation is unique in that it does not perform a monolithic fixpoint computation over a global decision diagram, as standard breadth-first iteration strategies do. Instead, it divides the monolithic fixpoint computation into light-weight computations on each decision-diagram node. This exploits event locality and respects the underlying semantic concept of interleaving. We refer the reader to [9,10,11] for details.

To adapt Saturation to bounded reachability checking, it is important to note that, in symbolic algorithms, transition relation and state set can be described by different DD types; e.g., one may use MDDs to represent the transition relation and ADDs to represent state sets. Moreover, in order to bound the state space exploration, one may encode not just the reachable states but also their distance from \mathcal{S}^{init} within decision diagrams. This can be achieved by using either ADDs or EDDs, where EDDs can be exponentially more compact than ADDs. In Sec. 3, also another way of accomplishing Bounded Saturation is proposed, which eliminates the need to store distances and uses plain MDDs instead. This may reduce the sizes of decision diagrams substantially when compared to ADD or EDD encodings. Our formal algorithms of Saturation for bounded state-space exploration using MDDs, EDDs, and ADDs are described in Sec. 3. In the following, we first define ADDs and EDDs.

2.6 Algebraic Decision Diagrams

ADDs [2] are a well-known variant of BDDs that can represent non-boolean functions by allowing an arbitrary

finite set of terminal nodes instead of just the two terminal nodes corresponding to the boolean values *true* and *false*. Here, ADDs are used to encode bounded state spaces as well as the state distances from the set of initial states, and are thus defined over the semi-ring $(\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$, where $\mathbb{N} \cup \{\infty\}$ is the underlying carrier; \min and $+$ are the two binary arithmetic operators minimum and plus, respectively; infinity ∞ is the identity for operator \min and the annihilator for operator $+$; and 0 is the identity for $+$. We extend the original definition of ADDs presented in [2] and allow each variable x_l , for $l \in \{K, \dots, 1\}$, to take $n_l \geq 2$ different values.

Definition 1 (ADD [2]). An ADD on the domain $\widehat{\mathcal{S}} = \mathcal{S}_K \times \dots \times \mathcal{S}_1$ is a directed, acyclic multi-graph, where:

- Each node p belongs to a *level* in $\{K, \dots, 1, 0\}$, denoted by $p.lvl$.
- There is a single root node r^* at level K .
- Level 0 contains a finite set of *terminal* nodes, which are all distinct and where one terminal node is labeled with ∞ , and the other terminal nodes are labeled with natural numbers.
- A node p at level $l > 0$ has n_l outgoing edges, labeled from 0 to $n_l - 1$. The edge labeled by i_l points to a node q , which is either a node at level $p.lvl - 1$ or the terminal node ∞ . We write $p[i_l] = q$, if the i^{th} edge of p points to node q .
- There are no *duplicate* nodes, i.e., if $\forall 0 \leq i < n_l. p[i] = q[i]$, then $p = q$.

The function $f_p : \mathcal{S}_l \times \dots \times \mathcal{S}_1 \rightarrow \mathbb{N} \cup \{\infty\}$ encoded by ADD node p , with $p.lvl = l > 0$, is $f_p(i_l, \dots, i_1) = f_{p[i_l]}(i_{l-1}, \dots, i_1)$. For terminal nodes, we let $f_{i_0} = i_0$, for $i_0 \in \mathbb{N}$, and $f_\infty = \infty$. The function encoded by the entire ADD is f_{r^*} . □

Figs. 2(a) and (b) show two ADDs storing a total function f_1 and a partial function f_2 , respectively. Here, “partial” means that some of its values are ∞ ; for better readability, we omit the terminal node ∞ and the edges and nodes that lead to it from the graphical representation.

2.7 Edge-valued Decision Diagrams

EDDs [12] are an alternative to ADDs for encoding functions of the form $\mathcal{S}_K \times \dots \times \mathcal{S}_1 \rightarrow \mathbb{N} \cup \{\infty\}$. They store function values in their edges instead of the terminal nodes, and often result in a more compact encoding than ADDs in our application.

Definition 2 (EDD [12]). An EDD on the domain $\widehat{\mathcal{S}} = \mathcal{S}_K \times \dots \times \mathcal{S}_1$ is a directed, acyclic graph with labeled and weighted edges, where:

- Each node p belongs to a *level* in $\{K, \dots, 1, 0\}$, denoted $p.lvl$.

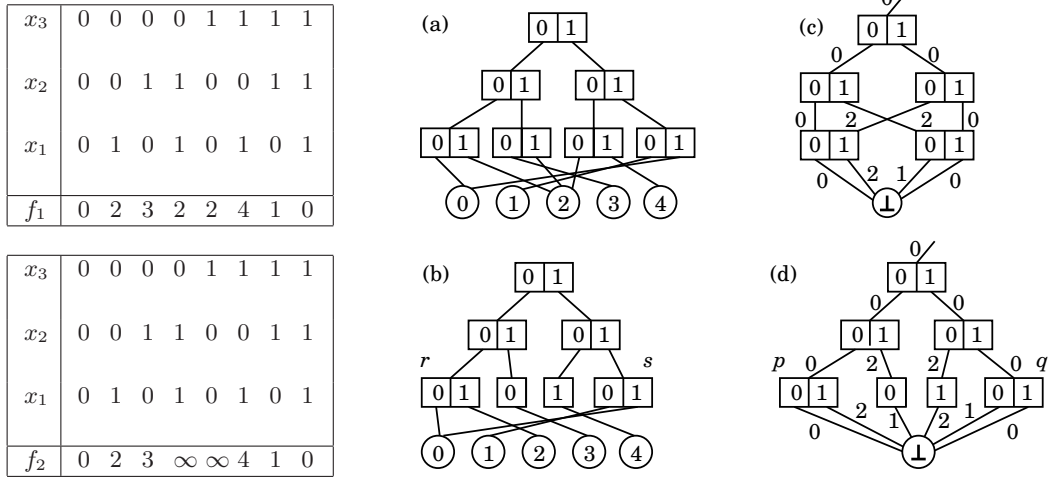


Figure 2: The ADD (a) or EDD (c) to store total function f_1 , and the ADD (a) or EDD (c) to store partial function f_2 .

- There is a single root node r^* at level K , with an incoming “dangling” edge having weight $\rho^* \in \mathbb{N}$. We write such an edge as $\langle \rho^*, r^* \rangle$.
- Level 0 contains a single terminal node, \perp .
- A non-terminal node p at level $l > 0$ has $n_l \geq 2$ outgoing edges, labeled 0 to $n_l - 1$. We write $p[i] = \langle v, q \rangle$ if the i^{th} edge has weight $v \in \mathbb{N} \cup \{\infty\}$ and points to node q . In addition, we write $p[i].val = v$ and $p[i].node = q$.
- If $p[i].val = \infty$, then $p[i].node = \perp$; otherwise, $p[i].node$ is at level $p.lvl - 1$.
- Each non-terminal node has at least one outgoing edge labeled 0.
- There are no *duplicate* nodes, i.e., if $\forall 0 \leq i < n_l$. $p[i].node = q[i].node$ and $p[i].val = q[i].val$, then $p = q$.

The function $f_{\langle v, p \rangle} : \mathcal{S}_l \times \dots \times \mathcal{S}_1 \rightarrow \mathbb{N} \cup \{\infty\}$ encoded by edge $\langle v, p \rangle$, with $p.lvl = l > 0$, is $f_{\langle v, p \rangle}(i_l, \dots, i_1) = v + f_{\langle p[i_l].val, p[i_l].node \rangle}(i_{l-1}, \dots, i_1)$, where $f_{\langle x, \perp \rangle} = x$. Thus, the function encoded by the entire EDD is $f_{\langle \rho^*, r^* \rangle}$, where ρ^* is the minimum value assumed by this function. \square

As defined, EDDs can canonically represent *any* function of the form $\widehat{\mathcal{S}} \rightarrow \mathbb{N} \cup \{\infty\}$, except the constant ∞ , for which we use an EDD with $r^* = \perp$ (at level 0, not K), and $\rho^* = \infty$. Figs. 2 (c) and (d) show two EDDs storing the total function f_1 and the partial function f_2 , respectively. Whenever partial function f_2 has value ∞ , we omit the EDD edges from its graphical representation. We point out that EDDs allow for the efficient implementation of many standard operations on the functions they encode, including the pointwise *minimum* of two functions [12] which is needed in our bounded reachability algorithms.

3 Bounded reachability checking

Given a model \mathcal{M} and a state property ϕ , a generic breadth-first bounded reachability checking algorithm starts with some initial guess for the bound B , computes the set \mathcal{S}^B of states within distance B of the initial states \mathcal{S}^{init} , and, if any state in \mathcal{S}^B violates ϕ , returns *Error*. If no such state exists, B is increased and these steps are repeated until some given bound is reached or until the entire state space has been explored. In the latter case, ϕ is declared valid.

Our goal is to develop bounded state-space exploration algorithms that are guaranteed to terminate even when the state space \mathcal{S} is infinite, as long as any state can reach only a finite number of states within one step. This last condition is guaranteed to hold if, as we assume, the following is true:

1. The set \mathcal{E} of model events is finite; and
2. The effect of firing an event α has only a finite number of possible outcomes, i.e., $|Image(\mathbf{i}, \mathcal{D}_\alpha)| < \infty$ for any $\mathbf{i} \in \widehat{\mathcal{S}}$.

Let the distance of a global state $\mathbf{j} \in \widehat{\mathcal{S}}$ from a global state $\mathbf{i} \in \widehat{\mathcal{S}}$, or from a set $\mathcal{I} \subseteq \widehat{\mathcal{S}}$ of states, be defined as:

$$\delta(\mathbf{i}, \mathbf{j}) = \min\{d : \mathbf{j} \in Image^d(\{\mathbf{i}\}, \mathcal{R})\},$$

$$\delta(\mathcal{I}, \mathbf{j}) = \min\{\delta(\mathbf{i}, \mathbf{j}) : \mathbf{i} \in \mathcal{I}\}.$$

Then, we seek algorithms that, given a discrete-state model $(\widehat{\mathcal{S}}, \mathcal{S}^{init}, \mathcal{R})$ and a bound B , build a set \mathcal{S}^B of states satisfying:

1. $\forall \mathbf{j} \in \widehat{\mathcal{S}}. \delta(\mathcal{S}^{init}, \mathbf{j}) \leq B \Rightarrow \mathbf{j} \in \mathcal{S}^B$
2. $\exists B' > B. \forall \mathbf{j} \in \widehat{\mathcal{S}}. \delta(\mathcal{S}^{init}, \mathbf{j}) \geq B' \Rightarrow \mathbf{j} \notin \mathcal{S}^B$.

The first condition guarantees that all states within distance B are in \mathcal{S}^B , while the second condition gives an upper bound B' for the states' distance in set \mathcal{S}^B . In

```

MDD BoundedBfsChain()
1  $\mathcal{S} \leftarrow \mathcal{S}^{init}$ ;
2 for  $d = 1$  to  $B$  do
3   for  $l = 1$  to  $K$  do
4     foreach  $\alpha \in \mathcal{E}_l$  do
5        $\mathcal{S} \leftarrow \text{Union}(\mathcal{S}, \text{Image}(\mathcal{S}, \mathcal{D}_\alpha))$ ;
6 return  $\mathcal{S}$ ;

```

Figure 3: Symbolic bounded BFS state-space generation with chaining.

our proposed bounded approaches, the upper bound B' varies due to the trade-off between accuracy and efficiency. Some methods, e.g., the ADD-based method of Fig. 6, have exact bounds (i.e., $B' = B$), while other methods have approximate bounds. For example, our MDD-based globally-bounded method of Fig. 8 satisfies $B' = B^K$, where K is the number of MDD levels.

3.1 MDDs with BFS-style chaining

Before presenting our main contribution of Bounded Saturation algorithms, we first show how the standard BFS-search algorithm can be improved when dealing with MDD-encoded state spaces of event-based asynchronous systems, using ideas from both *event locality* and *forward chaining* [35]. The improved BFS algorithm serves as one of the reference algorithms in our experimental studies of Sec. 4 and is shown in Fig. 3.

Exploiting event locality for an event α , we can ignore MDD levels above $Top(\alpha)$ and modify *in-place* MDD nodes at level $Top(\alpha)$. Indeed, the call to *Image* in Fig. 3 does not even access nodes below $Bot(\alpha)$, only *Union* does. This has been shown experimentally to significantly reduce the peak number of MDD nodes during state-space generation [10].

Chaining [35] compounds the effect of multiple events within a single iteration. For example, if (i) the set of reachable states known at iteration B is \mathcal{X}^B , (ii) $\mathbf{j} \notin \mathcal{X}^B$ can be reached from $\mathbf{i} \in \mathcal{X}^B$ by firing the sequence of distinct events (α, β, γ) , and (iii) one happens to explore events in that exact order, then \mathbf{j} will be included in \mathcal{X}^{B+1} . Thus, $\mathcal{X}^B \supseteq \mathcal{S}^B$ since some states in $\mathcal{S}^{|\mathcal{E}| \cdot B} \setminus \mathcal{S}^B$ might be present in \mathcal{X}^B . Reducing the number of iterations does not in principle imply greater efficiency, as the MDD for \mathcal{X}^B could be much larger than the one for \mathcal{S}^B ; however, it has been shown experimentally that chaining often reduces both time and memory requirements [35].

It is well known that the chosen variable order is essential in decision-diagram-based algorithms [4]. Furthermore, in our setting, the variable order affects the values *Top* and *Bot*, as well as the order of firing events. Therefore, we employ the heuristic introduced in [33] to automatically generate good static variable orders; this heuristic aims to minimize the sum of the values *Top* over all events.

3.2 Bounded Saturation using EDDs

In several studies, Saturation has been shown superior to BFS-style iterations when symbolically computing the state space (as a least fixpoint) of asynchronous models [10, 11]. The challenge in adapting Saturation to bounded model checking arises from the need to bound the symbolic traversal in its nested fixpoint computations. This section explores algorithms that use EDDs to encode both the bounded state space and the distance information within the same symbolic data structure. Thus, we bound the traversal during the EDD symbolic operations by using the distance information, instead of limiting the number of outermost iterations performed in a traditional BFS-style approach.

Fig. 4 shows two EDD approaches that differ in how they bound the symbolic traversal. They are obtained by replacing the *Truncate* call (line 5 in procedure *BoundedSaturate* and line 7 in procedure *BoundedEDDImage*) with either *TruncateExact* or *TruncateApprox*. The former computes the exact bounded state space \mathcal{S}^B ; the latter computes a superset of \mathcal{S}^B that may contain reachable states with distance at most $K \cdot B$, where K is the number of state variables, i.e., EDD levels. Recall that transition relations are stored using MDDs, with $\mathbf{0}$ and $\mathbf{1}$ denoting an MDD's terminal nodes.

Both approaches start from an EDD where states in \mathcal{S}^{init} have distance 0 and states in $\widehat{\mathcal{S}} \setminus \mathcal{S}^{init}$ have distance ∞ (line 1 in *BoundedEDDSaturation*); thus, $\rho^* = 0$. Then, procedure *BoundedEDDSaturate* is called on all EDD nodes, starting from those at level 1, to compute the bounded state space. Each EDD node p at level l represents a set of (sub-)states and distance information consisting of variables at level l and below. When calling procedure *BoundedEDDSaturate* on an EDD node p at level l , a least fixpoint encoding the (sub-)state space and distance with respect to the set \mathcal{E}_l of events with top level l is computed. During the computation of *BoundedEDDSaturate* on node p at level l , each event in \mathcal{E}_l is exhaustively fired to perform bounded forward traversal, until no new reachable (sub-)states are found.

BoundedEDDImage performs a bounded forward traversal by first computing the forward image, followed by either an exact truncation to prune all (sub-)states exceeding bound B (procedure *TruncateExact*), or a faster but approximate truncation to prune only (sub-)states whose edge value in the current EDD node exceeds B

<pre> void <i>BoundedEDDSaturation</i>() 1 $r^* \leftarrow$ root of EDD encoding $f(i) = \begin{cases} 0 & \text{if } i \in \mathcal{S}^{init} \\ \infty & \text{otherwise} \end{cases}$; 2 for $k = 1$ to K do 3 foreach node p at level k do 4 <i>BoundedEDDSaturate</i>(p); </pre>	<pre> edge <i>Minimum</i>(edge $\langle v,p \rangle$, edge $\langle w,q \rangle$) 1 if $v = \infty$ then return $\langle w,q \rangle$; 2 if $w = \infty$ then return $\langle v,p \rangle$; 3 $k \leftarrow p.lvl$; \bulletgiven our quasi-reduced form, $q.lvl = k$ 4 if $k = 0$ then \bulletreached terminal node 5 return $\langle \min\{v,w\}, \perp \rangle$; 6 $s \leftarrow$ <i>NewNode</i>(k); \bulletempty node at level k 7 $\gamma \leftarrow \min\{v,w\}$; 8 foreach $i \in \mathcal{S}_k$ do 9 $x \leftarrow v - \gamma + p[i].val$; 10 $y \leftarrow w - \gamma + q[i].val$; 11 $s[i] \leftarrow$ <i>Minimum</i>($\langle x,p[i].node \rangle$, $\langle y,q[i].node \rangle$); 12 return $\langle \gamma,s \rangle$; </pre>
<pre> node <i>BoundedEDDSaturate</i>(node p) 1 $l \leftarrow p.lvl$; 2 repeat 3 choose $\alpha \in \mathcal{E}_l$, $i, j \in \mathcal{S}_l$ s.t. $p[i].val < B$; 4 $\langle v,q \rangle \leftarrow$ <i>BoundedEDDImage</i>($p[i], \mathcal{D}_\alpha[i][j]$); 5 $\langle w,s \rangle \leftarrow$ <i>Truncate</i>($v+1, q$); \bulletexact or approximate 6 $p[j] \leftarrow$ <i>Minimum</i>($p[j], \langle w,s \rangle$); 7 until p does not change; 8 return p; </pre>	<pre> edge <i>Normalize</i>(node p) 1 $v \leftarrow \min\{p[i].val : i \in \mathcal{S}_{p.lvl}\}$; 2 foreach $i \in \mathcal{S}_{p.lvl}$ do 3 $p[i].val \leftarrow p[i].val - v$; 4 return $\langle v,p \rangle$; </pre>
<pre> edge <i>BoundedEDDImage</i>(edge $\langle v,q \rangle$, MDD f) 1 if $f = \mathbf{0}$ then return $\langle \infty, \perp \rangle$; 2 if $f = \mathbf{1}$ or $q = \perp$ then return $\langle v,q \rangle$; 3 $k \leftarrow q.lvl$; \bulletgiven our quasi-reduced form, $f.lvl = k$ 4 $s \leftarrow$ <i>NewNode</i>(k); \bulletedges initialized to $\langle \infty, \perp \rangle$ 5 foreach $i \in \mathcal{S}_k, j \in \mathcal{S}_k$ s.t. $q[i].val \leq B$ do 6 $\langle v,u \rangle \leftarrow$ <i>BoundedEDDImage</i>($q[i], f[i][j]$); 7 $\langle w,o \rangle \leftarrow$ <i>Truncate</i>($\langle v,u \rangle$); \bulletexact or approximate 8 $s[j] \leftarrow$ <i>Minimum</i>($s[j], \langle w,o \rangle$); 9 $s \leftarrow$ <i>BoundedEDDSaturate</i>(s); 10 $\langle \gamma,s \rangle \leftarrow$ <i>Normalize</i>(s); 11 return $\langle \gamma+v,s \rangle$; </pre>	<pre> edge <i>TruncateExact</i>(edge $\langle v,p \rangle$) 1 if $v > bound$ then return $\langle \infty, \perp \rangle$; 2 foreach $i \in \mathcal{S}_{p.lvl}$ do 3 $p[i] \leftarrow$ <i>TruncateExact</i>($\langle v+p[i].val, p[i].node \rangle$); 4 return $\langle v,p \rangle$; </pre>
	<pre> edge <i>TruncateApprox</i>(edge $\langle v,p \rangle$) 1 if $v > bound$ then return $\langle \infty, \perp \rangle$; 2 else return $\langle v,p \rangle$; </pre>

Figure 4: Bounded Saturation using EDDs.

(procedure *TruncateApprox*). Procedures *BoundedEDDSaturate* and *BoundedEDDImage* are mutually recursive, as *BoundedEDDImage* performs a bounded forward traversal of the reachable state space, while all the created nodes in the new image are saturated by *BoundedEDDSaturate* (line 9 in procedure *BoundedEDDImage*). Procedure *Minimum* computes the pointwise minimum of the functions encoded by its two argument EDDs. Finally, procedure *Normalize* takes a node p , ensures that it has at least one outgoing edge with value 0, and returns the excess in the edge value v .

We now examine the manipulation of the edge values in more detail. When an event α is fired, the distance of the image states is the distance of the corresponding “from” states incremented by 1. *BoundedEDDSaturate* fires α by calling *BoundedEDDImage* (line 4), which returns the root of the image, so that the “dangling” edge value must be incremented by 1 in order to account for the firing of α (line 6). Procedure *BoundedEDDImage* performs the symbolic image computation of the same event α fired by *BoundedEDDSaturate*, and the distance of the new image is incremented by the distance of the “from” states at the return statement (line 11). The distance of the image states can be greater than the distance of their “from” states by more than one, due to saturation of the image states. Observe that *BoundedEDDSaturate* uses the test $p[i].val < B$ (line 3), but *BoundedEDDImage* uses instead the test $q[i].val \leq B$, since the

increment of the edge value by 1 is performed in the former, but not in the latter.

Compared to BFS-style MDD approaches, our two new EDD approaches use Saturation, i.e., a more advanced iteration order, but at the cost of a more expensive symbolic data structure, i.e., EDDs. The experimental results of Sec. 4 show that this trade-off is effective in both time and memory, as the new algorithms often outperform BFS in our benchmarks.

3.3 EDD approach on our running example

Fig. 5 illustrates the execution of Bounded Saturation using *TruncateApprox* as the truncation procedure, on the running example of Fig. 1 with bound $B = 1$. Snapshot (a) shows the $2K$ -level MDDs for the disjunctively partitioned transition relation. \mathcal{D}_a and \mathcal{D}_g have identity transformations for variables i and p , respectively; thus, the corresponding levels in the decision diagram are skipped to exploit event locality. Snapshots (b)–(f) show the evolution of the bounded state space encoded by the EDD, from the initial state to the final bounded state space, listing the key procedure calls. We denote the nodes of the EDD encoding the state space with capital letters (A to E), highlight two specific MDD nodes in the transition relation encoding by f and h , and color a node black once it is saturated. The algorithm starts by saturating nodes A and B , which are saturated im-

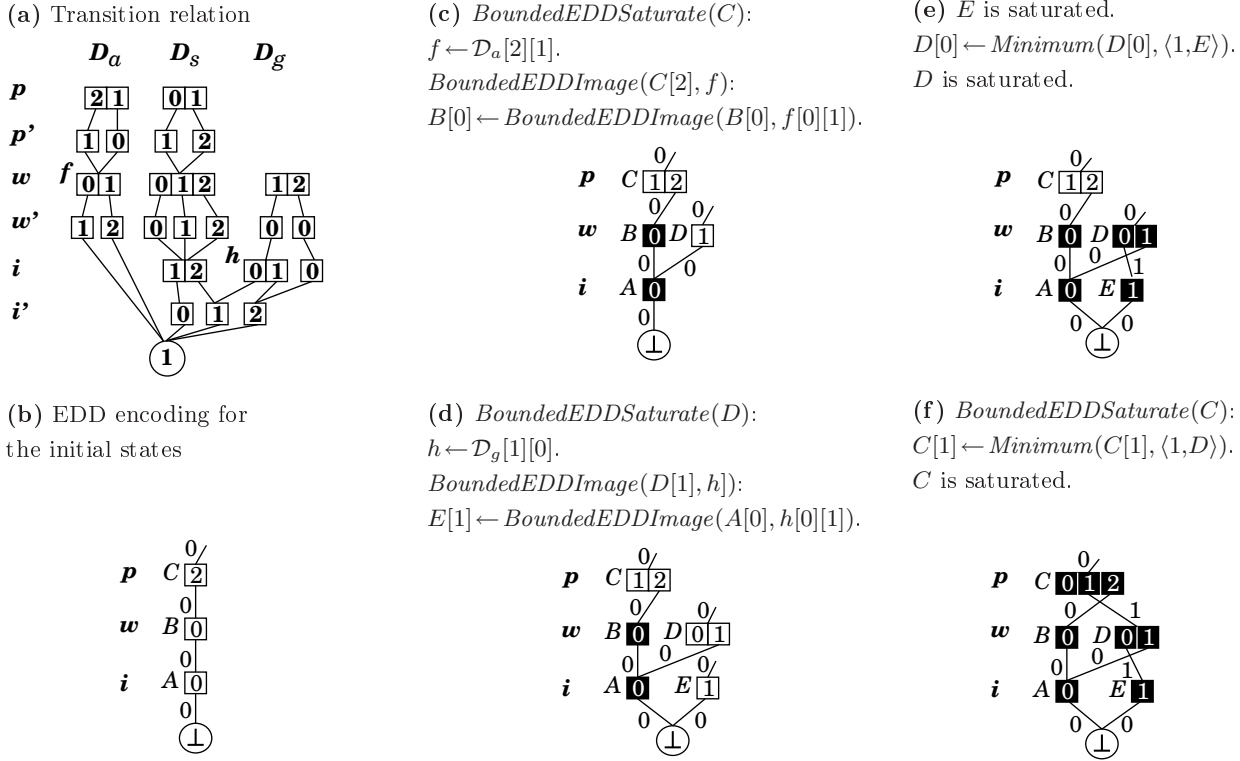


Figure 5: Bounded Saturation using EDDs, when applied to our running example. Snapshot (a) shows the partitioned transition relation, while snapshots (b)–(f) show the evolution of the bounded state space encoded by the EDD.

mediately since no events are enabled in them (Snapshot (c)). Nodes E , D , and C are saturated in that order. The procedure stops when root node C becomes saturated. Not all procedure calls are shown; for example, procedure $BoundedEDDImage(C[1], \mathcal{D}_s[1][2])$ is called in Snapshot (f) before node C becomes saturated, but does not generate any new node.

3.4 Bounded Saturation using ADDs

In this section, we propose a Bounded Saturation algorithm that uses ADDs to store both the state space and the distances. For a bound B , the ADD has $B + 2$ terminal nodes corresponding to the distances of interest, $\{0, 1, \dots, B, \infty\}$, where ∞ is used to denote any state distance greater than B .

The Bounded Saturation algorithm using ADDs is shown in Fig. 6, where the standard ADD procedure $Minimum$ computes the pointwise minimum of the functions encoded by its two argument ADDs. Similar to the EDD-based approaches, the ADD approach starts from an ADD where states in \mathcal{S}^{init} have distance 0 and states in $\widehat{\mathcal{S}} \setminus \mathcal{S}^{init}$ have distance ∞ (line 1 in $BoundedADDSSaturation$). Then procedure $BoundedADDSSaturate$ is called on all ADD nodes, starting from those at level 1. Each ADD node p at level l encodes a set of (sub-)states and distance information. Calling $BoundedADDSSaturate$ on

a node p at level l computes a least fixpoint encoding the (sub-)state space and distance with respect to event set \mathcal{E}_l , where each event in \mathcal{E}_l is exhaustively fired to perform a bounded forward traversal.

Unlike EDD-based approaches, however, the ADD approach keeps the distance information and bounds the forward traversal at the terminal nodes. This is done at lines 4-6 of procedure $BoundedADDImage$, which handles the cases when the terminal ADD nodes are reached. In particular, when calling the $BoundedADDImage$ procedure on a terminal node $q \neq \infty$, the value of q denotes the distance of the “from” states, and the distance of the new image states is obtained by incrementing q by 1 if the result is still less than distance bound B (line 5 of $BoundedADDImage$); otherwise, the new image states have distance greater than B and are therefore truncated by returning terminal node ∞ (line 6 of $BoundedADDImage$). This truncation mechanism for ADD is an “exact-distance” method, since all states with distance greater than B are truncated, and only the states within distance B are kept. Thus, the ADD approach of Fig. 6 computes exactly the state space \mathcal{S}^B , without using a BFS-style iteration.

3.5 Bounded Saturation using MDDs

In the above EDD and ADD approaches, decision diagrams are used to store both the bounded state space

<pre> void <i>BoundedADDSaturation</i>() 1 $r^* \leftarrow$ root of ADD encoding $f(\mathbf{i}) = \begin{cases} 0 & \text{if } \mathbf{i} \in \mathcal{S}^{init} \\ \infty & \text{otherwise} \end{cases}$; 2 for $k = 1$ to K do 3 foreach node p at level k do 4 <i>BoundedADDSaturate</i>(p); </pre>	<pre> ADD <i>BoundedADDSaturate</i>(ADD p) 1 $l \leftarrow p.lvl$; 2 repeat 3 choose $\alpha \in \mathcal{E}_l, i \in \mathcal{S}_l, j \in \mathcal{S}_l$ s.t. $\mathcal{D}_\alpha[i][j] \neq \mathbf{0}$; 4 $q \leftarrow$ <i>BoundedADDImage</i>($p[i], \mathcal{D}_\alpha[i][j]$); 5 $p[j] \leftarrow$ <i>Minimum</i>($p[j], q$); 6 until p does not change; 7 return p; </pre>
<pre> ADD <i>BoundedADDImage</i>(ADD q, MDD f) 1 if $f = \mathbf{0}$ then return ∞; 2 else if $f = \mathbf{1}$ or $q = \infty$ then return q; 3 $k \leftarrow q.lvl$; ●given our quasi-reduced form, $f.lvl = k$ 4 if $k = 0$ then ●reach terminal nodes 5 if $q < bound$ return $q + 1$; ●increment the distance 6 else return ∞; ●truncated 7 $s \leftarrow$ <i>NewNode</i>(k); ●empty ADD node at level k 8 foreach $i \in \mathcal{S}_k, j \in \mathcal{S}_k$ s.t. $f[i][j] \neq \mathbf{0}$ do 9 $o \leftarrow$ <i>BoundedADDImage</i>($q[i], f[i][j]$); 10 $s[j] \leftarrow$ <i>Minimum</i>($s[j], o$); 11 $s \leftarrow$ <i>BoundedADDSaturate</i>(s); 12 return s; </pre>	<pre> ADD <i>Minimum</i>(ADD p, ADD q) 1 if $p = \infty$ then return q; 2 if $q = \infty$ then return p; 3 $k \leftarrow p.lvl$; ●given our quasi-reduced form, $q.lvl = k$ 4 if $k = 0$ then 5 return $\min(p, q)$; 6 $s \leftarrow$ <i>NewNode</i>(k); ●empty ADD node at level k 7 foreach $i \in \mathcal{S}_k$ do 8 $s[i] \leftarrow$ <i>Minimum</i>($p[i], q[i]$); 9 return s; </pre>

Figure 6: Bounded Saturation using ADDs. The distance encoded at the ADD terminal node is used to bound the state space exploration.

<pre> void <i>BoundedMDDSaturation</i>() ●locally-bounded 1 return <i>BoundedMDDSaturate</i>(root); </pre>	<pre> void <i>BoundedMDDSaturation</i>() ●globally-bounded 1 return <i>BoundedMDDSaturate</i>(0, root); </pre>
<pre> node <i>BoundedMDDSaturate</i>(MDD p) 1 $l \leftarrow p.lvl$; 2 $r \leftarrow p$; ●update in place 3 if $l > 1$ then 4 $r[i] \leftarrow$ <i>BoundedMDDSaturate</i>($r[i]$); 5 for $d = 1$ to B do ●BFS-style bounded iteration 6 $s \leftarrow$ <i>CopyNode</i>(r); 7 foreach $\alpha \in \mathcal{E}_l, i \in \mathcal{S}_l, j \in \mathcal{S}_l$ s.t. $\mathcal{D}_\alpha[i][j] \neq \mathbf{0}$ do 8 $t \leftarrow$ <i>BoundedMDDImage</i>($r[i], \mathcal{D}_\alpha[i][j]$); 9 $s[j] \leftarrow$ <i>Union</i>($s[j], t$); <i>FreeNode</i>(t); 10 if $s = r$ then break; ●new image is empty 11 else $r \leftarrow s$; 12 return r; </pre>	<pre> node <i>BoundedMDDSaturate</i>(int ρ, MDD p) 1 $l \leftarrow p.lvl$; 2 $r \leftarrow$ <i>CopyNode</i>(p); ●work on a copy 3 if $l > 1$ then 4 $r[i] \leftarrow$ <i>BoundedMDDSaturate</i>($r[i]$); 5 for $d = 1$ to $B - \rho$ do ●BFS-style bounded iteration 6 $s \leftarrow$ <i>CopyNode</i>(r); 7 foreach $\alpha \in \mathcal{E}_l, i \in \mathcal{S}_l, j \in \mathcal{S}_l$ s.t. $\mathcal{D}_\alpha[i][j] \neq \mathbf{0}$ do 8 $t \leftarrow$ <i>BoundedMDDImage</i>($\rho + d - 1, r[i], \mathcal{D}_\alpha[i][j]$); 9 $s[j] \leftarrow$ <i>Union</i>($s[j], t$); <i>FreeNode</i>(t); 10 if $s = r$ then break; ●new image is empty 11 else $r \leftarrow s$; 12 return r; </pre>
<pre> MDD <i>BoundedMDDImage</i>(MDD q, MDD T) 1 if $T = \mathbf{0}$ then return 0; if $T = \mathbf{1}$ then return q; 2 $k \leftarrow q.lvl$; 3 $s \leftarrow$ <i>NewNode</i>(k); 4 foreach $i, j \in \mathcal{S}_k$ such that $q[i] \neq \mathbf{0}$ and $T[i][j] \neq \mathbf{0}$ do 5 $t \leftarrow$ <i>BoundedMDDImage</i>($q[i], T[i][j]$); 6 $s[j] \leftarrow$ <i>Union</i>($s[j], t$); <i>FreeNode</i>(t); 7 $s \leftarrow$ <i>BoundedMDDSaturate</i>(s); 8 return s; </pre>	<pre> MDD <i>BoundedMDDImage</i>(int ρ, MDD q, MDD T) 1 if $T = \mathbf{0}$ then return 0; if $T = \mathbf{1}$ then return q; 2 $k \leftarrow q.lvl$; 3 $s \leftarrow$ <i>NewNode</i>(k); 4 foreach $i, j \in \mathcal{S}_k$ such that $q[i] \neq \mathbf{0}$ and $T[i][j] \neq \mathbf{0}$ do 5 $t \leftarrow$ <i>BoundedMDDImage</i>($\rho, q[i], T[i][j]$); 6 $s[j] \leftarrow$ <i>Union</i>($s[j], t$); <i>FreeNode</i>(t); 7 $s \leftarrow$ <i>BoundedMDDSaturate</i>(ρ, s); 8 return s; </pre>

Figure 7: Locally-bounded Saturation using MDDs. The state-space exploration is bounded using the “for”-loop in *BoundedMDDSaturate*.

Figure 8: Globally-bounded Saturation using MDDs. The state-space exploration is bounded using the “for”-loop in *BoundedMDDSaturate* and a global counter.

and the distance information. However, storing the distance information can sometimes increase the sizes of EDDs or ADDs substantially, when compared to the sizes of the MDDs encoding only state spaces. For example, EDD nodes p and q in Fig. 2(d) would be merged, if no distance information were stored. Similarly, ADD nodes r and s in Fig. 2(b) would be merged.

In this section, we explore Bounded Saturation algorithms based on MDDs instead of EDDs or ADDs. The challenge is to bound the symbolic traversal in the Saturation-based fixpoint iteration, so that termination is guaranteed even if the distance information is not encoded by the decision diagram.

Two Bounded Saturation algorithms using MDDs are presented in Figs. 7 and 8. Both start from procedure

BoundedMDDSaturation, where the root MDD node encoding the initial states \mathcal{S}^{init} is saturated by calling the recursive procedure *BoundedMDDSaturate*, which then saturates all the MDD nodes reachable from the root node at lower levels, in a bottom-up fashion. The procedure terminates when the root node is saturated. As for other Saturation-style algorithms, *BoundedMDDSaturate* and *BoundedMDDImage* are mutually recursive, i.e., all newly created MDD nodes in procedure *BoundedMDDImage* are immediately saturated by *BoundedMDDSaturate*.

To perform bounded forward traversal instead of full state-space exploration, procedure *BoundedMDDSaturate* of Figs. 7 and 8 uses BFS-style bounded forward traversals, where MDD nodes r and s are used to denote the reachable (sub-)states of the previous and new iterations, respectively. Thus, the BFS-style iterations are stopped when s is the same as r . In the BFS-style iterations, instead of exploring state spaces from all the reachable (sub-)states, we could have chosen to explore them from the frontier (sub-)states only, computed with an MDD *SetDifference* operation between the newly reachable states with the previously reachable states. However, our experiments showed that this is less efficient, since computing the frontier tends to create many MDD nodes at lower levels.

The two approaches to *BoundedMDDSaturate* differ in the way they bound the BFS-style iteration. The approach of Fig. 7 uses the value B to bound the BFS iteration. The approach of Fig. 8 refines the first approach by utilizing an additional parameter ρ to recursively count the number of event firings that occurred along the path through which the recursion reached MDD node p . The counter ρ is initialized to 0 in procedure *BoundedMDDSaturate*, and then incremented with the iteration number d at line 8 of procedure *BoundedMDDSaturate*. To bound the forward traversal, the number of BFS-style iterations that can be performed when saturating MDD node p is then reduced to $B - \rho$ in procedure *BoundedMDDSaturate*.

We call the approach of Fig. 7 *locally-bounded* and the one of Fig. 8 *globally-bounded*. This is because, in the latter case, the bound of the BFS-style iteration does not only take into account the event firings that have occurred locally when saturating an MDD node p , but also those along the path reaching node p . Given a bound B , both approaches compute a superset of the bounded state space \mathcal{S}^B . The locally-bounded approach may contain reachable states with distance at most B^K , whereas the globally-bounded approach may contain reachable states with distance at most $\binom{B+K-1}{K}$, where K is the number of state variables, i.e., MDD levels. We prove these bounds by induction. Let D be the maximum state distance to \mathcal{S}^{init} in the current state space during the bounded symbolic forward traversal, initially set to 0.

In the locally-bounded approach, each call to recursive procedure *BoundedMDDSaturate* on an MDD node

at level 1 can increase D by at most B , due to the bound enforced in line 8 of *BoundedMDDSaturate*. This proves the base case. Now assume that procedure *BoundedMDDSaturate*, on any node at level $l-1$, increases D by at most B^{l-1} . Then, for a node p at level l and each BFS-style iteration of *BoundedMDDSaturate* on p , procedure *BoundedMDDImage* calls on the child nodes of p at level $l-1$ and generates a new node at level $l-1$, which is also saturated by *BoundedMDDSaturate* and thus increases D by at most B^{l-1} . Because there are at most B such iterations, *BoundedMDDSaturate* on node p at level l can therefore increment D by at most $B \times B^{l-1} = B^l$. Our upper-distance bound then follows since $l = K$ for the root node.

For the globally-bounded approach, we prove that a call to recursive procedure *BoundedMDDSaturate* with counter ρ on some MDD node p at level l can increment D by at most $\binom{B-\rho+l-1}{l}$, for $B \geq \rho \geq 0$. Our upper-distance bound can then be proved to hold since $\rho = 0$ and $l = K$ for the root node. Regarding the base case, for a node at level 1, it is easy to see that D can be incremented by at most $B - \rho$. Now assume that the bound formula is true for level $l-1$. Regarding level l , the number of BFS-style iterations in procedure *BoundedMDDSaturate* is $B - \rho$, and at the d^{th} iteration, the result of *BoundedMDDImage* is saturated by *BoundedMDDSaturate* at level $l-1$, which can increment D by at most $\binom{B-\rho-d+l-1}{l-1}$. The total increment is therefore $\sum_{d=1}^{B-\rho} \binom{B-\rho-d+l-1}{l-1}$, which can be simplified to $\binom{B-\rho+l-1}{l}$, as desired.

Compared to the locally-bounded approach, the new counter ρ restricts the explored state space more, since its distance bound $\binom{B+K-1}{K}$ is $o(B^K)$, i.e., asymptotically smaller than for the locally-bounded approach. However, both *BoundedMDDSaturate* and *BoundedMDDImage* have to compute new results for different values of ρ , whence ρ becomes part of the search key in the operation caches for these procedures. This results in fewer cache hits and higher memory usage.

4 Experimental results

We implemented our Bounded Saturation algorithms in the verification tool SMART [8], which supports Petri nets as front-end. This section reports our experimental results for a suite of asynchronous Petri-net benchmarks when checking for deadlock-freedom, as an example of bounded reachability checking. For our symbolic algorithms, the deadlock check simply requires us, for each event α , to remove the set of states enabling α , i.e., $Image^{-1}(\widehat{\mathcal{S}}, \mathcal{D}_\alpha)$, from the final bounded state space; any remaining state corresponds to a deadlock. In the following, we compare the performance of several decision-diagram-based methods and the SAT-based methods of Heljanko et al. [22,23] and Ogata et al. [31], when applied to this task.

Model	#P	#E	Approximate distance methods																		
			EVMDD-Approx			EVBDD-Approx			MDD-SatL			MDD-SatG			MDD-Chain			SAT-S		SAT-C	
			B	Time	Mem	B	Time	Mem	B	Time	Mem	B	Time	Mem	B	Time	Mem	B	Time	B	Time
byzagr4(2a)	579	473	49	2.23	2.41	49	9.14	3.43	2	2.15	1.47	2	3.24	4.18	6	7.3	9.24	8	0.79	2	2.07
mmgt(3)	122	172	9	0.11	0.2	8	1.28	0.34	2	0.06	0.11	3	0.2	0.46	5	0.07	0.16	7	0.09	3	1.04
mmgt(4)	158	232	17	1.22	1.15	17	2.15	1.67	2	0.4	0.56	2	0.6	1.41	3	0.11	0.2	8	0.23	4	5.52
dac(15)	105	73	4	0.01	0.0	4	0.03	0.01	2	0.01	0.01	2	0.01	0.01	2	0.01	0.01	3	0.01	2	0.04
hs(75)	302	152	151	0.01	0.03	151	0.36	0.05	3	0.03	0.03	4	0.03	0.07	93	0.08	0.53	151	5.84	1	0.07
hs(100)	402	202	201	0.03	0.04	201	0.78	0.07	2	0.05	0.03	3	0.05	0.09	116	0.14	0.78	201	14.85	1	0.13
sentest(75)	252	102	45	0.0	0.02	45	0.21	0.03	2	0.02	0.01	3	0.03	0.03	32	0.03	0.21	83	4.27	3	0.13
sentest(100)	327	127	61	0.01	0.03	61	0.34	0.05	2	0.03	0.02	3	0.03	0.04	73	0.07	0.47	108	10.71	4	0.29
speed(1)	29	31	4	0.01	0.02	2	0.24	0.01	3	0.0	0.01	4	0.01	0.04	3	0.01	0.04	4	0.01	2	0.03
dp(12)	72	48	2	0.01	0.02	2	0.02	0.03	1	0.01	0.02	1	0.01	0.01	1	0.0	0.01	1	0.0	1	0.02
q(1)	163	194	9	0.01	0.03	8	1.45	0.04	2	0.03	0.04	3	0.05	0.11	7	0.06	0.14	9	0.13	1	0.07
elevator(3)	326	782	8	15.07	9.46	7	28.5	9.83	3	5.94	2.1	4	23.44	14.33	6	0.87	0.58	8	0.42	2	3.77
key(2)	94	92	13	0.06	0.14	18	0.16	0.19	3	0.07	0.12	4	0.06	0.19	14	0.07	0.2	36	2.88	2	0.05
key(3)	129	133	17	0.2	0.48	17	0.55	0.71	2	0.26	0.38	2	0.18	0.42	14	0.21	0.52	37	4.39	2	0.1
key(4)	164	174	17	0.69	1.48	15	2.4	1.39	2	1.35	1.56	3	3.43	5.75	17	0.67	1.54	38	4.21	2	0.18
key(5)	199	215	17	2.04	4.15	17	5.97	6.66	2	2.88	3.38	3	26.74	23.72	15	1.73	3.37	39	8.07	2	0.25
fms(3)	22	16	9	0.06	0.02	5	0.74	0.02	6	0.0	0.02	8	0.0	0.06	7	0.01	0.08	10	0.75	3	1.25
fms(7)	22	16	19	0.07	0.26	11	4.4	0.69	14	0.04	0.28	16	0.12	1.29	15	0.24	2.58	18	>600	6	>600
fms(10)	22	16	28	0.12	0.99	6	>600	-	20	0.15	1.14	22	0.51	5.69	21	1.35	14.75	16	>600	7	>600
kanban(1)	17	16	28	0.04	0.0	27	0.33	0.01	10	0.0	0.0	12	0.0	0.01	13	0.0	0.01	19	0.05	5	0.09
kanban(3)	17	16	82	0.05	0.06	79	5.34	0.34	30	0.01	0.05	32	0.12	0.64	19	0.03	0.23	12	>600	3	>600
kanban(10)	17	16	271	0.84	10.43	1	>600	-	100	1.69	8.48	102	46.96	317.48	54	2.83	29.29	1	>600	1	>600

Table 1: Experimental results (Time in sec, Mem in MB). “>600” means runtime exceeds 600sec or memory exceeds 1GB, and “-” means the memory usage is not available due to time out.

We ran our experiments on a 3GHz Pentium machine with 1GB RAM. Benchmarks *byzagr4*, *mmgt*, *dac*, *hs(hartstone)*, *sentest*, *speed*, *dp*, *q*, *elevator*, and *key* are taken from Corbett [19] and were translated into safe Petri nets by Heljanko [21].¹ Benchmarks *fms* and *kanban* are deadlocked versions of non-safe Petri-net manufacturing system models which are included in the SMART distribution; these are automatically translated into safe Petri nets by SMART. All benchmarks have deadlocks.

BDDs and EVBDDs are natural candidates for our decision-diagram-based approaches when models have binary variables, as is the case for safe Petri nets. However, thanks to a heuristic to merge binary variables and exploit Petri net invariants [38], we can instead use MDDs and EDDs, thereby achieving time and memory savings. In the following, we thus present the multi-valued version of our algorithms and, for comparison, consider only one EVBDD-based approach (EVBDD-Approx), applied to safe Petri net models. The MDD- and EDD-based approaches employ the merging heuristic for the safe nets of Corbett’s benchmarks, while they use the non-safe Petri nets *fms* and *kanban* as-is. Moreover, variable orders for our experiments were automatically obtained using the heuristic in [33].

4.1 Result tables

Tables 1 and 2 show the results for our “approximate” methods and “exact” methods, respectively. The “approximate” methods are:

- **MDD-Chain**, the BFS-style, event-locality-based chaining technique of Fig. 3;
- **SAT-S**, the circuit SAT-based method with step semantics of [23];
- **SAT-C**, the CNF SAT-based method with forward chaining of [31];

as well as those methods that compute a superset of the states \mathcal{S}^B within distance B :

- **EDD-Approx** and **EVBDD-Approx**, our EDD-based Bounded Saturation (*TruncateApprox*);
- **MDD-SatL**, our MDD-based Locally-bounded Saturation;
- **MDD-SatG**, our MDD-based Globally-bounded Saturation.

The “exact” methods, which limit their search to exactly \mathcal{S}^B , are:

- **SAT-I**, the circuit SAT-based method with interleaving semantics of [23];
- **EDD-Exact** and **ADD-Exact**, our EDD or ADD-based Bounded Saturation (*TruncateExact*).

¹ A Petri net is safe if any place can contain at most one token, and it is non-safe but N -bounded, if any place can contain at most N ($N > 1$) tokens. A non-safe Petri net can be translated into a safe one by binary encoding (bit-blasting) of its non-safe places with safe places.

The first three columns of both tables are identical, and display the model name and parameters, as well as the number of places ($\#P$) and events ($\#E$). For each approximate method of Table 1, we report the smallest bound B at which either a deadlock is found or the runtime exceeds 10 minutes. For the exact methods of Table 2, we state the exact distance bound B of the deadlock, except for the case marked “?”, where none of the exact methods could find a deadlock within 10 minutes. All the decision-diagram-based methods are implemented in SMART, and their runtime and memory consumptions are included in the table, while only the runtimes are available for the SAT-based tools.

Corbett’s benchmarks and the SAT-I and SAT-C tools are taken from [22]. In our experiments, SAT-S performs at least as well as the analogous approach using process semantics [21] (this is also confirmed by the results in Heljanko and Junttila’s recent tutorial [22]). Therefore, we report only results for the former approach in Table 1. With Corbett’s benchmarks, we show different bounds for SAT-C than those reported in [31]; this is due to choosing a different initial state, the same as the one considered in [22]. For SAT-I and SAT-C, both the encoding time and the **bczchaff** circuit SAT-solver runtime are displayed in Table 1. For a fair comparison, the runtime of SAT-C includes the preprocessing steps for scheduling events, the encoding of the safe Petri nets into boolean formulas and then into CNF formulas, and the querying of the **zchaff** SAT-solver for deadlocks.

4.2 Discussion

From Tables 1 and 2, we can roughly classify benchmarks *byzagr*, *hs*, *sentest*, *fms*, and *kanban* as models with “deep” deadlocks, where the minimum bounds required to detect deadlocks range from 30 to 500, and all other benchmarks as models with “shallow” deadlocks, where the minimum bounds are less than 30. For benchmarks with “deep” deadlocks, the newly proposed EDD-Approx and MDD-SatL methods achieve the best performance. For models with “shallow” deadlocks, it seems that almost all methods perform reasonably well, including our MDD-Chain method. When comparing EDD-Approx with EVBDD-Approx, we observe that the former always performs better than the latter. Further, MDD-SatL always performs better than MDD-SatG in terms of both time and memory, and the latter always finds deadlock states at a deeper bound. The comparison between EDD-Exact and ADD-Exact shows that they complement each other. EDD-Approx and MDD-SatL are arguably the two methods with the best overall performance, except for the elevator model, where they perform worse than the MDD-Chain method and the SAT-S method. This might be because a very large superset of \mathcal{S}^B is computed; we also suspect that our variable order heuristic is not performing well on this model.

Model	#P	#E	Exact distance methods					
			B	SAT-I	EDD-Exact	ADD-Exact		
				Time	Time	Mem	Time	Mem
byzagr4(2a)	579	473	?	>600	>600	–	>600	–
mmgt(3)	122	172	10	1.37	0.32	0.55	0.41	0.33
mmgt(4)	158	232	20	1.24	4.36	3.12	12.87	3.61
dac(15)	105	73	20	0.01	0.03	0.05	0.06	0.04
hs(75)	302	152	151	7.94	0.15	0.03	0.13	0.34
hs(100)	402	202	201	20.31	0.3	0.04	0.23	0.58
sentest(75)	252	102	88	8.51	0.06	0.02	0.08	0.14
sentest(100)	327	127	113	21.85	0.12	0.03	0.22	0.25
speed(1)	29	31	7	0.02	0.1	0.04	0.02	0.01
dp(12)	72	48	12	0.06	0.96	1.77	0.33	0.12
q(1)	163	194	21	0.83	0.08	0.15	0.19	0.13
elevator(3)	326	782	20	2.74	>600	–	7.54	1.83
key(2)	94	92	50	>600	0.15	0.2	0.22	0.34
key(3)	129	133	50	>600	0.62	0.67	2.8	1.64
key(4)	164	174	50	>600	2.02	2.11	9.71	3.15
key(5)	199	215	50	>600	16.87	10.52	33.65	10.03
fms(3)	22	16	30	>600	0.07	0.06	0.05	0.14
fms(7)	22	16	70	>600	0.8	2.2	1.12	4.7
fms(10)	22	16	100	>600	5.37	14.37	5.24	24.11
kanban(1)	17	16	40	16.56	0.08	0.0	0.01	0.01
kanban(3)	17	16	120	>600	0.1	0.07	0.27	0.64
kanban(10)	17	16	400	>600	14.4	10.46	51.76	187.9

Table 2: Experimental results (Time in sec, Mem in MB). “>600” means runtime exceeds 600sec or memory exceeds 1GB, and “–” means the memory usage is not available due to time out.

In addition, we observe that the well-known poor performance of SAT-solvers for unsatisfiable boolean formulas makes it hard to guess bound B . If the guess is too large, the resulting boolean formula is huge; if it is too small, the formula is unsatisfiable. Both cases have severe performance penalties. For example, SAT-I finds a deadlock in benchmark $q(1)$ in less than 1 second when $B = 21$ but, when $B = 20$, the formula is unsatisfiable and the runtime exceeds 600 seconds. Decision-diagram-based methods tend instead to have “well-behaved” runtimes, monotonically increasing in B .

5 Related work

This section discusses our approaches to bounded reachability checking in light of related work.

5.1 SAT-solving for Petri nets

We first add some details to the two SAT-based approaches to deadlock checking of safe Petri nets [21, 31], against which we compared ourselves in the previous section regarding runtime efficiency.

Heljanko’s work [21] established the so-called *process semantics* of Petri nets as the ‘best’ net semantics for translating bounded reachability into a propositional satisfiability problem, in the sense that the resulting SAT

problem can be solved more efficiently than for step or interleaving semantics. However, this technique can only be safely applied to safe Petri nets, as otherwise these semantics may not coincide. In contrast, our technique is applicable to general Petri nets, even to Petri nets exhibiting infinite state spaces.

Ogata, Tsuchiya, and Kikuno’s approach [31] focuses on the translation of Petri nets, which must again be safe, into propositional formulas. The ordinary encoding of safe nets into propositional formulas results in large formulas, thereby degrading the performance of SAT solvers and hampering scalability. The authors suggest a more succinct encoding, albeit at the price of exploring not only states with a distance up to the considered bound but also some states with larger distance. This is similar to our Bounded Saturation, for which it is also more efficient to collect some additional states. The authors leave a comparison to Heljanko’s approach as future work; this comparison has now been conducted by us, and the results reported in the previous section show that neither method is superior in all cases.

5.2 BDD vs. SAT on synchronous systems

As mentioned above, the common belief that SAT-based model checkers outperform model checkers based on decision diagrams was already proved wrong by Cabodi, Nocco, and Quer [6], for a class of digital circuits that

exhibit largely synchronous behavior. The advocated approach relies on improving standard BDD-based techniques by mixing forward and backward traversals, dovetailing approximate and exact methods, adopting guided and partitioned searches, and using conjunctive decompositions and generalized cofactor-based BDD simplifications.

Our research complements their findings, regarding asynchronous systems. In a nutshell, our improvement over standard techniques lies in the local manipulation of decision diagrams by exploiting the event locality inherent in asynchronous systems, interleaving semantics, and disjunctive partitioning. These are the central ideas behind *Saturation* [9], on which our *Bounded Saturation* algorithms are based. Similar to the algorithm proposed in [6], we also achieve efficiency by including some states with a distance larger than the given bound B ; such states have a distance of up to $K \cdot B$ in our EDD-based approach and up to $E \cdot B$ in [6], where K and E are the number of components and events, respectively, in the studied Petri net.

Together, the results of Cabodi et al. and ours, and also further recent research [36], revise some of the claims made in the literature, especially regarding the performance of decision-diagram-based bounded model checking. It must be noted here that our results have been obtained with static variable orders that have been computed using a simple heuristic [33]. Thus, unlike in [18], no fine-tuning of models by hand was necessary.

5.3 Petri net unfoldings

Both SAT-based and decision-diagram-based techniques are established approaches to addressing the state-space explosion problem. The Petri net community has developed another successful approach to this problem, which was first suggested in a seminal paper by McMillan [28]. The idea is to finitely unfold a Petri net until the resulting prefix has exactly the same reachable markings as the original net. For certain Petri nets such finite prefixes exist and often prove to be small in practice. In contrast to bounded reachability checking, analysis techniques based on unfoldings are thus complete, as they capture a net's entire behavior. However, unfoldings are limited to finite-state Petri nets, although recent work suggests an extension to some infinite-state systems [1].

6 Conclusions and future work

This article explored the utility of decision diagrams for the bounded reachability checking of asynchronous systems. To this end, we reconsidered *Saturation*, a state-space generation algorithm that is based on Multi-valued Decision Diagrams (MDDs) and exploits the event locality and interleaving semantics inherent in asynchronous

systems. As the search strategy of *Saturation* is unlike breadth-first search, bounding searches required us to either employ EDDs or ADDs, which allow for storing states together with their distances from the set of initial states, or use altogether new variants of *Saturation*.

An extensive experimental analysis of the resulting *Bounded Saturation* algorithms showed that they often compare favorably to the competing SAT-based approaches introduced in [21,22,31]. In many cases, *Bounded Saturation* could build bounded state spaces and check for deadlocks at least as fast and frequently faster, while using acceptable amounts of memory. Thus, decision-diagram-based techniques can well compete with SAT-based techniques for the bounded reachability checking of asynchronous systems, and the widespread perception that decision diagrams are not suited for bounded model checking [18] is unfounded.

Future work should investigate whether the *Bounded Saturation* algorithms proposed in this article can be efficiently applied beyond reachability checking. We also intend to investigate whether the event locality inherent in asynchronous systems can be exploited in SAT-based reachability checking.

Acknowledgments

We thank K. Heljanko, T. Jussila, and T. Tsuchiya for providing us with benchmarks and software tools that we used in our study. We especially thank M. Y. Vardi for inspiring comments and suggestions after we presented an earlier version of this paper at the TACAS 2007 conference. In particular, his suggestion to look into ways to avoid having to store distance information explicitly, prompted us to derive the MDD-based *Bounded Saturation* algorithms of Sec. 3.5.

References

1. P. Abdulla, S. Iyer, A. Nylén. SAT-solving the coverability problem for Petri nets. *FMSD*, 24(1):25–43, 2004.
2. R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, F. Somenzi. Algebraic decision diagrams and their applications. *FMSD*, 10(2/3):171–206, 1997.
3. A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu. Symbolic model checking without BDDs. *TACAS*, LNCS 1579, pp. 193–207, 1999. Springer.
4. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Comp. Surv.*, 24(3):293–318, 1992.
5. J. R. Burch, E. M. Clarke, D. E. Long. Symbolic model checking with partitioned transition relations. *VLSI*, pp. 49–58, 1991.
6. G. Cabodi, S. Nocco, S. Quer. Are BDDs still alive within sequential verification? *STTT*, 7(2):129–142, 2005.
7. G. Ciardo. Petri nets with marking-dependent arc multiplicity: properties and analysis. *ICATPN*, LNCS 815, pp. 179–198, 1994. Springer.

8. G. Ciardo, R. L. Jones, A. S. Miner, R. Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578–608, 2006.
9. G. Ciardo, G. Lüttgen, R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. *TACAS*, LNCS 2031, pp. 328–342, 2001. Springer.
10. G. Ciardo, R. Marmorstein, R. Siminiceanu. The Saturation algorithm for symbolic state space exploration. *STTT*, 8(1):4–25, 2006.
11. G. Ciardo, A. S. Miner, G. Lüttgen. Exploiting interleaving semantics in symbolic state-space generation. *FMSD*, 31(1):63–100, 2007.
12. G. Ciardo, R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. *FM-CAD*, LNCS 2517, pp. 256–273, 2002. Springer.
13. G. Ciardo, A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. *CHARME*, LNCS 3725, pp. 146–161, 2005. Springer.
14. A. Cimatti, E. M. Clarke, F. Giunchiglia, M. Roveri. NuSMV: A new symbolic model verifier. *CAV*, LNCS 1633, pp. 495–499, 1999. Springer.
15. E. M. Clarke, M. Fujita, X. Zhao. Application of multi-terminal binary decision diagrams. *IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 1995.
16. E. M. Clarke, A. Biere, R. Raimi, Y. Zhu. Bounded model checking using satisfiability solving. *FMSD*, 19(1):7–34, 2001.
17. E. M. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press, 1999.
18. F. Copt, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, M. Y. Vardi. Benefits of bounded model checking at an industrial setting. *CAV*, LNCS 2102, pp. 436–453, 2001. Springer.
19. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–180, 1996.
20. The VIS Group. VIS: A system for verification and synthesis. *CAV*, LNCS 1102, pp. 428–432, 1996. Springer.
21. K. Heljanko. Bounded reachability checking with process semantics. *CONCUR*, LNCS 2154, pp. 218–232, 2001. Springer.
22. K. Heljanko, T. Junttila. Advanced tutorial on bounded model checking. *ACSD/ICATPN*, 2006. <http://www.tcs.hut.fi/~kepa/bmc-tutorial.html>.
23. K. Heljanko, I. Niemelä. Answer set programming and bounded model checking. *Answer Set Programming*, 2001.
24. F. Ivančić, Z. Yang, M. Ganai, A. Gupta, P. Ashar. F-Soft: Software Verification Platform. *CAV*, LNCS 3576, 2005. Springer.
25. T. Kam, T. Villa, R. Brayton, A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
26. Y.-T. Lai, S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. *DAC*, pp. 608–613, 1992. IEEE Press.
27. B. Li, C. Wang, F. Somenzi. Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. *STTT*, 7(2):143–155, 2005.
28. K. McMillan. A technique of state space search based on unfolding. *FMSD*, 6(1):45–65, 1995.
29. K. McMillan. Interpolation and SAT-Based Model Checking. *CAV*, LNCS 2725, pp. 1–13, 2003. Springer.
30. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an efficient SAT solver. *DAC*, pp. 530–535, 2001. ACM Press.
31. S. Ogata, T. Tsuchiya, T. Kikuno. SAT-based verification of safe Petri nets. *ATVA*, LNCS 3299, pp. 79–92, 2004. Springer.
32. I. Rabinovitz, O. Grumberg. Bounded model checking of concurrent programs. *CAV*, LNCS 3576, pp. 82–97, 2005. Springer.
33. R. Siminiceanu, G. Ciardo. New metrics for static variable ordering in decision diagrams. *TACAS*, LNCS 3920, pp. 90–104, 2006. Springer.
34. M. Sheeran, S. Singh, G. Stålmarck. Checking safety properties using induction and a SAT-solver. *FM-CAD*, LNCS 1954, pp. 108–125, 2000. Springer.
35. M. Solé, E. Pastor. Traversal techniques for concurrent systems. *FM-CAD*, LNCS 2517, pp. 220–237, 2002. Springer.
36. R. Tzoref, M. Matusevich, E. Berger, I. Beer. An optimized symbolic bounded model checking engine. *CHARME*, LNCS 2860, pp. 141–149, 2003. Springer.
37. Rüdiger Valk. Generalizations of Petri nets. In *Mathematical foundations of computer science*, LNCS 118, pp. 140–155, 1981. Springer.
38. A. J. Yu, G. Ciardo, G. Lüttgen. Improving static variable orders via invariants. *ICATPN*, LNCS 4546, pp. 83–103, 2007. Springer.