A decision diagram operation for reachability

Brand, S.O.; Bäck, T.H.W.; Laarman, A.W.; Chechik, M.; Katoen, J.P.; Leucker, M.

Citation

Brand, S. O., Bäck, T. H. W., & Laarman, A. W. (2023). A decision diagram operation for reachability. *Lecture Notes In Computer Science*, 514-532. doi:10.1007/978-3-031-27481-7_29

Version:	Publisher's Version
License:	Licensed under Article 25fa Copyright Act/Law (Amendment Taverne)
Downloaded from:	https://hdl.handle.net/1887/3715032

Note: To cite this publication please use the final published version (if applicable).

A Decision Diagram Operation for Reachability

Sebastiaan Brand(✉), Thomas Bäck, and Alfons Laarman

Leiden Institute of Advanced Computer Science,
Leiden University, Leiden, The Netherlands
{s.o.brand,t.h.w.baeck,
a.w.laarman}@liacs.leidenuniv.nl

Abstract. Saturation is considered the state-of-the-art method for computing fixpoints with decision diagrams. We present a relatively simple decision diagram operation called REACH that also computes fixpoints. In contrast to saturation, it does not require a partitioning of the transition relation. We give sequential algorithms implementing the new operation for both binary and multi-valued decision diagrams, and moreover provide parallel counterparts. We implement these algorithms and experimentally compare their performance against saturation on 692 model checking benchmarks in different languages. The results show that the REACH operation often outperforms saturation, especially on transition relations with low locality. In a comparison between parallelized versions of REACH and saturation we find that REACH obtains comparable speedups up to 16 cores, although falls behind saturation at 64 cores. Finally, in a comparison with the state-of-the-art model checking tool ITS-tools we find that REACH outperforms ITS-tools on 29% of models, suggesting that REACH can be useful as a complementary method in an ensemble tool.

Keywords: Model checking · Reachability · Saturation · Decision diagrams · BDDs · MDDs

1 Introduction

Reachability Analysis. Model checking is an important technique for ensuring that systems work according to specification. A core task in model checking is reachability analysis [8,18], i.e., computing forward or backward reachable states of a system. Typically, the state space of a program grows exponentially with the number of variables and threads. One method for dealing with this explosion is the use of symbolic methods such as decision diagrams. Decision diagrams [12] are directed, acyclic graphs that succinctly represent sets of states by leveraging the exponential growth in paths from a dedicated root node to a leaf. The data structure provides various efficient manipulation operations, such as logical disjunction and conjunction and image computation.

In this work, we present a new decision diagram operation for reachability.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
M. Chechik et al. (Eds.): FM 2023, LNCS 14000, pp. 514–532, 2023.
https://doi.org/10.1007/978-3-031-27481-7_29

Related Work. While SAT-based methods for model checking have become increasingly popular, doing reachability analysis with decision diagrams is still an important component of many state-of-the-art model checking tools, as can be seen in the Model Checking Contest (MCC) [30,40]. Symbolic reachability analysis with binary decision diagrams (BDDs) and other variants [6,20,27,41] is done by encoding both the initial system state S_{init} and its transition relation R in the diagram. The set of reachable states S is then iteratively computed using the image operation [33], denoted by $S.R$, starting from S_{init}. Since the order of exploration (e.g. breadth-first search, depth-first search, or other strategies) greatly influences the sizes of the intermediate decision diagrams, various exploration strategies, like saturation [16], chaining [36], and sweep-line [15], have been considered. These algorithms have in common the use of the image computation $S.R$ as their main operation.

Saturation stands out from other approaches, not only because it often performs better [17] and leading MCC tools use it in decision diagram based reachability [40], but also because it integrates the image computations into the traversal of the decision diagram of S. Saturation avoids redundant reconstructions by building the decision diagram for the reachable states bottom-up, eagerly *saturating* the bottom nodes by exhaustively applying all relevant transitions.

Contribution. We present three new decision diagram operations for reachability: ReachBdd and ReachMdd (for BDDs and MDDs respectively), as well as a parallel version of ReachBdd. These algorithms partially construct the decision diagram of S from the bottom up, but unlike saturation do not require partial relations and can handle monolithic transition relations. An additional advantage of these new reachability operations is their relative simplicity in comparison with saturation.

We implement these new operations in the decision diagram package Sylvan [21], and experimentally compare them against saturation on a total of 692 problem instances from three model checking benchmark sets: DVE (BEEM [35]), Petri nets (MCC [31]), and Promela models [7,26]. We find that our methods are competitive with saturation, and tend to outperform saturation on larger instances. The parallel speedups obtained by ReachBdd up to 16 cores are comparable to those achieved in a parallel version of saturation [22], although they fall behind on 64 cores. Aside from the comparison against saturation, we also compare ReachMdd against the state-of-the-art model checking tool ITS-tools [40], where we find that ReachMdd performs better than ITS-tools on 29% of models, and can therefore be useful as a complementary method in an ensemble tool.

Outline. Section 2 discusses preliminaries. Section 3 explains the new reachability algorithms, and is then followed by an empirical evaluation of these algorithms in Sect. 4. Finally, Sect. 5 concludes this work.

2 Preliminaries

2.1 Binary Decision Diagrams

Binary decision diagrams (BDDs) [12,38] are a data structure for representing Boolean functions $f(x_1, \dots, x_n)$, i.e., functions of type $f : \{0,1\}^n \to \{0,1\}$. Structurally, a BDD is a rooted, directed, acyclic graph with two types of nodes: terminal nodes with values $\{0,1\}$ and non-terminal nodes v that have two children, $v[0]$ (*low*) and $v[1]$ (*high*), and a variable label $\mathsf{var}(v) \in \{1, \dots, n\} = [n]$, indexing into $\{x_i\}_{i\in[n]}$. Figure 1 shows examples of *ordered* BDDs, i.e., BDDs where on each path variable labels occur in a fixed order $x_1 < x_2 < \cdots < x_n$.

A non-terminal node v with $\mathsf{var}(v) = i$, shown right, can be read as the *Shannon decomposition "if* $x_i = 1$ *then* $v[1]$ *else* $v[0]$*."* The function represented by the node v, call it f^v, is thus given by

$$f^v(x_1, \dots, x_n) \triangleq \begin{cases} x_i f^{v[1]} \vee \overline{x_i} f^{v[0]} & \text{if } v \notin \{0,1\}\,, \\ v & \text{if } v \in \{0,1\}\,. \end{cases} \tag{1}$$

The definition in Eq. 1 shows that a conditioned subfunction $f^v_{|\vec{a}}$, as in Eq. 2 below, is represented by a decision diagram node, namely the one following the path $v[\vec{a}] \triangleq v[a_1][a_2]\dots[a_k]$, assuming the BDD is ordered and no variables are skipped.

$$f^v_{|a_1,\dots,a_k}(x_1, \dots, x_n) \triangleq f^v(x_1 = a_1, \dots, x_k = a_k, x_{k+1}, \dots, x_n) \tag{2}$$

The insight that BDDs exploit to realize succinct representations of commonly-encountered Boolean functions is that many subfunctions for different $\vec{a}, \vec{b} \in \{0,1\}^*$ can be isotropic, i.e., $f_{|\vec{a}} = f_{|\vec{b}}$. Take $f_{|00} = f_{|11}$ in Fig. 1a. In the diagram, this means that the isomorphic subgraphs below the nodes representing $f_{|00}$ and $f_{|11}$ can be merged, as in Fig. 1c.

An ordered BDD is *reduced* when, in addition to isomorphic sub-graph merging, all *redundant nodes* (nodes v with $v[0] = v[1]$) are removed. Take

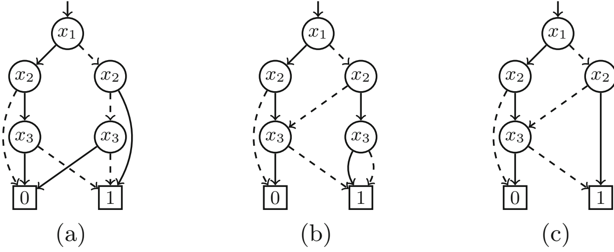


Fig. 1. (Ordered) BDDs representing function $f = (x_1 \wedge x_2 \wedge \overline{x_3}) \vee (\overline{x_1} \wedge \overline{x_2} \wedge \overline{x_3}) \vee (\overline{x_1} \wedge x_2)$. For node v, we draw $\mathsf{var}(v) = i$ as x_i. Dashed lines lines represent low branches ($v[0]$) and solid lines high branches ($v[1]$). Only the BDD in (c) is completely reduced. The BDD in (a) can be reduced to (c) by merging the two isomorphic nodes for x_3, while the BDD in (b) can be reduced to (c) by removing the (right-most) redundant node x_3.

$f_{|010} = f_{|011} = f_{|01} = 1$, in Fig. 1c. Removed redundant nodes can be reconstructed by recognizing that a variable is skipped on a path, as done Fig. 1b.

Reduced and ordered BDDs (ROBDDs) are canonical representations of Boolean functions, i.e., any two functions with the same truth table are uniquely represented by (the root node of) an ROBDD. Canonicity allows for equivalence checking in constant time through hashing of nodes v as tuples $\langle \mathsf{var}(v), v[1], v[0] \rangle$ (provided that nodes $v[1]$,$v[0]$ are already canonically represented, i.e., the BDD is build in a bottom-up fashion). This in turn allows efficient manipulation operations ($\wedge, \vee, \dots$) as discussed below.

In this text, we fix the variable order $x_1 < x_2 < \dots < x_n$. For conciseness, we will often consider quasi-ROBDD, which are ROBDDs where all redundant nodes are reconstructed (e.g. Fig. 1b). In many settings, this stronger definition does not lose generality, as quasi-ROBDD are also canonical, and at most a factor $\frac{n}{2}$ larger than an ROBDD for the same function [29]. At the same time, because for all quasi-ROBDD nodes v we have $f^v_{\vec{a}} = v[\vec{a}]$, this assumption greatly simplifies algorithm representation and reduces cases in proofs. We denote with *level* i the set of all nodes with the variable label i.

2.2 Multi-valued Decision Diagrams

Multi-valued decision diagrams (MDDs) [27,37] are a generalization of BDDs for encoding functions $\mathcal{D}_1 \times \dots \times \mathcal{D}_n \to \{0, 1\}$, where $\mathcal{D}_i = \{0, 1, \dots, m-1\}$ for some m and all i. Each MDD node with variable x_i has m outgoing edges, each with a label in $\mathcal{D}_i$. The interpretation of following an edge remains the same as for BDDs: for an MDD which encodes a function f and has root node v with $\mathsf{var}(v) = i$, following an edge with label $a \in \mathcal{D}_i$ leads to an MDD which encodes the sub-function $f|_{x_i=a}$.

Similar to BDDs, MDDs are typically reduced by merging isomorphic subgraphs. However, unlike ROBDDs, redundant nodes are usually not removed in MDDs, which means variables are never skipped on any path. So an MDD with $m = 2$ is a Quasi-ROBDD. Figure 2 shows an example.

A list decision diagram (LDD) [9] is the Knuth transform of an MDD into a left-child right-sibling binary tree. Siblings (right-ward chains) are stored as

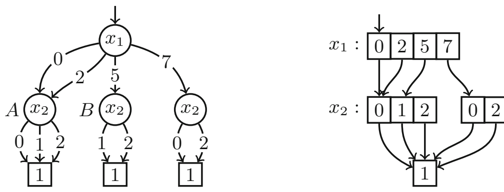


Fig. 2. An MDD (left) and LDD (right) which both encode the set $\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle, \langle 2,0\rangle, \langle 2,1\rangle, \langle 2,2\rangle, \langle 5,1\rangle, \langle 5,2\rangle, \langle 7,0\rangle, \langle 7,2\rangle\}$. Arrays represent right-ward sibling chains in the LDD. To improve legibility, we omit edges pointing to the 0 terminal and replicate the 1 terminal for MDDs.

(ordered) linked lists, which allows the reuse of common sibling suffixes, as shown in the example in Fig. 2 (e.g. $f_{|5}$ reuses a part of the siblings of $f_{|2} = f_{|0}$).

Finally, a BDD or MDD representing a function $f(x_1, \ldots, x_n)$ can also be interpreted as a set of strings $\vec{a}$ of length n, according to the characteristic function $\{\vec{a} \mid f(\vec{a}) = 1\}$. So the BDDs in Fig. 1 all represent $\{000, 010, 011, 110\}$.

2.3 Decision Diagram Operations

What makes BDDs and MDDs so useful, aside from their possible succinctness, is that many manipulation operations, such as disjunction (set union) and conjunction (set intersection), can be performed in polynomial time in the number of decision diagram nodes in the operands [3,12]. While other operations that have been shown to be NP-complete [33], such as unbounded existential quantification and thus also image computation [33], are often still efficient in practice [13].

As an example, the algorithm below computes the union of two quasi-reduced diagrams A and B, i.e., $A \cup B$ (or in the functional interpretation: $A \vee B$). Because any BDD is defined by its root node, the arguments A and B are simply given as nodes. The algorithm first considers leafs as a base case, treating them according to the semantics of $\vee$. On line 7 and 8, the function is called recursively on the children of the input BDDs, synchronizing on the low and high branches. The results from these recursive calls are then combined with the $\textsc{MakeNode}(x, L, H)$ function which creates a *reduced BDD node* v with $v[0] = L$, $v[1] = H$ and $\mathsf{var}(v) = x$. To ensure reduction, it returns L $(= H)$ when the node is redundant and looks up the tuple $\langle x, L, H\rangle$ in a *unique table*, as discussed in Sect. 2.1. For MDDs, we assume a function $\textsc{MakeNode}(x, A_0, \ldots, A_{m-1})$ which returns a (quasi-)reduced MDD node and, for notational convenience, takes $m+1$ positional arguments: a variable x, and one MDD node A_i for each of its m children (some of which can be 0).

Lastly, it is important to realize that a decision diagram with $|V|$ nodes can have $\mathsf{exp}(|V|)$ paths from the root to a terminal node. To achieve polynomial runtimes, decision diagram operations use top-down dynamic programming (see

```
1  def Union(A, B) :          ▷ For quasi-ROBDDs A and B on n variables.
2      if A = 0 then return B
3      if B = 0 then return A
4      if A = 1 ∨ B = 1 then return 1
5      if res ← cache[Union, A, B] then return res
6      x ← var(A)              ▷ By virtue of quasi reduction var(A) = var(B).
7      L ← Union(A[0], B[0])
8      H ← Union(A[1], B[1])
9      res ← MakeNode(x, L, H)
10     cache[Union, A, B] ← res
11     return res
```

line 5, 10). This ensures that different paths leading to the same (pairs of) nodes are caught by the cache, avoiding recomputation.

2.4 Encoding Symbolic Transition Systems

A symbolic transition system is a tuple $(\vec{x}, \mathbb{S}, R, S_{\text{init}})$, where $\vec{x}$ is a tuple of Boolean variables $(x_1, \ldots, x_n)$, $\mathbb{S} = \{0,1\}^n$ is the state space (including unreachable states), $R \subseteq \mathbb{S} \times \mathbb{S}'$ is a transition relation, and $S_{\text{init}} \subseteq \mathbb{S}$ is a set of initial states. The relation R is a constraint over variables $\vec{x}, \vec{x}'$, where $\vec{x}$ encodes the source states and $\vec{x}'$ (consisting of primed copies of $\vec{x}$) encodes target states. While R monolithically encodes the system's behavior, we also consider a local variant, discussed in the example which follows.

As an example we give a transition system which captures the dining philosophers problem. We have k processes (philosophers) and k resources (forks). Each process P_i, with $i \in \{1, \ldots, k\}$, attempts to allocate two resources: a fork i on the left and a fork $j = ((i-1) \bmod k) + 1$ on the right. If a fork is unavailable the philosopher waits until it becomes available. To describe the state space we use $3k$ Boolean variables: $\vec{x} = (a_1, l_1, r_1, \ldots, a_k, l_k, r_k)$, where a_i ($\overline{a_i}$) indicates fork i is (not) available, l_i ($\overline{l_i}$) indicates philosopher i does (not) hold a fork in their left hand, and r_i ($\overline{r_i}$) idem for their right hand. The starting state $S_{\text{init}} = (a_1, \overline{l_1}, \overline{r_1}, \ldots, a_k, \overline{l_k}, \overline{r_k}.)$

To define R_i for $k > 1$ processes, we define three local relations R_i^m that implement picking up and putting down the left/right fork, and eating.

$$R_i^1 = (a_i \oplus a_i') \wedge (l_i \oplus l_i') \qquad \triangleright \texttt{ pick up / put down left}$$
$$R_i^2 = (r_i \oplus r_i') \wedge (a_j \oplus a_j') \qquad \triangleright \texttt{ pick up / put down right}$$
$$R_i^3 = l_i \wedge l_i' \wedge r_i \wedge r_i' \qquad \triangleright \texttt{ eat (hold on to both forks)}$$

Let $\mathsf{support}(R_i^m)$ be the set of (primed and unprimed) variables in R_i^m. Notice that the support of each local relation contains a different subset of the target variables $\vec{x}'$. To ensure that the other target variables are not left unconstrained, we need to add the constraint $x \Leftrightarrow x'$ for all missing variables $x' \in \vec{x}'$. For instance, pick up / put down left should be extended as:

$$(a_i \oplus a_i') \wedge (l_i \oplus l_i') \wedge (r_i \Leftrightarrow r_i') \wedge \bigwedge_{j \neq i} (a_j \Leftrightarrow a_j' \wedge r_j \Leftrightarrow r_j' \wedge l_j \Leftrightarrow l_j')$$

The same needs to happen when merging multiple processes R_i into a single global relation R, i.e. $R = \bigvee_{i=1}^{k} \mathcal{R}_i$, where $\mathcal{R}_i \triangleq R_i \wedge \bigwedge_{x' \in \vec{x}' \setminus \mathsf{support}(R_i)} x \Leftrightarrow x'$. Section 2.5 discusses how extending local relations in this way can be avoided.

If a transition relation R is composed of R_i's, where each $\mathsf{support}(R_i)$ is a small subset of $\vec{x}, \vec{x}'$, we say that R has high locality. If R cannot be split up into such partial relations we say that R has low locality, or is *monolithic*.

2.5 Reachability with Decision Diagrams

For a transition system with n Boolean variables $\{x_1, \ldots, x_n\}$, a single state is given by a bit string $s \in \{0,1\}^n$. A set of states $S \subseteq \{0,1\}^n$ can be encoded in a BDD. Likewise, a transition relation $R \subseteq \{0,1\}^n \times \{0,1\}^n$ can be encoded in a BDD with $2n$ variables. The variables are ordered by interleaving the source state variables $\vec{x} = (x_1, \ldots, x_n)$ with target state variables (primed copies: $x'_1, \ldots, x'_n$), i.e., $(x_1, x'_1, \ldots, x_n, x'_n)$, as it is both convenient for the implementation of BDD algorithms, and it reduces the diagram size. To compute the successors to an initial set of states S, we can then use the decision diagram operation $\textsc{Image}(S, R)$ [21,33]:

$$\textsc{Image}(S, R) = (\exists x_1, \ldots, x_n : (S \wedge R))_{[x'_1,\ldots,x'_n \,:=\, x_1,\ldots,x_n]}, \tag{3}$$

where $[x' := x]$ indicates the relabeling of the target variables to source variables. The $\textsc{Image}$ operation can also be implemented for partial relations R_i, with the benefit that existential quantification only needs to happen over $\mathsf{support}(R_i)$, rather than over all variables. The union of the image under each R_i separately equals the image under the global transition relation:

$$\exists \vec{x} : ((\mathcal{R}_1 \vee \cdots \vee \mathcal{R}_k) \wedge S)_{[\vec{x}' \,:=\, \vec{x}]} = (\exists \vec{x}'_1 : R_1 \wedge S)_{[\vec{x}' \,:=\, \vec{x}]} \vee \cdots \vee (\exists \vec{x}'_k : R_k \wedge S)_{[\vec{x}' \,:=\, \vec{x}]}$$

where $\vec{x}'_i = \mathsf{support}(R_i) \cap \vec{x}'$ and $\mathcal{R}_i$ is the extension of R_i as defined in Sect. 2.4.

With $\textsc{Image}$, the set of reachable states can be computed by repeatedly applying R to a growing set of reachable states until no new states are found, which we denote with $S.R^*$.

Finally, decision diagrams often represent relations more succinctly when source variables $\vec{x}$ are interleaved with target variables $\vec{x}'$ in the order [33].

2.6 Saturation

Saturation [16] is a method for computing reachability that exploits locality of transitions. Aside from the initial states S_{init}, the algorithm takes as input several local transition relations R_i (see Sect. 2.4), ordered such that $\mathsf{var}(R_i) \geq \mathsf{var}(R_{i+1})$.

To illustrate how saturation works, let us give an example. Say we have a global state space given by $\{x_1, \ldots, x_4\}$, and three partial relations R_i with $\mathsf{support}(R_1) = \{x_3, x'_3, x_4, x'_4\}$, $\mathsf{support}(R_2) = \{x_2, x'_2, x_3, x'_3\}$, and $\mathsf{support}(R_3) = \{x_1, x'_1, x_2, x'_2\}$. The "dependency matrix" on the right visualizes the dependencies of the partial relations on each of the variables. The saturation algorithm traverses the decision diagram of a set of states S and *saturates* the nodes from the bottom-up. What this means in the case of the example is that the partial relation R_1 is applied to all nodes v in S with $\mathsf{var}(v) = 3$, until v has converged. After saturating this node v the algorithm backtracks upwards in the decision diagram to a node w with $\mathsf{var}(w) = 2$, and saturates this node by exhaustively applying R_2, while eagerly saturating new nodes created below w.

$$\begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \begin{array}{c} \begin{array}{ccc} R_1 & R_2 & R_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \end{array}$$

While breadth-first search (BFS) suffers from large intermediate diagram sizes [25], saturation often avoids this by ensuring that the lower levels of the decision diagram reach their final configuration early. Generally this works well if the (average) bandwidth (the distance between the first and the last non-zero entry in each row) of the dependency matrix is low. This occurs for example in asynchronous systems where processes mainly modify local variables or communicate only with "neighboring processes" through channels or shared variables dedicated to neighboring pairs. This is for example the case in the dining philosophers example given in Sect. 2.4. Finding a variable order and organizing the partial relations such that both this bandwidth and sizes of the decision diagrams are minimized is generally hard (even finding an optimal variable order for a single BDD is NP-complete [10]) but good heuristics exist [1,2,4,5,34].

While originally proposed for MDDs, saturation has since been implemented for BDDs as well [22].

3 Decision Diagram Operation for Reachability

3.1 For BDDs

We present an operation ReachBdd(S, R) which computes the reachable states $S.R^*$ for BDDs. As is typical in BDD operations, our algorithm splits the computation into recursive calls on smaller, factored BDDs, after which the results are composed again in the backtracking step. The way we factor R is inspired by [32], where a BDD algorithm is given for computing the closure R^* of R (computing the closure R^* is generally much more expensive [32, §6], hence we want to compute $S.R^*$ directly for a given S). ReachBdd is given in Algorithm 1, and its correctness is discussed in Sect. 3.4.

Algorithm 1: A BDD operation for computing reachability. Cache lookup/insert for dynamic programming after line 5 and 10 are omitted.

```
1  def ReachBdd(S, R) :          ▷ For quasi-ROBDDs S, R on n, 2n variables.
2      if S = 0 then return 0
3      if R = 0 then return S
4      if S = 1 then return 1
5      if R = 1 ∧ S ≠ 0 then return 1
6      while S did not converge do
7          S[0] ← ReachBdd(S[0], R[00])
8          S[1] ← Union(S[1], Image(S[0], R[01]))
9          S[1] ← ReachBdd(S[1], R[11])
10         S[0] ← Union(S[0], Image(S[1], R[10]))
11     return MakeNode(var(S), S[0], S[1])
```

The algorithm recurses on the low and high child of a BDD S. This splits the state space into $\mathbb{S}_{|0}$ (all states starting with a 0) and $\mathbb{S}_{|1}$. The relation R can be split up accordingly as shown in Fig. 3. The self loops in this figure represent

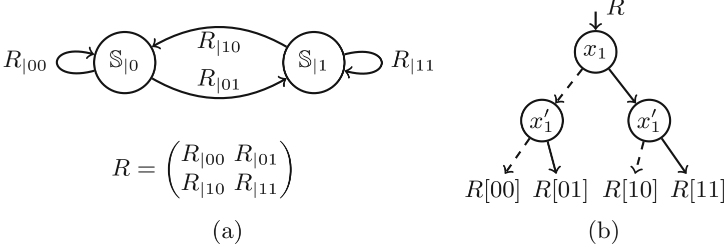


Fig. 3. The state space $\mathbb{S}$ can be split up into states where the first variable equals 0, and states where the first variable equals 1 (a), and the transition relation R is split up accordingly. Because source and target variables are interleaved as usual, these partitions of R can be easily accessed in the BDD structure (b).

$S_{|0}.R^*_{|00}$ and $S_{|1}.R^*_{|11}$, and correspond to recursive calls to ReachBdd (line 7, 9). The results of these calls need to be propagated using image computation $S_{|i}.R_{|ij} = \text{Image}(S[i], R[ij])$ (line 8, 10) until S has converged, so we incorporate a loop (line 6). For notational convenience we assume $S[0]$ and $S[1]$ are program variables to which we can assign new BDDs.

The base cases for the algorithm are as follows: If the set of initial states or the transition relation is empty ($S = 0$ or $R = 0$) there are no successors and the set of reachable states is the set of initial states. If the set of initial states contains all states ($S = 1$), or if R contains transitions from all states to all other states ($R = 1$) and S is not empty, then all states are reachable.

With the decision diagram framework Sylvan [24], we can parallelize decision diagram operations through Spawn/Sync commands, which respectively fork and join light-weight tasks [24]. However, the order of (parallel) operations is something to take into account. In particular, line 7 and 8 from Algorithm 1 are dependent, so cannot be executed in parallel. In order to parallelize ReachBdd, we change the order of calls in this loop and introduce ReachBddPar in Algorithm 2.

Algorithm 2: ReachBddPar parallelizes the loop on line 6-10 in Alg. 1. The Image and Union calls are also parallelized [23].

6 **while** S did not converge **do**
7   Spawn(ReachBddPar($S[0], R[00]$)) ▷ `Spawn call as task (fork)`
8 $S[1] \leftarrow$ ReachBddPar($S[1], R[11]$) ▷ `Call directly`
9 $S[0] \leftarrow$ Sync ▷ `Obtain task result (join)`
10   Spawn(Image($S[1], R[10]$)) ▷ `Spawn call as task (fork)`
11 $T_1 \leftarrow$ Image($S[0], R[01]$) ▷ `Call directly`
12 $T_0 \leftarrow$ Sync ▷ `Obtain task result (join)`
13   Spawn(Union($S[0], T_0$)) ▷ `Spawn call as task (fork)`
14 $S[1] \leftarrow$ Union($S[1], T_1$) ▷ `Call directly`
15 $S[0] \leftarrow$ Sync ▷ `Obtain task result (join)`

3.2 Analysis

To provide some intuition for the complexity behavior of ReachBdd, we provide two cases: one where ReachBdd is exponentially faster than BFS, and one where ReachBdd reduces to BFS.

Ideal Case. We give a concrete instance of a relation R and an initial set of states S_{init} for which ReachBdd performs exponentially better than a simple BFS. Consider a transition relation R which simply increases a (program) counter of n bits. This counts form a starting state $S_{\text{init}} = 0 = \langle 00 \dots 0 \rangle$ in steps of 1 to $2^n - 1 = \langle 11 \dots 1 \rangle$. As the state space is a line graph, the BFS algorithm discovers one new state every iteration, requiring $O(2^n)$ calls to the Image function.

To illustrate the behavior of the ReachBdd algorithm, let us explicitly write all $R_{|ij}$ for $n = 3$:

$$R_{|00} = \left\{ \begin{matrix} (\not{0}00, \not{0}01), \\ (\not{0}01, \not{0}10), \\ (\not{0}10, \not{0}11) \end{matrix} \right\} \quad R_{|11} = \left\{ \begin{matrix} (\not{1}00, \not{1}01), \\ (\not{1}01, \not{1}10), \\ (\not{1}10, \not{1}11) \end{matrix} \right\} \quad R_{|01} = \left\{ (\not{0}11, \not{1}00) \right\} \quad R_{|10} = \emptyset$$

While $R_{|00}$ and $R_{|11}$ represent different sets, the BDDs $R[00]$ and $R[11]$ are equal.

For all non-terminal cases Algorithm 1 does the following:

8	$S[0] \leftarrow$ ReachBdd$(S[0], R[00])$	▷ `computes states` S_0^{all}
9	$S[1] \leftarrow$ Union$(S[1],$ Image$(S[0], R[01]))$	▷ `generates 'seed' state` S_1^{init}
10	$S[1] \leftarrow$ ReachBdd$(S[1], R[11])$	▷ `computes states` S_1^{all}
11	$S[0] \leftarrow$ Union$(S[0],$ Image$(S[1], R[10]))$	▷ `produces no new states`

First, ReachBdd computes all reachable states which start with a 0, let us call these S_0^{all}. Next, the Image call produces exactly one new state, $\langle 100 \dots 0 \rangle$, which will act as a "seed" state for the next ReachBdd call. Since the BDDs of S_0^{init} and S_1^{init} are equal, just as the BDDs $R[00]$ and $R[11]$, the second ReachBdd call can be looked up from cache. Finally, since $R[10] = \emptyset$, no new states will be added to S_0, and both S_0 and S_1 will have converged.

We find two things: first, all the reachable states are found in a single loop iteration, and second, each call to ReachBdd only generates one recursive call to ReachBdd (because the second call can always be looked up from cache). Overall, ReachBdd only makes $O(n)$ recursive calls to itself and to the Image function.

Due the monolithic nature of the transition relation, saturation will behave like BFS in this case.

Bad Case. Here we provide an instance for which ReachBdd reduces to breadth-first search. From an arbitrary transition relation R we create a new relation R' for which ReachBdd behaves like to BFS. If R is a relation over $2n$ variables $\{x_1, x_1', \dots x_n, x_n'\}$, we let R' be a relation over $2(n+1)$ variables $\{x_0, x_0', x_1, x_1', \dots x_n, x_n'\}$. Specifically, let $R' := x_0 \oplus x_0' \wedge R$. The corresponding decomposition into sub-functions (as visualized in Fig. 3) looks like

$$R' = \begin{pmatrix} 0 & R'_{|01} \\ R'_{|10} & 0 \end{pmatrix}.$$

For a relation like this, the loop on line 6 relies entirely on the image computation steps to expand the set of reachable states, while the recursive calls never add any states. This effectively turns ReachBdd into BFS.

3.3 MDD Generalization

In this section, we generalize ReachBdd (Algorithm 1) from a BDD operation to an MDD operation (Algorithm 3, correctness discussed in Sect. 3.4). For simplicity, let us assume we have a MDD encoding a set of states of n variables, each of which takes values from the same domain $\mathcal{D} = \{0, 1, \ldots, m-1\}$, and an MDD which encodes the transition relation which has $2n$ variables (n source variables and n target variables). As shown below, the state MDD can be divided into m parts, and the relation MDD into m^2 parts, similar to how the BDDs are split up in Fig. 3. For ease of notation we denote $m-1 = m'$:

$$S = \begin{pmatrix} S_{|0} \\ S_{|1} \\ \vdots \\ S_{|m'} \end{pmatrix} \qquad R = \begin{pmatrix} R_{|00} & R_{|01} & \cdots & R_{|0,m'} \\ R_{|10} & R_{|11} & \cdots & R_{|1,m'} \\ \vdots & \vdots & \ddots & \vdots \\ R_{|m',0} & R_{|m',1} & \cdots & R_{|m',m'} \end{pmatrix}$$

Where the BDD algorithm iterates over four transition relations $R_{|ij}$, the MDD algorithm simply iterates over all m^2 relations $R_{|ij}$. When $i = j$, $R_{|ij}$ contains transitions for which the first variable stays the same, and we can call ReachMdd($S_{|i}, R_{|ij}$). For all cases where $i \neq j$ we use Image($S_{|i}, R_{|ij}$) instead.

Algorithm 3: An MDD operation for computing reachability. Cache lookup/insert for dynamic programming after line 5 and 11 are omitted.

```
1  def ReachMdd(S, R) :                  ▷ For MDDs S, R on n, 2n variables.
2      if S = 0 then return 0
3      if R = 0 then return S
4      if S = 1 then return 1
5      if R = 1 ∧ S ≠ 0 then return 1
6      while S did not converge do
7          for i, j ∈ D do
8              if i = j then
9                  S[i] ← ReachMdd(S[i], R[ij])
10             else then
11                 S[j] ← Union(S[j], Image(S[i], R[ij]))
12     return MakeNode(var(S), S[0], ..., S[m − 1])
```

We note that ReachBdd does not generalize so well to MDDs, in the following sense: for BDDs, Algorithm 1 has two ReachBdd calls and two Image calls inside the loop. However for MDDs, we get $O(m)$ ReachMdd calls and $O(m^2)$ Image calls every loop iteration. In the MDD case, a larger part of the computation is no longer handled by recursive calls, but instead by image computations.

3.4 Correctness

In this section we give a sketch of the correctness proofs for both ReachMdd (Theorem 1) and ReachBdd (Corollary 1). A complete proof can be found in [11, App. A].

Theorem 1. *Given two MDDs S and R with n and $2n$ variables respectively,* ReachMdd(S, R) *(Algorithm 3) computes all the reachable states $S.R^*$.*

Proof Sketch. The correctness of ReachMdd can be shown by means of algorithm transformation from breadth-first search. The algorithm for BFS (given below) directly follows from the definition $S.R^* = \bigcup_{k=0}^{\infty} S.R^k$, as shown by the Knaster-Tarski theorem [39].

```
1 while S did not converge do
2   S ← Union(S, Image(S, R))
3 return S
```

The two main steps in the transformation are as follows: first, using the Shannon decomposition, the computation of Image$(S, R))$ can be split up into calls Image$(S[i], R[ij]))$ for all $i, j \in \mathcal{D}$, the results of which can be combined with a MakeNode function as on line 12 of Algorithm 3. Second, since we are ultimately computing $S.R^*$, the calls Image$(S[i], R[ij]))$ can be replaced with ReachMdd$(S[i], R[ij])$ when $i = j$. The algorithm ReachMdd follows directly from these two steps.

Corollary 1. *Given two BDDs S and R,* ReachBdd(S, R) *(Algorithm 1) computes all the reachable states $S.R^*$.*

Proof Sketch. The correctness of ReachBdd can be shown from Theorem 1 by taking an MDD with $\mathcal{D} = \{0, 1\}$.

4 Empirical Evaluation

4.1 Experimental Setup

We implemented the new algorithms in the decision diagram package Sylvan [21,23,24], using the task-based scheduling as described in Sect. 3.1. Instead of MDDs, Sylvan supports LDDs (see Sect. 2.2), which can be seen as a particular implementation of MDDs. For our experiments, we compare against the saturation procedure for BDDs and LDDs as implemented in Sylvan [22].[1]

[1] The implementation of our algorithms, along with the repeatable experiments can be found here: https://github.com/sebastiaanbrand/reachability.

We use a number of existing benchmark sets to evaluate the performance of our algorithms. Specifically, we use the BEEM benchmark set, consisting of 300 instances of models in the DVE language, a benchmark set of over 300 Petri nets from the Model Checking Contest 2016 (MCC 2016) [31], and a small set of Promela models, compiled for the SpinS extension of LTSmin [7] (Table 1).

Table 1. Overview of used benchmarks

Source	Type	#
BEEM [35]	DVE	300
MCC'16 [31]	Petri-nets	357
SPINS [7]	Promela	35

For these benchmarks we use the same experimental setup as used in [22]: we first use the model checker LTSmin [28] to generate BDDs and LDDs for the (partial) transition relations and initial set of states, which are exported to `.bdd` and `.ldd` files. In LTSmin, the partial transition relations are "learned" on-the-fly, during the exploration [28], as opposed to directly building the transition relations from a modeling language like NuXMV [14]. The variables in these BDDs and LDDs have been reordered by LTSmin with Boost's Sloan algorithm, since this reordering strategy has shown good results for saturation [4,5,34].

Since we are interested in comparing the Reach algorithms against saturation, we require a single transition relation for Reach. Therefore, as a preprocessing step, for the Reach algorithms only, we merge the partial relations from LTSmin into a monolithic transition relation over all variables. The run time of the merging of partial relations is included in the total run times reported in Fig. 4. This approach is slightly disadvantaging Reach because we could also change the setup to generate monolithic relations directly (as we do for Petri nets in the comparison with ITS-tools in Sect. 4.2), but using the setup from [22] allows us to make a direct comparison with parallel saturation from [22].

We limit the run time of each reachability method to 10 min. The sequential benchmarks were performed on a machine with an AMD Ryzen 7 5800x CPU and 64 GB of available memory. The parallel benchmarks on a machine with 4 Intel Xeon E7-8890 v4 CPUs with 24 physical cores each (96 in total) and 2 TB of memory. Aside from the experiments which test parallelism specifically, all reported run times are for a single core.

4.2 Results

Comparison with Saturation. A comparison between the runtimes of saturation and Reach is given in Fig. 4. In this discussion we differentiate between smaller models (run times $\leq$1 s) and larger models (run times $\geq$1 s). For BDDs, Reach outperforms saturation on a large number of bigger DVE models, but encounters timeouts as well. For LDDs, Reach appears competitive with saturation on DVE and Promela models, while the trend for the larger Petri net models shows Reach outperforming saturation by up to a factor 100. For both BDDs and LDDs saturation is often faster on the smaller instances. This is in part due to the fact that the relative overhead of merging the partial relations is greater for smaller instances.

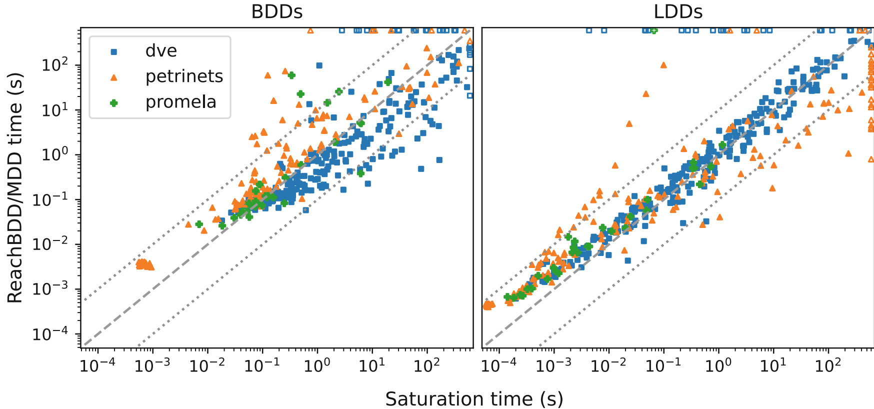


Fig. 4. The run time of finding all reachable states with BDDs (left) and LDDs (right) using ReachBdd and ReachMdd versus saturation. Open markers indicate timeouts.

Locality. As discussed in Sect. 2.6, saturation is known to work well on transition systems where the partial transition relations exhibit locality. To get insight into how locality affects our new algorithms relative to saturation, we define the *average relative bandwidth* as a metric for locality: For k partial relations $R_1, \ldots, R_k$, sorted in an ascending order based on their first variable, we can define a $k \times k$ matrix M with entries m_{ij} such that $m_{ij} = 1$ if R_i and R_j share at least one variable, and 0 otherwise. We define the bandwidth of a row $m_{i,*}$ as the distance between the first and the last non-zero element in this row. The average bandwidth is then simply the average of the bandwidths of all the rows $m_{i,*}$. The average *relative* bandwidth is the average bandwidth divided by k. Note that this $k \times k$ matrix is different from (although related to) the $k \times n$ variable matrix shown in Sect. 2.6. The matrix M and the locality metric derived from it are independent of the variable order in the decision diagram.

Plotting the run time of ReachBdd divided by the run time of saturation against this average relative bandwidth (Fig. 6), we see that there is a negative correlation. Although not extremely strong, this correlation shows that the benchmarks on which saturation outperforms our algorithms are predominantly the instances where the partial relations are relatively local, while on instances with less locality our algorithms have a greater edge over saturation.

Table 2. Parallel speedups

Algorithm	Cores	Speedup		
		P_{95}	P_{99}	$P_{99.5}$
saturation [22]	16	×8.1	×11	×11
ReachBddPar	16	×6.6	×8.3	×8.8
saturation [22]	64	×8.7	×22	×22
ReachBddPar	64	×5.6	×9.1	×17

Parallelism. Figure 5 shows the speedups obtained by ReachBddPar and the parallelized version of saturation from [22] on 16 and 64 cores. The table on the right gives the 95th, 99th and 99.5^{th} percentile of the speedups. We see that for the 16 core runs ReachBddPar is able to keep up with [22], although falling slightly behind. For the

64 core runs, while ReachBddPar falls behind [22] on the 99th percentile, it is still able to achieve a ×17 speedup on in the 99.5th percentile, compared to [22]'s ×22 (Table 2).

Comparison Against ITS-Tools. We also briefly compare how Reach performs against a state-of-the-art model checking tool. For this we pick ITS-tools [40], the overall highest scoring tool in the Model Checking Contest 2021 [30]. Since here we compare against a different tool, as opposed to comparing algorithms within the same package, we need to slightly extend our setup. We add two things: first we create a small program `pnml-encode` which builds the decision diagrams of the transition relations directly from the Petri net files. Second, we extend our LDDs with a (much simpler) version of homomorphisms which are also used in the set decision diagrams (SDDs) [19], which are a part of ITS-tools.

The results are given in Fig. 7. While ITS-tools outperforms ReachMdd on average, there is a significant number of instances where ITS-tools gives timeouts and Reach does not. Including these timeouts, Reach is faster than ITS-tools on 29% of instances. This suggest that Reach could be useful as a complementary method in an ensemble tool, where a different method can be tried if the first one times out.

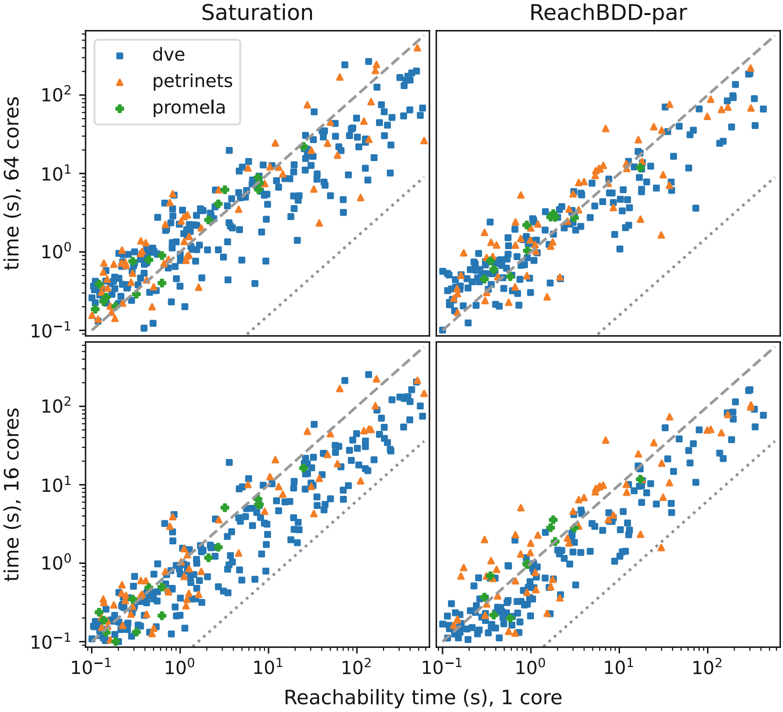
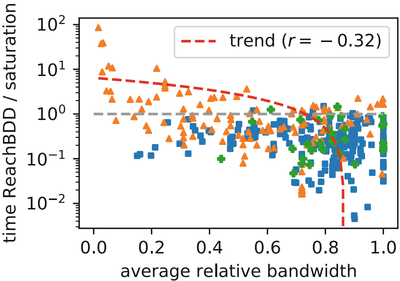


Fig. 5. Parallel speedup for saturation (left column) and ReachBddPar (right column). The dotted diagonal lines indicates a speedup of a factor 16 (bottom row) and 64 (top row) relative to the single core performance.



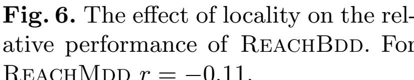


Fig. 6. The effect of locality on the relative performance of ReachBdd. For ReachMdd $r = -0.11$.

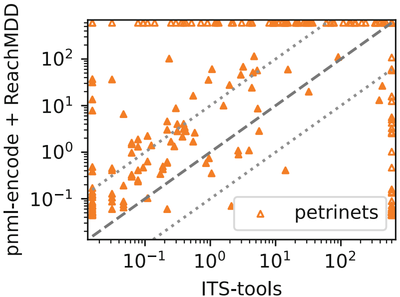


Fig. 7. Comparison of ReachMdd against computing reachable states with ITS-tools.

5 Conclusion

Summary. We presented two new reachability operations for decision diagrams: ReachBdd and ReachMdd. In contrast to other approaches, like saturation, these operations can act on a single monolithic transition relation. Similar to saturation, these new algorithms build the decision diagram for the reachable states (at least partially) bottom-up. One advantage of these operations is their simplicity. This simplicity allows us for example to more easily parallelize the decision diagram operations, as demonstrated for ReachBddPar. Empirical evaluation of ReachBdd and ReachMdd on a large number of benchmark sets shows that the new operations are competitive with saturation, and tend to outperform saturation on larger instances. Additionally, we find that ReachBddPar's peak parallel performance does not fall far behind that of saturation. Finally, our empirical results show that the Reach operations can solve 29% of instances faster than ITS-tools, which indicates Reach can be useful as a complementary algorithm.

Future Work. Saturation still outperforms our algorithms on a number of instances. Many of these instances have a lot of locality, which is exactly the regime where saturation is expected to do very well. With further investigation, ReachBdd and ReachMdd could potentially be modified to perform better on such instances. The current bottleneck, as illustrated by our analysis, is the reliance on the standard Image operation. Further integration of both operations could perhaps yield improvement.

Acknowledgements. This work was supported by the NEASQC project, funded by European Union's Horizon 2020, Grant Agreement No. 951821.

References

1. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Faster SAT and smaller BDDs via common function structure. In: ICCAD 2001, pp. 443–448. IEEE (2001)
2. Aloul, F.A., Markov, I.L., Sakallah, K.A.: FORCE: a fast and easy-to-implement variable-ordering heuristic. In: ACM VLSI, pp. 116–119 (2003)
3. Amilhastre, J., Fargier, H., Niveau, A., Pralet, C.: Compiling CSPs: a complexity map of (non-deterministic) multivalued decision diagrams. Int. J. Artif. Intell. Tools **23**(04), 1460015 (2014)
4. Amparore, E.G., Beccuti, M., Donatelli, S.: Gradient-based variable ordering of decision diagrams for systems with structural units. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 184–200. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_13
5. Amparore, E.G., Donatelli, S., Beccuti, M., Garbi, G., Miner, A.: Decision diagrams for Petri nets: a comparison of variable ordering algorithms. In: Koutny, M., Kristensen, L.M., Penczek, W. (eds.) Transactions on Petri Nets and Other Models of Concurrency XIII. LNCS, vol. 11090, pp. 73–92. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-58381-4_4
6. Bahar, R.I., et al.: Algebric decision diagrams and their applications. FMSD **10**(2), 171–206 (1997)
7. van der Berg, F., Laarman, A.: SpinS: extending LTSmin with Promela through SpinJa. ENTCS **296**, 95–105 (2013)
8. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. ENTCS **66**(2), 160–177 (2002)
9. Blom, S., van de Pol, J.: Symbolic reachability for process algebras with recursive data types. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 81–95. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85762-4_6
10. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. Trans. Comput. **45**(9), 993–1002 (1996)
11. Brand, S., Bäck, T., Laarman, A.: A decision diagram operation for reachability. arXiv preprint arXiv:2212.03684 (2022)
12. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. Trans. Comput. **100**(8), 677–691 (1986)
13. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
14. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
15. Christensen, S., Kristensen, L.M., Mailund, T.: A sweep-line method for state space exploration. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_31
16. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: an efficient iteration strategy for symbolic state—space generation. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 328–342. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_23
17. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 379–393. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_27

18. Cook, B., Podelski, A., Rybalchenko, A.: TERMINATOR: beyond safety. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 415–418. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_37
19. Couvreur, J.-M., Thierry-Mieg, Y.: Hierarchical decision diagrams to exploit model structure. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 443–457. Springer, Heidelberg (2005). https://doi.org/10.1007/11562436_32
20. Darwiche, A.: SDD: a new canonical representation of propositional knowledge bases. In: IJCAI (2011)
21. van Dijk, T., Laarman, A., van de Pol, J.: Multi-core BDD operations for symbolic reachability. ENTCS **296**, 127–143 (2013)
22. van Dijk, T., Meijer, J., van de Pol, J.: Multi-core on-the-fly saturation. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 58–75. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_4
23. van Dijk, T., van de Pol, J.: Sylvan: multi-core decision diagrams. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 677–691. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_60
24. van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. STTT **19**(6), 675–696 (2017)
25. Geldenhuys, J., Valmari, A.: Techniques for smaller intermediary BDDs. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 233–247. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44685-0_16
26. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
27. Kam, T.: Multi-valued decision diagrams: theory and applications. Multiple-Valued Logic **4**(1), 9–62 (1998)
28. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
29. Knuth, D.E.: The Art of Computer Programming, vol. 4A: Combinatorial Algorithms, Part 1. Pearson Education India (2011)
30. Kordon, F., et al.: Complete Results for the 2021 Edition of the Model Checking Contest (2021). http://mcc.lip6.fr/2021/results.php
31. Kordon, F., et al.: Complete results for the 2016 edition of the model checking contest (2016). https://mcc.lip6.fr/2016/results.php
32. Matsunaga, Y., McGeer, P.C., Brayton, R.K.: On computing the transitive closure of a state transition relation. In: International Design Automation Conference, pp. 260–265 (1993)
33. McMillan, K.L.: Symbolic model checking: an approach to the state explosion problem. Ph.D. thesis, Carnegie Mellon University (1992)
34. Meijer, J., van de Pol, J.: Bandwidth and wavefront reduction for static variable ordering in symbolic reachability analysis. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 255–271. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40648-0_20
35. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73370-6_17
36. Roig, O., Cortadella, J., Pastor, E.: Verification of asynchronous circuits by BDD-based model checking of Petri nets. In: De Michelis, G., Diaz, M. (eds.) ICATPN 1995. LNCS, vol. 935, pp. 374–391. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60029-9_50

37. Sanner, S., McAllester, D.: Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In: IJCAI, pp. 1384–1390 (2005)
38. Somenzi, F.: Binary decision diagrams. Nato ASI Subseries F CSS **173**, 303–368 (1999)
39. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pac. J. Math. **5**(2), 285–309 (1955)
40. Thierry-Mieg, Y.: Symbolic model-checking using ITS-tools. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 231–237. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_20
41. Vinkhuijzen, L., Laarman, A.: Symbolic model checking with sentential decision diagrams. In: Pang, J., Zhang, L. (eds.) SETTA 2020. LNCS, vol. 12153, pp. 124–142. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62822-2_8