Bounded Reachability Checking of Asynchronous Systems Using Decision Diagrams*

Andy Jinqing Yu[1], Gianfranco Ciardo[1], and Gerald Lüttgen[2]

[1] Department of Computer Science and Engineering, University of California, Riverside, CA 92521, USA {jqyu, ciardo}@cs.ucr.edu
[2] Department of Computer Science, University of York, York YO10 5DD, U.K. luettgen@cs.york.ac.uk

Abstract. Bounded reachability or model checking is widely believed to work poorly when using decision diagrams instead of SAT procedures. Recent research suggests this to be untrue with regards to synchronous systems, particularly digital circuits. This paper shows that the belief is also a myth for asynchronous systems, such as models specified by Petri nets. We propose *Bounded Saturation*, a new algorithm to compute bounded state spaces using Multi-way Decision Diagrams (MDDs). This is based on the established Saturation algorithm which benefits from a non-standard search strategy that is very different from breadth-first search. To bound Saturation, we employ Edge-Valued MDDs and rework its search strategy. Experimental results show that our algorithm often, but not always, compares favorably against two SAT-based approaches advocated in the literature for deadlock checking in Petri nets.

1 Introduction

Bounded model checking is a well-established technique to reason about reactive systems [3]. Unlike conventional model checking based on explicit or symbolic representations of state spaces [13], 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 which 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., [24]), such satisfiability problems can often be decided efficiently.

BDDs vs. SAT. Bounded model checking is an incomplete verification technique unless the bound exceeds the state space diameter. However, as many faults involve relatively short counterexamples in practice, the technique has proved itself an efficient debugging aid, and bounded model checkers are now used to verify *digital circuits* [12], *Petri nets* [17,25], and *software* [20,26]. Several studies have found such tools beneficial in industrial settings, especially when compared to symbolic model checkers using decision diagrams [14].

* Research supported by the NSF under grants CNS-0501747 and CNS-0501748 and by the EPSRC under grant GR/S86211/01.

It is often believed that SAT methods are key to the performance of bounded model checkers. Recent research by Cabodi et al. [5], however, counters this suggestion. Their work proposes enhancements to standard techniques based on *Binary Decision Diagrams* (BDDs), making BDD-based bounded model checking competitive with SAT-based approaches. These 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 governed by interleaving semantics, such as distributed algorithms expressed in Petri nets.

This paper. 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 a new approach for bounded reachability checking using decision diagrams based on *Saturation* [7], an established symbolic algorithm for generating the state space of asynchronous systems. By taking into account event locality and interleaving semantics and by using *Multi-way Decision Diagrams* (MDDs) instead of BDDs, Saturation is often orders of magnitude more efficient than breadth-first search algorithms implemented in popular model checkers [11].

The difficulty in adapting Saturation to bounded reachability checking lies in its non-standard search strategy that is completely different from breadth-first search. We then cope by storing not only the reachable states but also the distance of each state from the initial state(s), using the *edge-valued* decision diagrams of [9]. These extend EVBDDs [22] just as MDDs extend BDDs, and use a more general reduction rule. Each state stored in such a decision diagram corresponds to a path from the root to the only terminal node, whereas the distance of a state is the sum of the weights of the edges along that path.

The resulting *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 "components" of the underlying asynchronous system. Just as ordinary breadth-first search, both can find minimal-length counterexamples. However, the second variant is usually more efficient in terms of runtime and memory, even if it discovers more states. Such behavior, while counterintuitive at first, is not uncommon with decision diagrams.

Experiments and results. We evaluate Bounded Saturation against two SAT-based approaches for bounded reachability checking proposed by Heljanko [17] and Ogata, Tsuchiya, and Kikuno [25], both aimed at finding deadlocks in asynchronous systems specified by Petri nets. We implemented our algorithm in SMART [6], and ran it on the suite of examples used in both [17] and [25], first proposed by Corbett in [15], and on two models from the SMART release. The static variable ordering used in our algorithm was computed via a heuristic [27].

Our experiments show that Bounded Saturation performs better or at par with competing SAT-based algorithms, and is less efficient in only few cases. Thus, it is a myth that decision diagrams are uncompetitive w.r.t. SAT solvers for bounded model checking; just as the roles of bounded and unbounded model checking are complementary, so are the use of SAT solving and decision diagrams.

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 $(x_K, ..., x_1)$ of K *local state* variables where, for $K \geq l \geq 1$, $x_l \in \mathcal{S}_l = \{0, 1, ..., n_l - 1\}$ with $n_l > 0$, is the the l^{th} *local* state variable. Thus, $\widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ and we write $\mathcal{R}(\mathbf{i}[K], ..., \mathbf{i}[1], \mathbf{j}[K], ..., \mathbf{j}[1])$, or $\mathcal{R}(\mathbf{i}, \mathbf{j})$, if the model can move from *current state* $\mathbf{i}$ to *next state* $\mathbf{j}$ in one step.

Computation of the 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 $\mathcal{S} \supseteq Img(\mathcal{S}, \mathcal{R})$, where the *image computation* function $Img(\mathcal{X}, \mathcal{R}) = \{\mathbf{j} : \exists \mathbf{i} \in \mathcal{X}, \mathcal{R}(\mathbf{i}, \mathbf{j})\}$ describes the successors to the set of states $\mathcal{X}$. In bounded model checking, only part of this state space is considered, the set of states within some distance bound B from $\mathcal{S}^{init}$.

Most symbolic approaches encode x_l in b_l boolean variables, where b_l is either n_l or $\lceil \log n_l \rceil$ (*one-hot* or *binary* encoding), and a set of states using a BDD with $\sum_{K \geq l \geq 1} b_l$ levels. *Ordered multi-way decision diagrams* (MDDs) [21], 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 [6], or as an interface to BDDs [16].

Symbolic technique for asynchronous models. A BFS-based approach, as used for example by NuSMV [11], 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 partitioned into a set of *disjuncts* [4], 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.

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

For example, a guarded command language model consists of a set of commands of the form "$guard \rightarrow assignment_1 \parallel assignment_2 \parallel \cdots \parallel assignment_m$", with the meaning that, whenever the boolean predicate *guard* evaluates to *true*, the m parallel atomic assignments can be executed concurrently (synchronously). Commands 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 the input and output places connected to it. We use extended Petri nets as the input formalism in SMART [6].

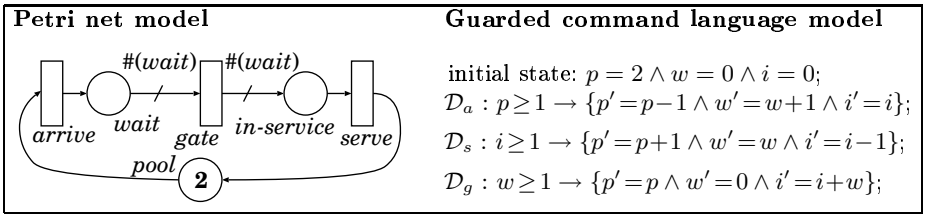

Fig. 1: A limited-arrival gated-service model with marking-dependent arc cardinalities.

Running example. Fig. 1 shows a Petri net, and its equivalent guarded command language expression, modeling a gated-service queue with a limited pool of customers. New arrivals wait at the gate until it is opened, then all the 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*, w for *wait*, and i for *in-service*. Assuming a pool of two customers, the model has an initial state (2,0,0) and six reachable states: $\mathcal{S} = \{(2,0,0), (1,1,0), (0,2,0), (1,0,1), (0,0,2), (0,1,1)\}$.

Event locality. In asynchronous models, the execution of each event usually modifies or depends on just a small subset of all the state variables. In the running example, for 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 set of variables

- $\mathcal{V}_M(\alpha) = \{x_l : \exists \mathbf{i}, \mathbf{j} \in \widehat{\mathcal{S}}, \mathcal{D}_\alpha(\mathbf{i}, \mathbf{j}) \wedge \mathbf{i}[l] \neq \mathbf{j}[l]\}$ and
- $\mathcal{V}_D(\alpha) = \{x_l : \exists \mathbf{i}, \mathbf{i}' \in \widehat{\mathcal{S}}, \forall k \neq l, \mathbf{i}[k] = \mathbf{i}'[k] \wedge \exists \mathbf{j} \in \widehat{\mathcal{S}}, \mathcal{D}_\alpha(\mathbf{i}, \mathbf{j}) \wedge \nexists \mathbf{j}' \in \widehat{\mathcal{S}}, \mathcal{D}_\alpha(\mathbf{i}', \mathbf{j}')\}$

that can be modified by α, or can disable α, respectively. Letting

$$Top(\alpha) = \max\{l : x_l \in \mathcal{V}_M(\alpha) \cup \mathcal{V}_D(\alpha)\}, Bot(\alpha) = \min\{l : x_l \in \mathcal{V}_M(\alpha) \cup \mathcal{V}_D(\alpha)\},$$

we can then partition the events according to the value of *Top*, into the subsets $\mathcal{E}_l = \{\alpha : Top(\alpha) = l\}$, for $K \geq l \geq 1$. In [8] we observed that a chaining [28] order where these subsets are applied to the MDD in bottom-up fashion results in good speedups with respect to a strict BFS symbolic state-space generation. The bounded version of chaining is shown in Fig. 3 and discussed in Sec. 3.

By exploiting this *event locality*, we can store $\mathcal{D}_\alpha$ 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*, i.e., remain unchanged.

Saturation-based symbolic fixpoint computation. The Saturation algorithm for computing the reachable state spaces of asynchronous systems was originally proposed in [7] for models in *Kronecker-product* form; it has since been extended to general models [10] and applied to shortest path computations and CTL model-checking [9]. Saturation has been shown to reduce memory and runtime requirements by several orders of magnitude with respect to BFS-based algorithms, when applied to asynchronous systems.

Saturation is unique in that it does not perform fixpoint computations over a global decision diagram, as standard breadth-first iteration strategies do, but recursively computes (sub-)fixpoints at each decision diagram node. This exploits the locality of events inherent in asynchronous systems as well as the semantic

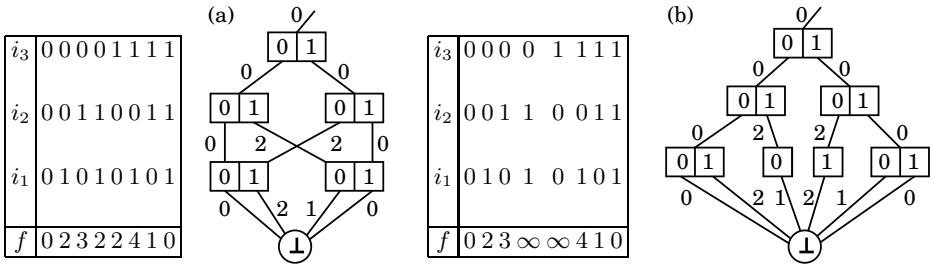

Fig. 2: Storing total (a) and partial (b) integer functions with EDDs.

concept of interleaving. The formal algorithm of Saturation on our variant of edge-valued decision diagrams is described in detail in Sec. 3. For details of the original Saturation algorithm on MDDs we refer the reader to [10].

To employ Saturation for bounded reachability checking we encode not just the reachable states, but also their distance from $\mathcal{S}^{init}$. This can be achieved using either edge-valued decision diagrams (EDDs, called EV$^+$MDDs in [9]) or the ADDs of [2]. ADDs are a well-known variant of BDDs that can encode non-boolean functions by having an arbitrary set of terminal nodes instead of just the two terminal nodes corresponding to the boolean values *true* and *false*. In our discussion, we focus instead on EDDs since they can be exponentially more compact than ADDs; see also the results in Sec. 4.

Definition 1 (EDD [9]). An EDD on the *domain* $\widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ is a directed acyclic graph with labeled and weighted edges where:

- Each node p belongs to a *level* in $\{K, ..., 1, 0\}$, denoted $p.lvl$.
- 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 from 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. We also 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 $p.lvl - 1$.
- There is a single root node, $r^\star$, at level K, with an incoming "dangling" edge having weight $\rho \in \mathbb{Z}$. We write such edges as $\langle \rho,r \rangle$.
- Each non–terminal node has at least one outgoing edge labeled with 0.
- There are no *duplicate* nodes, i.e., if $\forall i, 0 \leq i < n_l, p[i].node = q[i].node$ and $p[i].val = q[i].val$, then $p = q$.

The function $f_{\langle v,p \rangle} : \mathcal{S}_l \times \cdots \times \mathcal{S}_1 \to \mathbb{Z} \cup \{\infty\}$ encoded by an edge $\langle v,p \rangle$, with $p.lvl = l > 0$ is $f_{\langle v,p \rangle}(i_l, \ldots, i_1) = v + f_{\langle p[i_l].val, p[i_l].node \rangle}(i_{l-1}, \ldots, i_1)$, where we let $f_{\langle x,\perp \rangle} = x$. Thus, the function encoded by the entire EDD is $f_{\langle \rho,r \rangle}$. □

As defined, EDDs can canonically encode *any* function of the form $\widehat{\mathcal{S}} \to \mathbb{Z} \cup \{\infty\}$, except the constant ∞, for which we use a special EDD with $r = \perp$ and $\rho = \infty$. Fig. 2 shows two EDDs storing a total and partial function, respectively. Here, "partial" means that some of its values are ∞; whenever this is the case, we omit

```
mdd BoundedBfsChain( )
 1 S ← S^init;
 2 for d = 1 to B do
 3   for l = 1 to K do
 4     foreach α ∈ E_l do
 5       S ← Union(S, Image(S, D_α))
 6 return S;
```

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

the corresponding value and edge from the graphical representation. We point out that EDDs allow for efficient implementations of standard operations on the functions they encode, including computing the EDD representing the pointwise *minimum* of two functions, needed in our reachability algorithm [9].

3 Bounded reachability checking using decision diagrams

Given a model $\mathcal{M}$ and a property ϕ, a generic breadth-first bounded reachability checking algorithm starts with some initial guess for the bound B, computes the set of states $\mathcal{S}^B$ 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 except that, if $\mathcal{S}^B$ does not change between iterations, the *entire* state space has been explored, so one can stop and declare ϕ to hold.

MDDs with BFS-style event-locality-based chaining. Before presenting our main contribution, an algorithm for bounded reachability checking based on Saturation and EDDs, we first show how the above algorithm can be improved when dealing with MDD-encoded state spaces of event-based asynchronous systems, using ideas from *event locality* and *forward chaining*. This serves as one of the reference algorithms in our experimental studies of Sec. 4. The improved algorithm is displayed 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 to significantly reduce the peak number of MDD nodes [8].

Chaining [28] compounds the effect of multiple events within a single iteration. For example, if (i) the set of 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 events (α, β, γ), and (iii) we happen 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 guarantee greater efficiency, as the MDD for $\mathcal{X}^B$ can be much larger than that for $\mathcal{S}^B$; however, it has been shown experimentally that chaining often reduces both time and memory requirements [28].

It is well known that the variable order for the MDD representation is essential. Furthermore, in our setting, the variable order affects also the value of *Top* and *Bot* and the event order. In this paper we employ the heuristics reported in [27] to automatically generate a good order.

Bounded Saturation using EDDs. In several studies, Saturation has been shown superior to BFS-style iterations when symbolically computing least fixpoints for asynchronous models [8,9,10]. The challenge in adapting Saturation to bounded model checking lies in the need to bound the symbolic traversal in the nested fixpoint computations. We propose to use EDDs to encode both the bounded state space and the distance information in the same symbolic encoding. Thus, we bound the symbolic traversal during the EDD symbolic operations by using the distance information, instead of simply limiting the number of outermost iterations performed in a traditional BFS-style approach with or without chaining. Instead of EDDs, we could have used ADDs, but this can result in a performance penalty, as reported in Sec. 4.

Fig. 4 shows two EDD approaches differing in how they bound the symbolic traversal. They are obtained by replacing the *Truncate* call in procedure *BoundedImage* 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 the transition relations are stored using MDDs, with **0** and **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 *BoundedSaturation*). Then, procedure *BoundedSaturate* 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 encodes a set of (sub-)states and distance information consisting of variables at level l or below. When calling procedure *BoundedSaturate* 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 fixpoint computation of *BoundedSaturate* on node p at level l, each event in $\mathcal{E}_l$ is exhaustively fired to perform bounded forward traversal, until no more new reachable (sub-)states are found. *BoundedImage* performs bounded forward traversal by first computing the forward image, followed by either an exact truncation to prune all the (sub-)states exceeding bound B (procedure *TruncateExact*), or a faster but approximate truncation to prune only (sub-)states for which the edge value in the current EDD node would exceed B (procedure *TruncateApprox*). Procedures *BoundedSaturate* and *BoundedImage* are mutually recursive: *BoundedImage* performs a bounded forward traversal of the reachable state space, while all the newly created nodes in the new image are saturated by *BoundedSaturate* (line 7 in procedure *BoundedImage*). Procedure *Minimum* computes the pointwise minimum of the functions encoded by its two argument EDDs, i.e., computing the union of the state sets encoded by the arguments. Finally, procedure *Normalize* takes a node p and ensures that it has at least one outgoing edge with value 0, returning 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. *BoundedSaturate* fires α by calling *BoundedImage* (line 4), which returns the root of the image, so that the "dan-

```
void BoundedSaturation()
 1 r* ← root of the EDD encoding f(i) = 0 if i ∈ S^init, and f(i) = ∞ otherwise
 2 for l = 1 to K do foreach node p at level l do BoundedSaturate(p);

node BoundedSaturate(node p)
 1 l ← p.lvl;
 2 repeat
 3    choose α ∈ E_l, i ∈ S_l, j ∈ S_l s.t. p[i].val < B and D_α[i][j] ≠ 0;
 4    ⟨v,q⟩ ← BoundedImage(p[i], D_α[i][j]);
 5    p[j] ← Minimum(p[j], Truncate(v+1, q));          •exact or approximate
 6 until p does not change;
 7 return p;

edge BoundedImage(edge ⟨v,q⟩, mdd f )
 1 if f = 0 then return ⟨∞,⊥⟩; if f = 1 or q = ⊥ then return ⟨v,q⟩;
 2 k ← q.lvl;                          •given our quasi-reduced form, f.lvl = k as well
 3 s ← NewNode(k);          •create EDD node at level k with edges set to ⟨∞,⊥⟩
 4 foreach i ∈ S_k, j ∈ S_k s.t. q[i].val ≤ B and f[i][j] ≠ 0 do
 5    ⟨w,o⟩ ← Truncate(BoundedImage(q[i], f[i][j]));      •exact or approximate
 6    s[j] ← Minimum(s[j], ⟨w,o⟩);
 7 s ← BoundedSaturate(s);
 8 ⟨γ,s⟩ ← Normalize(s);
 9 return ⟨γ+v,s⟩;

edge Minimum(edge ⟨v,p⟩, edge ⟨w,q⟩)
 1 if v = ∞ then return ⟨w,q⟩; if w = ∞ then return ⟨v,p⟩;
 2 k ← p.lvl;                          •given our quasi-reduced form, q.lvl = k as well
 3 if k = 0 then return ⟨min{v,w},⊥⟩;          •the only node at level k = 0 is ⊥
 4 s ← NewNode(k);          •create EDD node at level k with edges set to ⟨∞,⊥⟩
 5 γ ← min{v,w};
 6 foreach i ∈ S_k do
 7    s[i] ← Minimum(⟨v−γ+p[i].val,p[i].node⟩, ⟨w−γ+q[i].val,q[i].node⟩);
 8 return ⟨γ,s⟩;

edge Normalize(node p)
 1 v ← min{p[i].val : i ∈ S_p.lvl};
 2 foreach i ∈ S_p.lvl do p[i].val ← p[i].val − v;
 3 return ⟨v,p⟩;

edge TruncateExact(edge ⟨v,p⟩)
 1 if v > bound then return ⟨∞,⊥⟩;
 2 foreach i ∈ S_p.lvl do p[i] ← TruncateExact(⟨v+p[i].val,p[i].node⟩);
 3 return ⟨v,p⟩;

edge TruncateApprox(edge ⟨v,p⟩)
 1 if v > bound then return ⟨∞,⊥⟩; else return ⟨v,p⟩;
```

Fig. 4: Bounded Saturation for state-space exploration using EDDs.

gling" edge value must be incremented by 1 to account for the firing of α (line 5). The first portion of procedure *BoundedImage* (lines 1–6) performs the symbolic image computation of the same event α fired by *BoundedSaturate*, and the distance of the new image is incremented by the distance of the "from" states at

the return statement (line 9). 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 (line 7). *BoundedSaturate* uses the test $p[i].value < B$ (line 3), but *BoundedImage* 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.

Comparing with BFS-style MDD approaches, our new proposed EDD approaches use Saturation, a more advanced iteration order, but at the cost of a more expensive symbolic data structure, EDDs (or ADDs). The experimental results of Sec. 4 show that this tradeoff is effective in both time and memory, as the new algorithms often outperform the BFS approach in our benchmarks.

Running example of the EDD approach. Fig. 5 shows the execution of bounded Saturation using *TruncateApprox* as the truncation procedure, on the running example of Fig. 1 with bound $B = 1$. In Fig. 5, 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 g, 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. For readability, edges with value ∞ are omitted. We denote the nodes of the EDD encoding the state space with capital letters (A to E), two specific MDD nodes in the transition relation encoding with f and h, and color a node black once it is saturated. The algorithm starts by saturating nodes A and B, which are saturated immediately since no events are enabled in them (Snapshot (c)). Nodes E, D, and C are saturated in that order. The procedure stops when the root C becomes saturated. Not all procedure calls are shown, e.g., $BoundedImage(C[1], \mathcal{D}_s[1][2])$ is called in Snapshot (f) before node C becomes saturated, but it is not shown since this call does not generate new nodes (states).

Bounded Saturation using ADDs. A version of Saturation using ADDs can be obtained by extending the MDD-based Saturation algorithm of [10], so that it uses an ADD to store the states and their distances, instead of a simple MDD. The ADD has $B + 1$ terminal nodes corresponding to the distances of interest, $\{0, 1, \ldots, B, \infty\}$. We omit this algorithm's details due to space limitations.

4 Experimental Results

We implemented Bounded Saturation in the verification tool SMART, which supports Petri nets as front-end. This section reports our experimental results on a suite of asynchronous Petri net benchmarks when checking for deadlock-freedom as an example of bounded reachability checking. For our symbolic algorithms, this check simply requires us to remove the set of states enabling α, i.e., $Img^{-1}(\widehat{\mathcal{S}}, \mathcal{D}_\alpha)$, for each event α, from the final bounded state space. We compare the performance of several decision-diagram-based methods and the SAT-based methods of Heljanko et al. [18,19] and Ogata et al. [25].

We conduct our experiments on a 3Ghz Pentium machine with 1GB RAM. Benchmarks *byzagr4, mmgt, dac, hs(hartstone), sentest, speed, dp, q, elevator,*

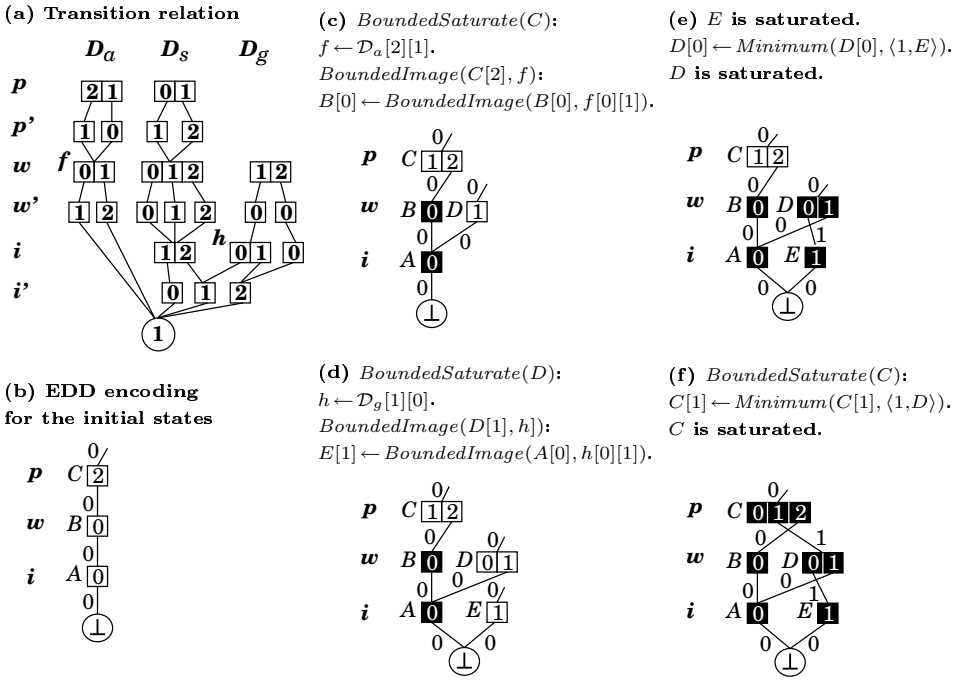

Fig. 5: Bounded Saturation applied to our running example.

key are taken from Corbett [15], and were translated into safe Petri nets by Heljanko [17]. Benchmarks *fms* and *kanban* are deadlocked versions of non-safe Petri net manufacturing system models in the SMART distribution, 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, we can instead use MDDs and EDDs, and achieve time and memory savings. In the following, we thus present the multi-valued version of our algorithms and consider only one EVBDD-based approach (EVBDD-Approx), applied to safe Petri net models, for comparison. The MDD- and EDD-based approaches apply the merging heuristic to the safe nets of Corbett's benchmarks, while they use the non-safe Petri nets *fms* and *kanban* as-is. Variable orders for our experiments are automatically obtained using the heuristic in [27].

Result table. The first three columns of Table 1 show the model name and parameters, and the number of places (#P) and events (#E). The other columns are either "approximate" methods that use a difference definition of distance:

- ***MDD-Chain*** (BFS-style event-locality-based chaining technique of Fig. 3)
- ***SAT-S*** (circuit SAT-based method with step semantics [19])
- ***SAT-C*** (CNF SAT-based method with forward chaining [25])

Or compute a superset of the states $\mathcal{S}^B$ within distance B:

- ***EDD-Approx*/*EVBDD-Approx*** (Bounded Saturation: *TruncateApprox*)

Or "exact" methods that limit their search to exactly $\mathcal{S}^B$:

- ***SAT-I*** (circuit SAT-based method with interleaving semantics [19])
- ***EDD-Exact*** / ***ADD-Exact*** (Bounded Saturation: *TruncateExact*).

For each approximate method, we report the smallest bound B at which either a deadlock is found or the runtime exceeds 10 minutes. For the exact methods, report the exact distance bound B of the deadlock, except for the cases 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 reported in the table, while for the SAT-based tools, only the runtime is available and reported.

Corbett's benchmarks and the SAT-I and SAT-C tools are from [18]. In our experiments, SAT-S performs at least as well as the analogous approach using process semantics [17] (this is also confirmed by the results in Heljanko and Junttila's recent tutorial [18]), therefore we report only the former in Table 1. With Corbett's benchmarks, we show different bounds for SAT-C than those reported in [25]; this is due to using a different initial state, the same as the one considered in [18]. For SAT-I and SAT-C both the encoding time and the `bczchaff` circuit SAT-solver runtime are reported in Table 1. For a fair comparison, the runtime of SAT-C includes the preprocessing steps for scheduling events, encoding the safe Petri net into a boolean formula and then into a CNF formula, and querying the `zchaff` SAT-solver for deadlocks.

Discussion. From Table 1 we can roughly classify benchmarks *byzagr, hs, sentest, fms, kanban* as models with "deep" deadlocks, where the smallest bounds to detect deadlocks range from 30 to 500, and classify all the other benchmarks as models with "shallow" deadlocks, where the smallest bounds are less than 30. For benchmarks with "deep" deadlocks, the newly proposed EDD-Approx method achieves the best performance. For models with "shallow" deadlocks, it seems almost all the methods perform reasonably well, including our MDD-Chain method. Comparing EDD-Approx with EVBDD-Approx, we observe that the former always performs better than the latter. The comparison between EDD-Exact and ADD-Exact shows that they can complement each other. EDD-Approx is arguably the method with the best overall performance, except for the elevator model, where it performs much 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, and the elevator model could be a case where doing so is not beneficial to the structure of the EDD. We also suspect that our variable order heuristic does not perform well on this model.

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

			Approximate distance methods													Exact distance methods					
			EDD-Approx			**EVBDD-Approx**			**MDD-Chain**			**SAT-S**		**SAT-C**			**SAT-I**	**EDD-Exact**		**ADD-Exact**	
Model	$\#P$	$\#E$	B	Time	Mem	B	Time	Mem	B	Time	Mem	B	Time	B	Time	B	Time	Time	Mem	Time	Mem
byzagr4(2a)	579	473	49	2.23	2.41	49	9.14	3.43	6	7.3	9.24	8	0.79	2	2.07	?	>600	>600	–	>600	–
mmgt(3)	122	172	9	0.11	0.2	8	1.28	0.34	5	0.07	0.16	7	0.09	3	1.04	10	1.37	0.32	0.55	0.41	0.33
mmgt(4)	158	232	17	1.22	1.15	17	2.15	1.67	3	0.11	0.2	8	0.23	4	5.52	20	1.24	4.36	3.12	12.87	3.61
dac(15)	105	73	4	0.01	0.0	4	0.03	0.01	2	0.01	0.01	3	0.01	2	0.04	20	0.01	0.03	0.05	0.06	0.04
hs(75)	302	152	151	0.01	0.03	151	0.36	0.05	93	0.08	0.53	151	5.84	1	0.07	151	7.94	0.15	0.03	0.13	0.34
hs(100)	402	202	201	0.03	0.04	201	0.78	0.07	116	0.14	0.78	201	14.85	1	0.13	201	20.31	0.3	0.04	0.23	0.58
sentest(75)	252	102	45	0.0	0.02	45	0.21	0.03	32	0.03	0.21	83	4.27	3	0.13	88	8.51	0.06	0.02	0.08	0.14
sentest(100)	327	127	61	0.01	0.03	61	0.34	0.05	73	0.07	0.47	108	10.71	4	0.29	113	21.85	0.12	0.03	0.22	0.25
speed(1)	29	31	4	0.01	0.02	2	0.24	0.01	3	0.01	0.04	4	0.01	2	0.03	7	0.02	0.02	0.04	0.02	0.01
dp(12)	72	48	2	0.01	0.02	2	0.02	0.03	1	0.0	0.01	1	0.0	1	0.02	12	0.06	0.96	1.77	0.33	0.12
q(1)	163	194	9	0.01	0.03	8	1.45	0.04	7	0.06	0.14	9	0.13	1	0.07	21	0.83	0.08	0.15	0.19	0.13
elevator(3)	326	782	8	15.07	9.46	7	28.5	9.83	6	0.87	0.58	8	0.42	2	3.77	20	2.74	>600	–	7.54	1.83
key(2)	94	92	13	0.06	0.14	18	0.16	0.19	14	0.07	0.2	36	2.88	2	0.05	50	>600	0.15	0.2	0.22	0.34
key(3)	129	133	17	0.2	0.48	17	0.55	0.71	14	0.21	0.52	37	4.39	2	0.10	50	>600	0.62	0.67	2.8	1.64
key(4)	164	174	17	0.69	1.48	15	2.4	1.39	17	0.67	1.54	38	4.21	2	0.18	50	>600	2.02	2.11	9.71	3.15
key(5)	199	215	17	2.04	4.15	17	5.97	6.66	15	1.73	3.37	39	8.07	2	0.25	50	>600	16.87	10.52	33.65	10.03
fms(3)	22	16	9	0.06	0.02	5	0.74	0.02	7	0.01	0.08	10	0.75	3	1.25	30	>600	0.07	0.06	0.05	0.14
fms(7)	22	16	19	0.07	0.26	11	4.4	0.69	15	0.24	2.58	18	>600	6	>600	70	>600	0.8	2.2	1.12	4.7
fms(10)	22	16	28	0.12	0.99	6	>600	–	21	1.35	14.75	16	>600	7	>600	100	>600	5.37	14.37	5.24	24.11
kanban(1)	17	16	28	0.04	0.0	27	0.33	0.01	13	0.0	0.01	19	0.05	5	0.09	40	16.56	0.08	0.0	0.01	0.01
kanban(3)	17	16	82	0.05	0.06	79	5.34	0.34	19	0.03	0.23	12	>600	3	>600	120	>600	0.1	0.07	0.27	0.64
kanban(10)	17	16	271	0.84	10.43	1	>600	–	54	2.83	29.29	1	>600	1	>600	400	>600	14.4	10.46	51.76	187.9

Table 1: Experimental results (Time in sec, Mem in MB). ">600" indicates that runtime exceeds 600 sec or memory exceeds 1GB.

5 Discussion and Related Work

SAT-solving for Petri nets. We first add some details to the two SAT-based approaches to deadlock checking of safe Petri nets [17,25], against which we compared ourselves in the previous section regarding run-time efficiency.

Heljanko's work [17] establishes 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 for safe Petri nets, i.e., finite nets, as otherwise these semantics may not coincide. In contrast, our technique is applicable to general Petri nets, even if they exhibit an infinite state space.

Ogata, Tsuchiya, and Kikuno's approach [25] 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 solving 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 a 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 is now included in the previous section, and shows that neither method is superior in all cases.

BDD vs. SAT on synchronous systems. As mentioned before, the common belief that SAT-based model checking outperforms decision-diagram-based model checking was proved wrong by Cabodi, Nocco, and Quer [5] for a class of digital circuits that largely exhibits 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 for 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* [7] on which our *Bounded Saturation* algorithm is based. Similar to the algorithm proposed in [5], 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 approach and up to $E \cdot B$ in [5], where K and E are the number of components and events in the studied Petri net, respectively.

Together, the results of Cabodi et al. and ours, as well as further recent research [29], 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 were obtained with static variable orders which have been computed using a simple heuristic [27]. Thus, no fine-tuning of models by hand was necessary, which was criticized in [14].

Petri net unfoldings. Both SAT-based and decision-diagram-based techniques are established techniques for addressing the state-space explosion problem. The Petri net community has developed another successful technique to address this problem, first suggested in a seminal paper by McMillan [23]. 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 often limited to finite-state Petri nets, although recent work suggests an extension to some infinite-state systems [1].

6 Conclusions and Future Work

This paper explored the utility of decision diagrams for bounded reachability checking of asynchronous systems. To this end, we reconsidered *Saturation*, a state-space generation algorithm which is based on Multi-way Decision Diagrams (MDDs) and exploits the event locality and interleaving semantics inherent in asynchronous systems. As the search strategy in Saturation is unlike breadth-first search, bounding the search required us to employ *Edge-Valued MDDs*, which allow for storing states together with their distances from the initial states.

Our extensive experimental analysis of the resulting *Bounded Saturation* algorithm showed that it often compares favorably to the competing SAT-based approaches introduced in [17,18,25]. 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 bounded reachability checking of asynchronous systems, and the widespread perception that decision diagrams are not suited for bounded model checking [14] is untrue.

Future work should investigate whether an efficient version of Bounded Saturation can be developed using standard decision diagrams, rather than decision diagrams with explicit distance counters built in. We also intend to investigate whether the event locality inherent in asynchronous systems can be exploited in SAT-based reachability checking.

Acknowledgements We would like to thank K. Heljanko, T. Jussila, and T. Tsuchiya for providing their inputs and software tools used in our study.

References

1. P. Abdulla, S. Iyer, and A. Nylén. SAT-solving the coverability problem for Petri nets. *FMSD*, 24(1):25–43, 2004.
2. R. I. Bahar, et. al. Algebraic decision diagrams and their applications. *FMSD*, 10(2/3):171–206, 1997.
3. A. Biere, A. Cimatti, E. Clarke, Y. Zhu. Symbolic model checking without BDDs. *TACAS*, LNCS 1579, pp. 193–207, 1999. Springer.

4. J. R. Burch, E. M. Clarke, D. E. Long. Symbolic model checking with partitioned transition relations. *VLSI*, pp. 49–58, 1991.
5. G. Cabodi, S. Nocco, S. Quer. Are BDDs still alive within sequential verification? *STTT*, 7(2):129–142, 2005.
6. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578–608, 2006.
7. G. Ciardo, G. Lüttgen, R. Siminiceanu. Saturation: an efficient iteration strategy for symbolic state-space generation. *TACAS*, LNCS 2031, pp. 328–342, 2001. Springer.
8. G. Ciardo, R. Marmorstein, R. Siminiceanu. The saturation algorithm for symbolic state space exploration. *STTT*, 8(1):4–25, 2006.
9. G. Ciardo, R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. *FMCAD*, LNCS 2517, pp. 256–273, 2002. Springer.
10. G. Ciardo, A. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. *CHARME*, LNCS 3725, pp. 146–161, 2005. Springer
11. A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri. NuSMV: A new symbolic model verifier. *CAV*, LNCS 1633, pp. 495–499, 1999. Springer.
12. E. Clarke, A. Biere, R. Raimi, Y. Zhu. Bounded model checking using satisfiability solving. *FMSD*, 19(1):7–34, 2001.
13. E. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT, 1999.
14. F. Copty, et. al. Benefits of bounded model checking at an industrial setting. *CAV*, LNCS 2102, pp. 436–453, 2001. Springer.
15. J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–180, 1996.
16. The VIS Group. VIS: A system for verification and synthesis. *CAV*, LNCS 1102, pp. 428–432, 1996. Springer.
17. K. Heljanko. Bounded reachability checking with process semantics. *CONCUR*, LNCS 2154, pp. 218–232, 2001. Springer.
18. K. Heljanko, T. Junttila. Advanced tutorial on bounded model checking, *ACSD/ICATPN*, 2006. http://www.tcs.hut.fi/∼kepa/bmc-tutorial.html.
19. K. Heljanko, I. Niemelä. Answer set programming and bounded model checking. *Answer Set Programming*, 2001.
20. F. Ivančić, Z. Yang, M. Ganai, A. Gupta, P. Ashar. F-Soft: Software Verification Platform. *CAV*, LNCS 3576, 2005. Springer.
21. 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.
22. Y.-T. Lai, S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. *DAC*, pp. 608–613, 1992. IEEE Press.
23. K. McMillan. A technique of state space search based on unfolding. *FMSD*, 6(1):45–65, 1995.
24. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an efficient SAT solver. *DAC*, pp. 530–535, 2001. ACM Press.
25. S. Ogata, T. Tsuchiya, T. Kikuno. SAT-based verification of safe Petri nets. *ATVA*, LNCS 3299, pp. 79–92, 2004. Springer.
26. I. Rabinovitz, O. Grumberg. Bounded model checking of concurrent programs. *CAV*, LNCS 3576, pp. 82–97, 2005. Springer.
27. R. Siminiceanu, G. Ciardo. New metrics for static variable ordering in decision diagrams. *TACAS*, LNCS 3920, pp. 90–104, 2006. Springer.
28. M. Solé, E. Pastor. Traversal techniques for concurrent systems. *FMCAD*, LNCS 2517, pp. 220–237, 2002. Springer.
29. R. Tzoref, M. Matusevich, E. Berger, I. Beer. An optimized symbolic bounded model checking engine. *CHARME*, LNCS 2860, pp. 141–149, 2003. Springer.