Computing Under-approximations of Multivalued Decision Diagrams

Seyedehzahra Hosseini(✉) and Gianfranco Ciardo

Department of Computer Science, Iowa State University, Ames, IA, USA
{hosseini,ciardo}@iastate.edu

Abstract. Efficient manipulation of binary or multi-valued decision diagrams (BDDs or MDDs) is critical in symbolic verification tools. Despite the applicability of MDDs to real-world tasks such as discovering the reachable states of a model, their large demands on hardware resources, especially memory, limit algorithmic scalability. In this paper, we focus on memory-constrained algorithms that employ a novel $\mathcal{O}(m \log n)$-time under-approximation technique for MDDs, where m and n are the number of MDD edges and nodes, respectively. The effectiveness of our approach is demonstrated experimentally by a reduction in peak memory usage for the symbolic reachability computation of a set of Petri nets.

Keywords: decision diagrams · under-approximation · memory constraints

1 Introduction

Multi-valued decision diagrams (MDDs) are a compact symbolic representation of discrete functions over finite domains, such as those used by verification algorithms to validate a system's intended properties, where we need to manipulate large propositional formulae. To this end, reachability analysis is often the first step in the study of a discrete-state system. The most basic MDD-based method to discover the reachable state space is symbolic breadth-first search (BFS), which finds new reachable states by applying the next-state function to the set of currently-known reachable states, until it finds a fixpoint (until it cannot find any new state). Such approach is highly effective, but it cannot complete reachability analysis for many finite but large systems [8,9]. In other words, "exact" formal methods approaches *always* provide a proof or counterexample for a system property by exhaustively searching through all potential behaviors *given enough resources*, but "under-approximation" approaches *may* deliver the same result *within given resource limitations*.

Ravi et al. proposed three approximations, based on the *density* of each node p (number of minterms for node p divided by number of nodes reachable from p, see Sect. 2) in a fully-reduced binary decision diagram (BDD). The first approach [8] computes each node's density, finds a replacement node (top-down, one of its children, grandchildren, or terminal **0**, in that order), and applies this

Work supported in part by National Science Foundation under grant CCF-2212142.

L. Gomes and R. Lorenz (Eds.): PETRI NETS 2023, LNCS 13929, pp. 243–263, 2023.
https://doi.org/10.1007/978-3-031-33620-1_14

replacement based on its impact (number of minterms that would be removed and a lower bound on number of nodes that would be eliminated). Its runtime is quadratic in the BDD size. The second approach [9] (heavy branch subsetting) considers the number of minterms in the node's children, and deletes the child with the fewest minterms until the BDD size drops below a given threshold. Its runtime is linear in the BDD size, but it might create a string of nodes at the top of the BDD, each with one child set to **0**. The third approach [9] (shortest-path) favors short paths, since they encode more minterms, by assigning a path-length to each node v (sum of the length of the shortest paths from root to v and from v to terminal **1**), and deleting nodes with largest path-length. This performs best when the BDD has many paths of various lengths. Up to now, this has been the state of the art on BDD under-approximation.

An important application where this under-approximation can be effectively used to answer questions about the original set is *partial model checking*. For example, suppose we are generating the state space of a system to find out whether it can experience a deadlock. If, at some point, we have generated a (partial) set of reachable states $\mathcal{X}$ encoded in an MDD, but we are running out of memory, we can eliminate some states in $\mathcal{X}$, resulting in a set $\mathcal{X}' \subset \mathcal{X}$, hopefully with much smaller memory requirements (many fewer MDD nodes). Then, we can restart the state space exploration from $\mathcal{X}'$ and, if at any point we find a deadlock state s, we know that the system would be able to reach this deadlock. Furthermore, if each under-approximation $\mathcal{X}' \subset \mathcal{X}$ is chosen with care (e.g., ensuring that the initial state of the system is retained), then the entire reachable state space can still be reached from $\mathcal{X}'$, given enough iterations. If instead we do not find a deadlock, we cannot conclude that the original system is deadlock free, unless we are sure that, upon termination, the set $\mathcal{X}$ of encoded states is not a strict under-approximation, i.e., it is actually the entire state space. This may happen because different reachability algorithms may build the same final set going through different sequences of MDDs, some much more compact than others. This is the reason for the success of the *saturation* algorithm [4].

The rest of this paper is organized as follows. Section 2 gives background on MDD under-approximation and decomposition. Section 3 introduces our MDD under-approximation approach, Sect. 4 shows how to improve its speed, and Sect. 5 uses it for state-space generation. Section 6 reports experimental results on a set of Petri net benchmarks. Finally, Sect. 7 concludes and discusses future work.

2 Preliminaries

An L-level *quasi-reduced multi-valued decision diagram* (MDD) is a directed acyclic edge-labeled multi-graph where:

- Level 0 can only contain the two *terminal* nodes **0** and **1**.
- Each *nonterminal* node p belongs to a level $p.lvl = k \in \{1, ..., L\}$ and has n_k outgoing edges labeled by distinct elements of $\mathcal{S}_k = \{0, ..., n_k - 1\}$, pointing

to nodes at level $k-1$ or to $\mathbf{0}$, but not all its outgoing edges can point to $\mathbf{0}$. If the edge labeled by i_k points to a node q, we write $p[i_k] = q$.
– There are no *duplicates*: if $p.lvl = q.lvl = k$ and $p[i_k] = q[i_k]$ for all $i_k \in \mathcal{S}_k$, then $p = q$.

(we recall that an alternative canonical version of MDDs, *fully-reduced* MDDs, forbids both duplicate and *redundant* nodes, i.e., any nonterminal node p such that all its outgoing edges point to the same node, $p[0] = p[1] = \cdots = p[n_k - 1]$).

MDDs encode functions of the form $\mathcal{S}_{L:1} = \mathcal{S}_L \times \cdots \times \mathcal{S}_1 \to \mathbb{B}$. Specifically, node p at level k encodes $f_p : \mathcal{S}_{k:1} = \mathcal{S}_k \times ... \times \mathcal{S}_1 \to \mathbb{B}$, defined recursively by

$$f_p(i_k, ..., i_1) = \begin{cases} p & \text{if } p.lvl = 0 \\ f_{p[i_k]}(i_{k-1}, ..., i_1) & \text{if } p.lvl > 0. \end{cases}$$

As defined, an MDD can have multiple *roots* at level L, each encoding a "function of interest" (except for the constant function 0, which is encoded by $\mathbf{0}$), but we focus on MDDs encoding a single function, with one root node $r^\star$ at level L, with the understanding that the MDD has no other roots unless stated otherwise, i.e., "MDD $r^\star$" means "the MDD rooted at $r^\star$".

Given node p at level $k > 0$, we recursively define the node reached from p through sequence $\alpha = (i_k, i_{k-1}, ..., i_{h+1}) \in \mathcal{S}_{k:h+1}$, for $L \geq k \geq h \geq 0$, as:

$$p[\alpha] = \begin{cases} p & \text{if } \alpha \text{ is the empty sequence} \\ \mathbf{0} & \text{if } \alpha = (i_k, \beta) \text{ and } p[i_k] = \mathbf{0} \\ q[\beta] & \text{if } \alpha = (i_k, \beta) \text{ and } p[i_k] = q \neq \mathbf{0}. \end{cases}$$

We can use an MDD $r^\star$ to encode a set of *states*. Let the substates reaching node p, or "above" p, and those encoded by p, or "below" p, be respectively $\mathcal{A}(p) = \{\alpha \in \mathcal{S}_{L:k+1} : r^\star[\alpha] = p\}$ and $\mathcal{B}(p) = \{\beta \in \mathcal{S}_{k:1} : p[\beta] = \mathbf{1}\}$. Thus, $\mathcal{A}(p)$ is the set of paths from $r^\star$ to node p and $\mathcal{B}(p)$ the set of paths from p to terminal $\mathbf{1}$. Then, the set of states "traversing" p is $\mathcal{S}(p) = \mathcal{A}(p) \times \mathcal{B}(p)$. As a special case, the set of states encoded by the MDD $r^\star$ is $\mathcal{S}(r^\star) = \mathcal{B}(r^\star) = \mathcal{A}(\mathbf{1}) = \mathcal{S}(\mathbf{1})$.

Finally, let $\mathcal{N}(p)$ be the set of nonterminal nodes reachable from p at level k, $\mathcal{N}(p) = \{q : k \geq q.lvl > 0 \wedge \exists \alpha \in \mathcal{S}_{k:q.lvl+1}, p[\alpha] = q\}$. As a special case, $\mathcal{N}(r^\star)$ is the entire set of nonterminal nodes in the MDD.

Letting $\mathcal{M}(k)$ be the set of MDD nodes at level $k \in \{1, ..., L\}$, we can partition the states encoded by MDD $r^\star$ according to which level-k node they traverse: for any $k \in \{1, ..., L\}$, we have $\mathcal{S}(r^\star) = \bigcup_{p \in \mathcal{M}(k)} \mathcal{S}(p) = \bigcup_{p \in \mathcal{M}(k)} \mathcal{A}(p) \times \mathcal{B}(p)$.

We assume that the nonterminal nodes of the MDD, $\mathcal{N}(r^\star) = \bigcup_{k=1}^{L} \mathcal{M}(k)$, are stored in a *unique table* organized by level. This allows us to access the nodes at specific level efficiently and avoid node duplication.

an MDD rooted at node r^* can encode a large, even enormous, set of states, but its memory efficiency, measured as the number of states it encodes divided by the number of nodes it uses to encode them, $|\mathcal{S}(r^*)|/|\mathcal{N}(r^*)|$, is highly dependent on the specific set being encoded. For example, the *full* set $\mathcal{S}_{L:1}$ requires only a

chain of L nonterminal nodes: the node at level k has all its n_k outgoing edges pointing to the node at level $k-1$, or terminal **1** if $k = 1$; this is the same number of nodes required to encode a single state $(i_L, ..., i_1)$: in this MDD, all the outgoing edges of the node at level k point to terminal **0**, except for the edge labeled with i_k, which points to the node at level $k-1$, or terminal **1** if $k = 1$. Furthermore, it is well-known that the size of the MDD encoding a given set can be highly dependent on the chosen *variable order* [2], that finding the optimal variable order is NP-hard [1], and that some particularly "difficult" subsets of $\mathcal{S}_{L:1}$ require an exponential number of nodes for *any* variable order [2].

One approach explored by researchers to reduce memory consumption (measured in number of nodes) is to under-approximate a set by encoding most of its elements (states), but with substantially fewer nodes. More precisely, we formulate a *threshold* version of the *under-approximation problem* as:

> Given MDD $r^\star$ and threshold $T \in \mathbb{N}$, find MDD $s^\star$ such that $|\mathcal{S}(s^\star)|$ is maximum among all MDDs $t^\star$ satisfying $|\mathcal{N}(t^\star)| \leq T$ and $\mathcal{S}(t^\star) \subseteq \mathcal{S}(r^\star)$.

3 Our under-approximation Algorithm

For any nonterminal node p, let its *unique-below-set* be the set of nonterminal nodes that can be reached from the root $r^\star$ only by first traversing p:

$$\mathcal{U}_b(p) = \{q \in \mathcal{N}(p) : \forall \alpha, r^\star[\alpha] = q \Rightarrow \exists \alpha', \alpha'', \alpha = \alpha' \cdot \alpha'' \wedge r^\star[\alpha'] = p\} \subseteq \mathcal{N}(p).$$

$\mathcal{U}_b(p)$ always includes p and has the property that, if we remove p from the MDD (by redirecting to **0** any edge pointing to p), the remaining nodes in $\mathcal{U}_b(p)$ become unreachable from $r^\star$, thus they, too, must be removed from the MDD.

Analogously, let the *unique-above-set* of p be the set of nonterminal nodes, at levels strictly above p, that can reach **1** only by traversing p:

$$\mathcal{U}_a(p) = \{q \in \mathcal{N}(r^\star) : \forall \alpha, q[\alpha] = \mathbf{1} \Rightarrow \exists \alpha', \alpha'', \alpha = \alpha' \cdot \alpha'' \wedge \alpha' \neq \epsilon \wedge q[\alpha'] = p\}.$$

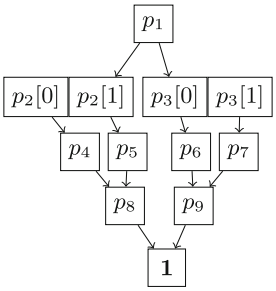
Again, if we remove p from the MDD, all nodes in $\mathcal{U}_a(p)$ must be removed from the MDD as well, as they encode the empty set.

Intuitively, the key idea in our under-approximation is to select a node $p^\star$ and remove the nodes in $\mathcal{U}(p^\star) = \mathcal{U}_b(p^\star) \cup \mathcal{U}_a(p^\star)$ from the MDD (by redirecting to **0** any edge pointing to them). Then, the resulting MDD $s^\star$ satisfies $\mathcal{S}(s^\star) \subset \mathcal{S}(r^\star)$ and $|\mathcal{N}(s^\star)| < |\mathcal{N}(r^\star)|$, since $|\mathcal{N}(r^\star)| \geq |\mathcal{N}(s^\star)| + |\mathcal{U}(p^\star)|$. After this step, we test whether $|\mathcal{N}(s^\star)| \leq T$, and continue removing nodes in this manner if this is not yet the case. It is then essential to devise a good and efficient strategy to pick node $p^\star$ at each iteration. We do so by defining the *density* of node p as

$$Density(p) = |\mathcal{S}(p)|/|\mathcal{U}(p)|,$$

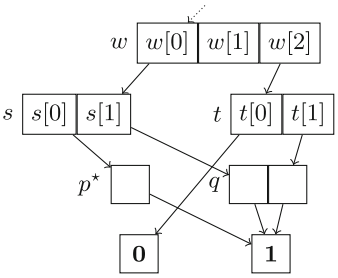
and letting $p^\star$ be a node with the smallest density (it may not be unique).

The approach must ensure that, by eliminating the selected node $p^\star$, the resulting MDD encodes a nonzero function, which would be an obvious but undesirable answer to any under-approximation problem. By checking that $p^\star$ is not



Node p	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
$\lvert\mathcal{S}(p)\rvert$	4	2	2	1	1	1	1	2	2
$\lvert\mathcal{U}(p^\star)\rvert$	9	4	4	1	1	1	1	4	4
$Density(p)$	4/9	1/2	1/2	1	1	1	1	1/2	1/2

Fig. 1. An MDD where the root is the node with the lowest density.



After removing node $p^\star$, edge $s[0]$ points to **0**, thus node s becomes equal to node t, and must be merged with it. This in turn makes node w's pointers $w[0] = w[2]$, and so on (i.e., node w could become equal to some other existing node at its level).

Fig. 2. A portion of an MDD where eliminating $p^\star$ produces a duplicate node.

the only node at its level, the algorithm ensures that the resulting MDD encodes a nonzero function. This is an issue because in some cases, the lowest density nodes in an MDD could those that are the only ones on their level (including $r^\star$), with density $|\mathcal{S}(r^\star)|/|\mathcal{N}(r^\star)$. Figure 1 shows such an MDD, together with the density of each node. The root node p_1 in this MDD is the (only) node with the lowest density. Our algorithm then avoids removing a node if it is the only one at its level (since doing so would remove *all* nodes and *all* states).

We wrote $|\mathcal{N}(r^\star)| \geq |\mathcal{N}(s^\star)| + |\mathcal{U}(p^\star)|$, not $|\mathcal{N}(r^\star)| = |\mathcal{N}(s^\star)| + |\mathcal{U}(p^\star)|$, because, after removing $\mathcal{U}(p^\star)$, some nodes with edges pointing to $p^\star$, once modified to point to **0** instead, might duplicate existing nodes, in which case they are merged with them (this in turn may make nodes pointing to them become duplicates as well, and so on). The MDD of Fig. 2 illustrates this situation.

Ideally, we would compute the number of nodes that become duplicates and are eliminated when removing each candidate node p, so that we could know beforehand the exact size of the resulting MDD if we removed $\mathcal{U}(p)$, but this is too computationally expensive (essentially, it amounts to performing the removal of p and observing its effect on the higher levels). Thus, we instead call a recursive algorithm to eliminate these duplicate nodes *after the fact*, so that $|\mathcal{U}(p^\star)|$ is just a lower bound on the number of nodes actually eliminated by removing $p^\star$, and our under-approximation algorithm is not guaranteed to be optimal.

The algorithm in Fig. 3 detects and removes the duplicate nodes created by the removal of $p^\star$, to ensure that the resulting MDD is canonical. It must be called as RmDuplicate($p^\star$), and it removes $p^\star$, $\mathcal{U}(p^\star)$, and any resulting duplicate node. Nodes in $\mathcal{U}_b(p^\star)$ are deleted by disconnecting $p^\star$, while nodes in $\mathcal{U}_a(p^\star)$ and the resulting duplicate nodes are eliminated by RmDuplicate. Algorithm RmDuplicate uses *Map*, a mapping of the identifiers of the nodes at level $k^\star = p^\star.lvl$, initialized to $Map(q.id) = q.id$ except for $Map(p^\star.id) = \mathbf{0}$:

$$\forall q \in \mathcal{N}(r^\star), q.lvl = p^\star.lvl, \quad Map(q.id) = \begin{cases} \mathbf{0} & \text{if } q = p^\star \\ q.id & \text{otherwise.} \end{cases}$$

RmDuplicate moves through the MDD levels, from $k^\star+1$ to L because, if node p is mapped to $\mathbf{0}$, all of its ancestors should be updated. At level l, it checks each node q at that level and updates it if any of its children is mapped to $\mathbf{0}$ or any other node (line 12). If the children of node q change, the algorithm checks to see if the modified node q duplicates a node already in the unique table. If the unique table already contains a node q' equal to the modified node q, any node at level $l+1$ pointing to q should point to q' instead (line 19); otherwise, a new node q^{new} should be created, and any node pointing to q should now point to node q^{new} (line 22). Either way, the required change is recorded by updating the entry for q in the *Map* for level $l+1$. After RmDuplicate completes, *Map* for level $k^\star$ contains $p^\star.id$, but the unique table does not include $p^\star$. $Map(p.id)$ is $p.id$ for each nonterminal node $p.id$ if and only if all of its nonterminal children are in the unique table level $k-1$; otherwise, node p is removed or modified because at least one of its children is removed or modified. Therefore,

$$Map(p.id) = \begin{cases} p.id & \text{if } \forall i \in \mathcal{S}_k : p[i] \in \mathcal{M}(k-1) \\ \mathbf{0} \text{ or } \{q.id : q \in \mathcal{M}(k)\} & \text{otherwise.} \end{cases}$$

Eliminating duplicate nodes induced by removing $p^\star$ requires $O(\mathcal{N}(r^\star))$ time. RmDuplicate is a specialization of Bryant's reduction algorithm [2]: we achieve the same effect, but require a smaller cache (*Map*) because we proceed by level. The rest of this section describes how to compute node densities. See Table 1 for a summary of the acronyms used in our under-approximation, and their meaning.

3.1 Incoming-edge-count

Algorithm Iec in Fig. 4 counts the number of incoming edges to each nonterminal node. It is called as Iec($r^\star$) after setting the "incoming edge" counter $p.iec$ to 0, for each nonterminal node p. Theorem 1 addresses the correctness of algorithm Iec.

Theorem 1. The call Iec($r^\star$) sets $p.iec$, for any nonterminal node p, to the number of incoming edges to p. Its runtime is $\mathcal{O}(\mathcal{N}(r^\star))$.

Table 1. Acronyms used in our under-approximation and their meaning.

$\mathcal{U}_b(p)$	Unique Below node set of node p
$\mathcal{U}_a(p)$	Unique Above node set of node p
$\mathcal{U}(p)$	Unique node set of node p
$IEC(p)$	Incoming Edge Count of node p
$ASC(p)$	Above State Count of node p
$BSC(p)$	Below State Count of node p
$\mathcal{H}(p)$	Highest-unique-below-set of node p
$\mathcal{L}(p)$	Lowest-unique-above-set of node p

Proof. For a node p at level k, the for-loop at lines 2- 5 is executed $|\mathcal{S}_k|$ times; thus, each outgoing edge from p, if it is an incoming edge for a corresponding nonterminal child of p, is traversed and counted only once. IEC(p) calls itself only on its unvisited children (identified by having their incoming-edge-count equal to 0). Considering the sizes $|\mathcal{S}_k|$ as constants, the runtime is linear in the number of MDD nodes, $\mathcal{O}(\mathcal{N}(r^\star))$. □

3.2 Above-state-count

Algorithm ASC in Fig. 5 computes the *above-state-counts*, i.e., the number $p.asc = |\mathcal{A}(p)|$ of substates from $r^\star$ to each nonterminal node $p \in \mathcal{N}(r^\star)$. It is called as ASC($r^\star$) after initializing $r^\star.asc = 1$, and $p.asc$ to 0 for all other nonterminal nodes p, and after having computed the incoming-edge-counts with the call IEC($r^\star$). To keep track of when all edges to a node p have been traversed (implying that counter $p.asc$ has the correct final value and the recursion can proceed downward), algorithm ASC decrements the incoming-edge-count $p.iec$ of node p every time p is reached, so that $p.iec$ will have value 0 after the call ASC($r^\star$) completes. Theorem 2 addresses the correctness of algorithm ASC.

Theorem 2. The call ASC($r^\star$) sets $p.asc$ to $|\mathcal{A}(p)|$ for any nonterminal node p. Its runtime is $\mathcal{O}(\mathcal{N}(r^\star))$.

Proof. To compute the above-state-count of node p, we need to have computed the correct value $q.iec$ for each node q with a path to p. The algorithm uses the fact that, in each recursive call ASC(p), $p.asc$ has the correct value of above-state-count for node p, and $p.iec = 0$.

Obviously, since IEC sets the incoming-edge-count of the root to 0, $r^\star.iec = 0$.

Similarly, the recursive call ASC($p[i]$) on line 6 occurs only if $p[i].iec$ is 0, which means that $p[i].asc$ has been updated to take into account the (correct) $q.asc$ value of each parent q of $p[i]$. Then, in each recursive call ASC(p), $p.asc$ has the correct value of the above-state-count. ASC($r^\star$) visits each node once, therefore its runtime is $\mathcal{O}(\mathcal{N}(r^\star))$ □

```
1: procedure RmDuplicate(p : node)
2:     k ← p.lvl;
3:     for all q ∈ M(k) do                                  ▷ Initialization for Map
4:         if q = p then Map[q] ← 0;
5:         else Map[q.id] ← q.id;
6:     for l = k to L − 1 do
7:         Clear(NextMap)
8:         for all q ∈ M(l + 1) do
9:             NextMap[q.id] ← q.id;    ▷ Remove node q from unique table level l + 1
10:            NodeChanged ← false
11:            for i = 0 to |S_k| − 1 do
12:                if q[i] ≠ Map[q[i]] then                          ▷ q will change
13:                    q[i] ← Map[q[i]];
14:                    NodeChanged ← true;
15:            if NodeChanged then
16:                if q = 0 then NextMap[q.id] ← 0;
17:                q'.id ← find(q, M(l + 1)); ▷ Find node q in unique table level l + 1
18:                if q'.id ≠ NULL then          ▷ Changed node q is a duplicate of q'
19:                    NextMap[q.id] ← q'.id;
20:                else
21:                    q^new ← NewNode(q);  ▷ Build a node based on changed node q
22:                    NextMap[q.id] ← q^new.id;
23:        Map ← NextMap;
```

Fig. 3. Algorithm to remove duplicate nodes.

3.3 Below-state-count

One of the fundamental unary operations for MDDs is to compute the *cardinality* of the set encoded by a node p, i.e., the number of paths from p to **1**. We call this the *below-state-count* of node p. Bsc($r^\star$) should be called after setting $p.bsc$ to 0 for all nonterminal nodes p. This algorithm to compute the cardinality is well known, so we include the pseudo-code in Fig. 6, but omit its proof.

3.4 Highest-unique-below-set

The *highest-unique-below-set* of node p is the subset of $\mathcal{U}_b(p) \setminus \{p\}$ containing all the nodes that are in the unique-below-set of p but not in the unique-below-set of a node $q' \neq p$ that is in the unique-below-set of p:

$$\mathcal{H}(p) = \{q \in \mathcal{U}_b(p) \setminus \{p\} : \forall q' \in \mathcal{U}_b(p) \setminus \{p, q\}, q \notin \mathcal{U}_b(q')\}.$$

We define $\mathcal{H}(p)$ because it turns out that every nonterminal node $q \neq r^\star$ belongs to $\mathcal{H}(p)$ for exactly one p, but possibly to $\mathcal{U}_b(p)$ for many nodes p, thus we can store all sets $\mathcal{H}(p)$ using memory linear in the number of MDD nodes, but, in general, (explicitly) storing all sets $\mathcal{U}_b(p)$ would require an overall quadratic memory in the number of MDD nodes. Furthermore, we could still obtain $\mathcal{U}_b(p)$

```
1: procedure IEC(p : node)                          ▷ Node p satisfies p.iec = 0
2:     for i = 0 to |S_{p.lvl}| − 1 do
3:         if p[i].lvl > 0 then
4:             if p[i].iec = 0 then IEC(p[i]);
5:             p[i].iec ← p[i].iec + 1;
```

Fig. 4. Algorithm to compute the incoming-edge-count of each node.

```
1: procedure ASC(p : node)                          ▷ Assume r*.asc has been initialized to 1
2:     for i = 0 to |S_{p.lvl}| − 1 do
3:         if p[i].lvl > 0 then
4:             p[i].asc ← p[i].asc + p.asc;
5:             p[i].iec ← p[i].iec − 1;
6:             if p[i].iec=0 then ASC(p[i]);    ▷ Visit p[i] if this was the last edge to it
```

Fig. 5. Algorithm to compute the above-state-count of each node.

as the transitive closure of $\mathcal{H}(p)$, this is explained and proved in Theorem 4, but we do not need to, as we merely need to know its size $|\mathcal{U}_b(p)|$ to compute our under-approximation, not its actual elements.

Theorem 3. The set $\{\mathcal{H}(q) : q \in \mathcal{U}_b(p), \mathcal{H}(q) \neq \emptyset\}$ is a partition of $\mathcal{U}_b(p) \setminus \{p\}$.

Proof. To prove the proposition we must verify that $\mathcal{H}(p)$ satisfies the following:

1. $\bigcup_{q \in \mathcal{U}_b(p)} \mathcal{H}(q) = \mathcal{U}_b(p) \setminus \{p\}$. First, it is easy to show that, if $t \in \bigcup_{q \in \mathcal{U}_b(p)} \mathcal{H}(q)$ then $t \in \mathcal{U}_b(p) \setminus \{p\}$. If $t \in \mathcal{H}(q)$ for some $q \in \mathcal{U}_b(p)$, then, all paths from $r^\star$ to q pass through p, and all paths from $r^\star$ to t pass through q. Therefore, all paths from $r^\star$ to t pass through p, so $t \in \mathcal{U}_b(p) \setminus \{p\}$. To prove containment in the other direction, consider the lowest node $q \in \mathcal{U}_b(p)$ such that $t \in \mathcal{U}_b(q)$; there must be such a node since, at the very least, we could have $q = p$. But since q is the lowest node satisfying $t \in \mathcal{U}_b(q)$, then no other node q' between q and t can satisfy $t \in \mathcal{U}_b(q')$, thus $t \in \mathcal{H}(q)$, by definition.
2. For any given pair of nodes $q, q' \in \mathcal{U}_b(p)$, $\mathcal{H}(q)$ and $\mathcal{H}(q')$ are disjoint, i.e., $\mathcal{H}(q) \cap \mathcal{H}(q') = \emptyset$. By contradiction, assume that $\exists q, s, t \in \mathcal{U}_b(p), s \neq t$ and $q \in \mathcal{H}(s) \cap \mathcal{H}(t)$, i.e., $q \in \mathcal{H}(s)$ and $q \in \mathcal{H}(t)$, therefore $q \in (\mathcal{U}_b(s) \setminus \{s\}) \cap (\mathcal{U}_b(t) \setminus \{t\})$, thus $q \in \mathcal{U}_b(s)$ and $q \in \mathcal{U}_b(t)$. This means that any path from $r^\star$ to q must pass through both s and t and, since $s \neq t$, nodes s and t must be at different levels. Without loss of generality, assume that s is above t, then q cannot be in $\mathcal{H}(s)$, thus we have a contradiction. □

Theorem 4. Let the reflexive and transitive closure of $\mathcal{H}(p)$ for a given node p be defined as $\mathcal{H}^*(p) = \mathcal{H}(p) \cup \mathcal{H}(\mathcal{H}(p)) \cup \cdots$, where $\mathcal{H}(\{p_1, ..., p_c\}) = \bigcup_{d=1}^{c} \mathcal{H}(p_d)$. If $\mathcal{U}_b(p)$ contains nodes beyond p, then $\{\mathcal{H}^n(p) : n \in \mathbb{N}, \mathcal{H}^n(p) \neq \emptyset\}$ is a coarser partition than $\{\mathcal{H}(q) : q \in \mathcal{U}_b(p), \mathcal{H}(q) \neq \emptyset\}$. Thus, $\mathcal{H}^*(p) = \mathcal{U}_b(p) \setminus \{p\}$.

Proof. We need to prove that, for any $q \in \mathcal{U}_b(p)$, there is a minimum n such that $\mathcal{H}(q) \subseteq \mathcal{H}^n(p)$. Consider $t \in \mathcal{U}_b(p) \setminus \{p\}$. Since Theorem 3 states that

```
1: function Bsc(p : node)
2:     if p = 1 ∨ p = 0 then return p;
3:     if p.bsc = 0 then
4:         for i = 0 to |S_{p.lvl}| − 1 do p.bsc ← p.bsc + Bsc(p[i]);
5:     return p.bsc;
```

Fig. 6. Algorithm to compute the below-state-count of each node.

```
1: function Ubc(p : node)                          ▷ ∀p ∈ N(r*) : p.ubc ← 1
2:     if p.vst = 1 then return p.ubc;
3:     for all q ∈ H(p) do p.ubc ← p.ubc + Ubc(q);   ▷ H(p) : Sec. 3.6.
4:     p.vst ← 1;
5:     return p.ubc;
```

Fig. 7. Algorithm to compute the unique-below-count.

$\{\mathcal{H}(q) : q \in \mathcal{U}_b(p), \mathcal{H}(q) \neq \emptyset\}$ is a partition of $\mathcal{U}_b(p) \setminus \{p\}$, there exists a q such that $t \in \mathcal{H}(q)$. If $q = p$, then $t \in \mathcal{H}^n(p)$ for $n = 1$. If $q \neq p$, then we know that $\mathcal{U}_b(p)$ contains p, q, and t, and that there exists a node q_1 such that $q \in \mathcal{H}(q_1)$. If $q_1 = p$, then $q \in \mathcal{H}(p)$, $t \in \mathcal{H}(q)$ which means that $t \in \mathcal{H}^2(p)$; otherwise, we can repeat the reasoning and eventually, since $\mathcal{U}_b(p)$ is a finite set, we must eventually find a $q_n = p$, implying that $t \in \mathcal{H}^{n+1}(p)$. □

Procedure Ubc($r^\star$) computes the size $|\mathcal{U}_b(p)|$ of the unique-below-set for any nonterminal node $p \in \mathcal{N}(r^\star)$, using the information in $\mathcal{H}(p)$.

3.5 Lowest-unique-above-set

The *lowest-unique-above set* of node p is the subset of $\mathcal{U}_a(p)$ containing all nodes that are in the unique-above-set of p but not in the unique-above-set of a node in the unique-above-set of p:

$$\mathcal{L}(p) = \{q \in \mathcal{U}_a(p) : \forall q' \in \mathcal{U}_a(p) \setminus \{q\}, q \notin \mathcal{U}_a(q')\}.$$

As for $\mathcal{H}(p)$, we define $\mathcal{L}(p)$ because it turns out that every node belongs to $\mathcal{L}(p)$ for exactly one p, but possibly to $\mathcal{U}_a(p)$ for many nodes p, thus we can store all sets $\mathcal{L}(p)$ using memory linear in the number of MDD nodes, but, in general, we cannot (explicitly) store all sets $\mathcal{U}_a(p)$ in linear memory. Furthermore, again, we could obtain $\mathcal{U}_a(p)$ as the transitive closure of $\mathcal{L}(p)$, as stated in Theorem 6, but we do not need to, we only need to compute its size $|\mathcal{U}_a(p)|$.

Theorem 5. The set $\{\mathcal{L}(q) : q \in \mathcal{U}_a(p)\}$ is a partition of $\mathcal{U}_a(p)$.

Proof. Similar to that of Theorem 3. □

Theorem 6. Let the reflexive and transitive closure of $\mathcal{L}(p)$ for a given node p be defined as $\mathcal{L}^*(p) = \mathcal{L}(p) \cup \mathcal{L}(\mathcal{L}(p)) \cup \cdots$, where $\mathcal{L}(\{p_1, ..., p_c\}) = \bigcup_{d=1}^{c} \mathcal{L}(p_d)$. If $\mathcal{U}_a(p)$ contains nodes beyond p, then $\{\mathcal{L}^n(p) : n \in \mathbb{N}\}$ is a coarser partition than $\{\mathcal{L}(q) : q \in \mathcal{U}_a(p)\}$. Thus, $\mathcal{L}^*(p) = \mathcal{U}_a(p)$.

```
1: function UAC(p : node)                          ▷ ∀p ∈ 𝒩(r*) : p.uac ← 0
2:     if p.vst = 1 then return p.uac;
3:     for all q ∈ ℒ(p) do p.uac ← p.uac + 1 + UAC(q);      ▷ ℒ(p) : Sec. 3.6
4:     p.vst ← 1;
5:     return p.uac
```

Fig. 8. Algorithm to compute the unique-above-count

Proof. Similar to that of Theorem 4. □

Procedure UAC($r^\star$) computes the size of the unique-above-set $|\mathcal{U}_a(p)|$, for any nonterminal node $p \in \mathcal{N}(r^\star)$, by recursively using the information in $\mathcal{L}(p)$.

3.6 Dominator and Post-dominator

A simplistic iterative algorithm to calculate $\mathcal{H}(p)$ and $\mathcal{L}(p)$ for all nodes p has quadratic complexity in the number of MDD nodes [5]; to reduce this complexity, we use a *dominator* algorithm. Given a flow graph with a single source and sink (in our case, $r^\star$ and $\mathbf{1}$), a node v *dominates* another node w, if every path from the $r^\star$ to w must traverse v Every node $w \neq r^\star$ has at least one dominator. A node v is the *immediate dominator* of w, denoted by $idom(w) = v$, if v dominates w and every other dominator of w also dominates v. Every node $w \neq r^\star$ has a unique $idom(w)$. Importantly, q is in $\mathcal{U}_b(p)$ iff q is in $dom(p)$, and is the immediate dominator of p iff it is the only node in p's highest-unique-below-set $\mathcal{H}(p)$.

The dominator algorithm builds a *dominator tree* whose nodes $\mathcal{V}$ are the MDD nodes and whose edges $\{(idom(w), w) : w \in \mathcal{V} \setminus \{r^\star\}\}$ form a direct tree rooted at $r^\star$. It performs a depth-first search and assigns the visit time to each node, effectively defining a total order, where $v > w$ means that the visit time of node v is larger than that of node w. The dominator algorithm uses the visit time of the node instead of the original node label in the following steps. Next, for each node $w \neq r^\star$, it defines the "semidominator" $sdom(w) \in \mathbb{N}$ as:

$$sdom(w) = \min\{v : \exists \text{ path } v = v_0, v_1, ..., v_j = w \text{ s.t. } v_i > w \text{ for } 1 \leq i \leq j-1\}.$$

The algorithm uses Theorem 7 to compute $sdom(w)$ for any $w \neq r^\star$:

Theorem 7. (from [6]) For any node $w \neq r^\star$:

$$sdom(w) = \min(\{v \mid (v, w) \text{ is an edge and } v < w\} \cup \{sdom(u) \mid u > w \text{ and there is an edge } (v, w) \text{ such that } u \text{ is an ancestor of } v,\ u \xrightarrow{*} v\}).$$

Then the algorithm uses Corollary 1 to compute the immediate dominator of all nodes using the semidominator information.

1: **procedure** UNDERAPPROXONE($r^\star : node, T_{min} : int, T_{max} : int$)
2: **if** $|\mathcal{N}(r^\star)| < T_{max}$ **then return**;
3: **while** $|\mathcal{N}(r^\star)| > T_{min}$ **do**
4: BSC($r^\star$);
5: IEC($r^\star$); ▷ Initialize $r^\star.iec$ to 0
6: ASC($r^\star$); ▷ Initialize $r^\star.asc$ to 1
7: UAC($r^\star$); UBC($r^\star$);
8: **for all** $p \in \mathcal{N}(r^\star)$ **do** $Density(p) \leftarrow (p.ac \cdot p.bc)/p.uac + p.ubc$;
9: $p^\star \leftarrow$ pick one of the nodes in $\{p \mid \forall p' \in \mathcal{N}(r^\star), Density(p) \leq Density(p')\}$;
10: RMDUPLICATE($p^\star$);

Fig. 9. Algorithm to under-approximate MDD $r^\star$ by selecting one node at a time.

Corollary 1. (from [6]) Let $w \neq r^\star$ and u be a node for which $sdom(u)$ is the minimum among nodes u satisfying $sdom(w) \xrightarrow{+} u \xrightarrow{+} w$, i.e., $sdom(w)$ is a proper ancestor of u and u is a proper ancestor of w, then

$$idom(w) = \begin{cases} sdom(w) & \text{if } sdom(w) = sdom(u) \\ idom(u) & \text{otherwise.} \end{cases}$$

A node w *post-dominates* another node v, if every path from v to **1** traverse w. A node v is the immediate post-dominator of w, if v post-dominates w and every other post-dominator of w also post-dominates v. Again, node q is in $\mathcal{U}_a(p)$ iff q is in $postdom(p)$ and node q is the post-dominator for node p iff node q is the only node in p's lowest-unique-above-set.

$$q \in postdom(p) \iff q \in \mathcal{U}_a(p) \qquad ipostdom(p) = q \iff \mathcal{L}(p) = \{q\}$$

The post-dominator algorithm applies the dominator algorithm to the reverse MDD (with source **1** and sink $r^\star$) to compute the lowest-unique-above-sets.

3.7 Under-approximation (one Node at a Time)

Given MDD $r^\star$, the call UNDERAPPROXONE($r^\star, T_{min}, T_{max}$) computes an under-approximation for $r^\star$ if the size of the MDD $r^\star$ is greater than T_{max}. The algorithm reduces the size of the MDD so that it does not exceed T_{min} (T_{min} must be at least the number of MDD levels). T_{max} and T_{min} introduce hysteresis to avoid calling the under-approximation too frequently. In practice, T_{max} should be as large as possible and T_{min} a fraction of T_{max} (in our experiments, it is $0.6 \cdot T_{max}$). The algorithm computes the below-state-count, above-state-count, unique-count, and density (lines 4 , 6, 7, and 8) for each node at each iteration of the while-loop. Then, it selects a single node $p^\star$ with lowest density (line 9) and removes from the MDD the nodes in $\mathcal{U}(p^\star)$ and any resulting duplicate using RMDUPLICATE, until the number of MDD nodes is at most T_{min}.

As the algorithm recomputes the information after each deletion, the selected node $p^\star$ is the "quasi-optimal" choice: it is optimal based on density information, but it ignores the effect of removing duplicate nodes.

4 Speeding up the under-approximation

In large models, recomputing the above-state-count, below-state-count, incoming-edge-count, and unique-count after deleting each set of nodes $\mathcal{U}(p^\star)$ can be costly. UnderApproxMany selects instead *a set* of nodes $\mathcal{P}^\star$, and deletes all nodes in $\bigcup_{p\in\mathcal{P}^\star}\mathcal{U}(p)$ before recomputing all node densities.

While eliminating duplicate nodes caused by deleting just *one* set $\mathcal{U}(p^\star)$ is slightly simpler, identifying and removing *all* duplicate nodes created by removing the set of nodes $\bigcup_{p\in\mathcal{P}^\star}\mathcal{U}(p)$ has the same time complexity, thus its cost can be better amortized. The call RmDuplicateSet($\mathcal{P}^\star$) in Fig. 10 finds and removes the duplicate nodes created by eliminating the nodes in $\bigcup_{p\in\mathcal{P}^\star}\mathcal{U}(p)$, to ensure MDD canonicity. $\mathcal{K} = \{p^\star.lvl : p^\star \in \mathcal{P}^\star\}$ stores the MDD levels of the selected nodes (line 2), and *Map* maps the identifiers of nodes at level $k^\star = \min\{\mathcal{K}\}$, initialized as

$$\forall q \in \mathcal{N}(r^\star), q.lvl = p^\star.lvl, \quad Map(q.id) = \begin{cases} q.id, & \text{if } q \notin \mathcal{P}^\star \\ \mathbf{0}, & \text{otherwise.} \end{cases}$$

RmDuplicateSet traverses the MDD from level $\min(\mathcal{K}) + 1$ to L. Like RmDuplicate, it starts at level $k^\star + 1$; if q's child $q[i]$ is mapped to another node (line 15), node q must change (line 16), and the algorithm checks if the new node is a duplicate of a node q'. Any edge pointing to q from higher-level nodes must be changed.

UnderApproxOne selects node $p^\star$ with lowest density, deletes $\mathcal{U}(p^\star)$, and recomputes the density ($\mathcal{U}$, $\mathcal{B}$, and $\mathcal{A}$), while UnderApproxMany selects a set of nodes $\mathcal{P}^\star$, one after another, but it does not update the density information after selecting each node. This reduces execution time since calculating density is a heavy duty operation, but uses increasingly stale, thus less precise, information. This is because not only the number of nodes eliminated or merged after calling RmDuplicateSet is not taken into account, but also because, by selecting a sequence of nodes $(p_1, p_2, ..., p_k)$, the selection of any node except for p_1 uses an approximation of the correct values for $\mathcal{U}$, $\mathcal{A}$, and $\mathcal{B}$.

When the MDD size $\mathcal{N}(r^\star)$ exceeds T_{max}, a call to UnderApproxMany reduces the size to T_{min} or less. Selecting a set of nodes $\mathcal{P}^\star$ instead of just node $p^\star$ increases the chances of deleting more nodes than necessary. This is because, ideally, every time the algorithm selects node $p_i \in \mathcal{P}^\star$, $\mathcal{S}(p_i)$ should be disjoint from $\mathcal{S}(p_j)$, for any other $p_j \in \mathcal{P}^\star$, but this is not necessarily the case. Before adding p_i to $\mathcal{P}^\star$, the algorithm checks that $|\mathcal{S}(p_i)| + \sum_{p_j\in\mathcal{P}^\star}|\mathcal{S}(p_j)| < |\mathcal{S}(r^\star)|$, to ensure that removing the set of nodes $\mathcal{P}^\star$ does not produce an empty MDD. UnderApproxMany uses a (lower bound) on the percentage ψ of $|\mathcal{S}(r^\star)|$ that must be kept as a constraint: if $|\mathcal{S}(p_i)| + \sum_{p_j\in\mathcal{P}^\star}|\mathcal{S}(p_j)| > \psi \cdot |\mathcal{S}(r^\star)|$, the algorithm does not add p_i to $\mathcal{P}^\star$; instead, it removes just $\mathcal{U}(\mathcal{P}^\star)$ and any duplicate nodes created by removing $\mathcal{U}(\mathcal{P}^\star)$ from the MDD, then it recomputes the new densities of the nodes of the resulting MDD.

The exact call is UnderApproxMany($r^\star, T_{min}, T_{max}, \psi$), where T_{min} is the selected T_{min}, T_{max} is the maximum (triggering) threshold, and ψ is the

```
1: procedure RmDuplicateSet(P*)
2:     K ← {l : p.lvl = l, p ∈ P*};                          ▷ K is a set of levels
3:     for p ∈ M(min(K)) do                                   ▷ Initialization loop
4:         if p ∈ P* then Map[p] ← 0;
5:         else Map[p] ← p.id;
6:     for l = min(K) + 1 to L do
7:         NextMap.Clear;
8:         if l ∈ K then K ← K \ {l};
9:         for all q ∈ M(l + 1) do
10:            NodeChanged ← false;
11:            if q ∉ P* then                    ▷ If node q is not marked for deletion
12:                NextMap[q.id] ← q.id;
13:                M(l+1).remove(q); ▷ Remove node q from unique table level l + 1
14:                for i = 0 to |S_k| − 1 do
15:                    if q[i] ≠ Map[q[i]] then                ▷ q should change
16:                        q[i] ← Map[q[i]];
17:                        NodeChanged ← true;
18:                if NodeChanged then
19:                    q'.id ← M(l+1).find(q);   ▷ Find q from unique table level l+1
20:                    if q'.id ≠ NULL then              ▷ Node q is a duplicate of q'
21:                        NextMap[q.id] ← q'.id;
22:                    else
23:                        q^new ← NewNode(q); ▷ Build q^new based on changed node q
24:                        NextMap[q.id] ← q^new.id;
25:            NextMap[q] ← 0;
26:        Map ← NextMap;
```

Fig. 10. Algorithm to remove a set of duplicate nodes.

maximum percentage of removed states (required to be strictly less than 100%). The greater ψ is, the less frequently the algorithm needs to recompute node densities. line 11 ensures that UnderApproxMany deletes at least one node at a time, even when ψ is near zero (in which case UnderApproxMany behaves like UnderApproxOne).

5 Application

The first step in the study of a discrete system is often reachability analysis, i.e., the computation of its reachable states. Given an initial state set $\mathcal{S}_{init} \subseteq \mathcal{S}_{L:1}$ and a *next-state* function of the form $\mathcal{T} : \mathcal{S}_{L:1} \to 2^{\mathcal{S}_{L:1}}$, the reachability set $\mathcal{S}_{rch}$ is the smallest set $\mathcal{X}$ satisfying $\mathcal{X} = \mathcal{X} \cup \mathcal{T}(\mathcal{X}) \cup \mathcal{S}_{init}$. The breadth-first (BF) method is a common exploration approach for MDD-based reachability analysis, as it naturally implements this definition of $\mathcal{S}_{rch}$ as a fixpoint. It starts by initializing $\mathcal{S}_{rch}$ to $\mathcal{S}_{init}$, and repeatedly adds to it the states reachable from it in one application of $\mathcal{T}$, until no more new states are found. At the i^{th} iteration, $\mathcal{S}_{rch}$ contains all states at distance up to i from $\mathcal{S}_{init}$. Thus, it builds $\mathcal{S}_{rch}$ as $\mathcal{S}_{init} \cup \mathcal{T}(\mathcal{S}_{init}) \cup \mathcal{T}^2(\mathcal{S}_{init}) \cup \cdots$.

```
1: procedure UnderApproxMany(r* : node, T_min : int, T_max : int, ψ : float)
2:     if |N(r*)| < T_max then return ;
3:     while |N(r*)| < T_min do
4:         Bsc(r*);
5:         Iec(r*);                                   ▷ Initialize r*.iec to 0
6:         Asc(r*);                                   ▷ Initialize r*.asc to 1
7:         Uac(r*); Ubc(r*);
8:         for all p ∈ N(r*) do Density(p) ← (p.ac × p.bc)/p.uc;
9:         RemovedNodes ← ∅;
10:        repeat                                     ▷ Select a set of nodes with lowest density.
11:            select node p ∉ RemovedNodes with lowest Density;
12:            if Σ_{p∈P*} |S(p)| > ψ × |S(r*)| then break;
13:                P* ← P* ∪ {p};
14:                RemovedNodes ← RemovedNodes ∪ U(p);
15:        until |N(r*)| − |RemovedNodes| > T_min
16:        RmDuplicateSet(P*);
```

Fig. 11. Under-approximating MDD $r^\star$ by selecting many nodes at a time.

```
1: procedure BF(S_init, T_min, T_max)
2:     S_rch ← S_init;
3:     S_pre ← ∅;
4:     while S_pre ≠ S_rch do
5:         S_pre ← S_rch;
6:         S_rch ← S_rch ∪ T(S_pre);
7:     return S_rch;
```

Fig. 12. Algorithm to compute the reachable state space using breadth-first.

The chained BF (ChBF) approach [7] observes that, if $\mathcal{T}$ is partitioned as $\mathcal{T} = \bigcup_{\alpha\in\mathcal{E}} \mathcal{T}_\alpha$, where $\mathcal{E}$ is a set of (asynchronous) *events*, runtime and memory requirements may be reduced by using a different iteration: if $\mathcal{E} = \{\alpha, \beta, \gamma\}$, the generic i^{th} ChBF iteration updates $\mathcal{S}_{rch}$ using three sequential steps: (1) $\mathcal{S}_{rch} \leftarrow \mathcal{S}_{rch} \cup \mathcal{T}_\alpha(\mathcal{S}_{rch})$; (2) $\mathcal{S}_{rch} \leftarrow \mathcal{S}_{rch} \cup \mathcal{T}_\beta(\mathcal{S}_{rch})$; (3) $\mathcal{S}_{rch} \leftarrow \mathcal{S}_{rch} \cup \mathcal{T}_\gamma(\mathcal{S}_{rch})$. This has the effect of potentially accelerating convergence to the fixpoint, as the i^{th} iteration discovers states reachable not just through one of the three single events, but also through one of the sequences of events $\alpha\beta$, $\alpha\gamma$, $\beta\gamma$, or $\alpha\beta\gamma$.

ChBF was proposed in conjunction to Petri net models [7], the formalism we use for our experiments. In this case the events are the Petri net transitions, which are by definition asynchronous, and $\mathcal{T}_\alpha(i) = \emptyset$ if Petri net transition α is not enabled in marking i, while $\mathcal{T}_\alpha(i) = \{j\}$ if transition α is enabled in marking i and its firing in i leads (deterministically) to marking j.

```
1: procedure ChBFUA(S_init, T_min, T_max, ψ)
2:     Y ← S_init;
3:     S_rch ← S_init;
4:     while true do
5:         for α ∈ E do
6:             Y ← S_rch;
7:             S_rch ← S_rch ∪ T_α;
8:             if S_rch ≠ Y then                    ▷ Applying α added some states
9:                 UnderApprox(S_rch, T_min, T_max, ψ);
10:                S_rch ← S_rch ∪ S_init;
11:                lastNonZeroIteration ← α;
12:            else if lastNonZeroIteration = α then   ▷ No event in E added states
13:                return S_rch;
```

Fig. 13. Chained breadth-first algorithm with under-approximation.

Figure 13 shows our ChBF approach invoking "UnderApprox", i.e., either UnderApproxOne or UnderApproxMany, whenever the MDD size exceeds T_{max} (line 9). Either under-approximation may delete the initial state(s), which would then make it impossible to generate the entire state space. Thus, ChBFUA adds back the initial state(s) after each under-approximation (line 10). ChBF is exactly the same as ChBFUA, except it does not have lines 9, and 10.

As shown, algorithm ChBFUA might not halt because, after calling under-approximation, the set of states could be exactly the same as after the previous under-approximation: the algorithm is in a cycle where it adds and removes the same set of states. To recognize this situation, we should keep the old set of states, but this would require storing two MDDs; we use instead the old number of states and nodes as a proxy to (conservatively) detect this problem and let ChBFUA output a partial state space $\mathcal{S}_{part}$ instead of the full state space $\mathcal{S}_{rch}$.

Assume that, before calling under-approximation, the number of MDD nodes is $n_{old} > T_{max}$ and the number of states is rs_{old}, and that, after calling under-approximation once and firing one or more events, the numbers are $n_{new} > T_{max}$ and rs_{new}. If $n_{new} = n_{old}$ and $rs_{new} = rs_{old}$, the algorithm applies one more event resulting in n'_{new} MDD nodes and rs'_{new} states. Then, three cases may arise:

1. If $n'_{new} > n_{new}$ and $rs'_{new2} > rs_{new}$, the MDD with n_{new} nodes is not a fixpoint; the algorithm conservatively decides that the MDD with n_{new} nodes is the same as that with n_{old} nodes, it refrains from calling under-approximation, and returns the partial state space $\mathcal{S}_{part}$ encoded by the MDD with n'_{new} nodes.
2. If $n'_{new} \leq n_{new}$ and $rs'_{new} > rs_{new}$, ChBFUA continues its normal execution.
3. If $n'_{new} = n_{new}$ and $rs'_{new} = rs_{new}$, the algorithm applies a new event and repeats the check for cases 1, 2, and 3, until either case 1 or 2 happens, or all events have been applied once without discovering new states (in which case it reached a fixpoint and returns $\mathcal{S}_{rch}$, encoded by $n_{old} = n_{new}$ nodes).

6 Results

We designed a set of experiments and ran them on a Linux workstation with 16GB of RAM. We implemented ChBFUA, chained breadth-first reachability with under approximation, within the model checker SMART [3]. Our benchmark is a subset of the bounded models from the Model Checking Contest (MCC) 2021 (https://mcc.lip6.fr/2021/). Models are described as Petri nets, and most of them have one or more scaling parameters that affect their state space size. 799 models in the MCC benchmark are bounded, 499 of which generate the next-state function within 60 seconds, and 259 of which generate the entire state space using ChBF within one hour. Of these, we eliminated 72 models because they have the same peak and final number of nodes using ChBF (the under-approximation algorithm does not make sense for such models; admittedly this cannot be determined a priori). Thus, we considered the remaining 187 models. For our experiments, we selected $T_{min} = 10{,}000$ and $T_{max} = 15{,}000$, the percentage ψ of the minimum number of states to be kept was set to 0.5, and the maximum execution time for each run was set to 24 hours. The peak number of nodes for 123 of the 187 models is less than 15,000, therefore the under-approximation is not triggered on those models (thus ChBFUA behaves exactly like ChBF on them). Using UnderApproxMany, 19 of the remaining 64 models generate the complete state space using under-approximation in less than 24 hours; in these models, whenever the number of node exceeds T_{max}, the algorithm selects a set of nodes $\mathcal{P}^\star$ and deletes $\bigcup_{p \in \mathcal{P}^\star} \mathcal{U}(p)$, until the number of nodes is less than T_{min}. The algorithm adds $\mathcal{S}_{init}$ back and finally generates the complete state space $\mathcal{S}_{rch}$. 15 models out of remaining 64 generate only a partial state space $\mathcal{S}_{part}$.

For the other 30 models out of the remaining 64 models, our algorithm is unable to generate either $\mathcal{S}_{part}$ or $\mathcal{S}_{rch}$ in 24 h. Given enough time, it would always terminate and generate the complete state space or a partial state space. For example, if we increase the running time from 24 to 48 hours, 4 of these 30 models can generate a partial state space. If the model is run indefinitely and the final number of nodes is greater than T_{max}, our approach would in principle eventually generate a partial state space because the number of increasing possible state space sequences is bounded.

6.1 Experimental Results

We compare ChBFUA with ChBF in terms of both memory and time. The more frequently the under-approximation calculates node densities, the slower our algorithm will be. Thus, UnderApproxOne is slower than UnderApproxMany, and we report only the results for the latter.

Figure 14 compares the peak node and time ratios for ChBF and ChBFUA ($Peak_{\text{ChBFUA}}/Peak_{\text{ChBF}}$ and $Time_{\text{ChBFUA}}/Time_{\text{ChBF}}$ respectively) for models where ChBFUA generates $\mathcal{S}_{rch}$. For these models, the final number of nodes is less than T_{max}, otherwise the ChBFUA would not be able to generate the entire state space (whenever the number of nodes is greater than T_{max}, ChBFUA

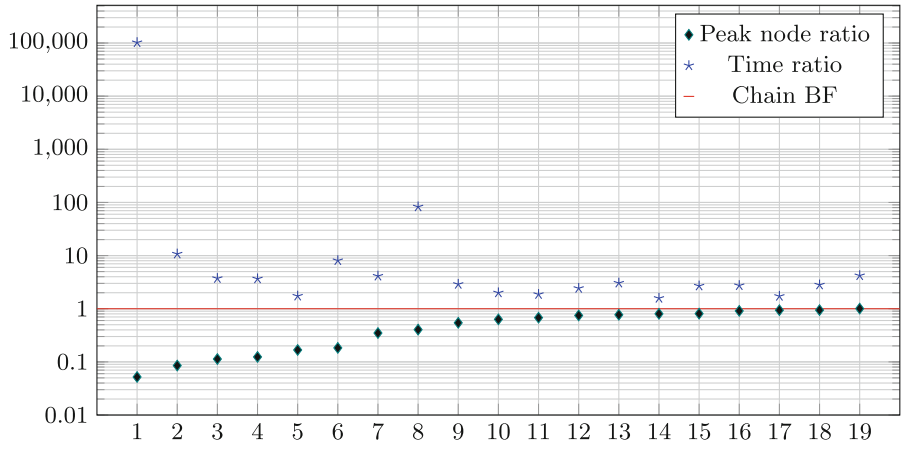


Fig. 14. Time and peak node ratios for the 19 models where ChBFUA generates the entire state space (sorted by increasing peak node ratio).

calls under-approximation to reduce the number of nodes to no more than T_{min}, thus this would eventually result in finding only a partial state space). This experiment shows that:

- The smaller the peak node ratio, the more the under-approximation algorithm is applicable to the model. The peak number of nodes generated by ChBFUA in most cases (except model 19) is less than the peak number of nodes generated by ChBF.
- The peak number of nodes for model 19 in ChBFUA is slightly higher than ChBF, i.e., the peak node ratio is greater than one. This can happen because, after deleting a set of nodes, even just applying the transition relation $\mathcal{T}_\alpha$ for one transition α may result in an MDD with more nodes than the peak number of nodes needed by the ChBF algorithm.
- The runtime ratio in all cases is greater than one, because once the number of nodes reaches T_{max} and ChBFUA invokes UnderApproxMany, it calculates node's density to select and delete nodes until the number of nodes is less than or equal to T_{min}. Calculating the density information and adding back removed states causes ChBFUA to have a higher runtime than ChBF.

Table 2 shows detailed experimental results for models where ChBFUA generates $\mathcal{S}_{rch}$. The more ChBFUA invokes the under-approximation (row "#UA calls"), the larger its runtime is than that of ChBF. Also, in most cases, the fewer times the under-approximation algorithm is invoked, the closer the peak of ChBFUA and peak of ChBF are; this is because it is more likely that T_{max} is close to the peak number of nodes in ChBF.

Figure 15 reports instead the final node and state space ratios for ChBFUA and ChBF ($FinalNode_{\text{ChBFUA}}/FinalNode_{\text{ChBF}}$ and $|\mathcal{S}_{\text{ChBFUA}}|/|\mathcal{S}_{\text{ChBF}}|$, where $FinalNode_{\text{ChBFUA}}$ is the final number of nodes generated by ChBFUA and

Table 2. Results for models where ChBFUA generates the complete state space.

Model#	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
#UA calls	533	137	71	101	27	217	59	423	43	15
Peak nodes ChBF	845,847	348,203	222,344	293,111	130,602	105,738	46,549	70,858	30,065	26,251
Peak nodes ChBFUA)	43,970	29,683	25,162	36,320	21,887	19,379	16,247	28,642	16,330	16,587
runtime ChBF (sec)	1,692	275	443	308	250	1,268	237	392	196	206
runtime ChBFUA (sec)	100,345	2,952	1,649	1,126	434	10,243	978	32,482	567	411
Model#	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	
#UA calls	7	11	5	9	4	6	4	6	28	
Peak nodes ChBF	26,917	24,983	20,213	20,078	18,873	23,875	21,813	17,235	18,773	
Peak nodesChBFUA	18,282	18,674	15,559	16,106	15,251	21,748	20,491	16,245	18,899	
runtime ChBF (sec)	233	31	32	240	61	12	22	41	133	
runtime ChBFUA (sec)	433	76	99	379	165	33	37	115	557	

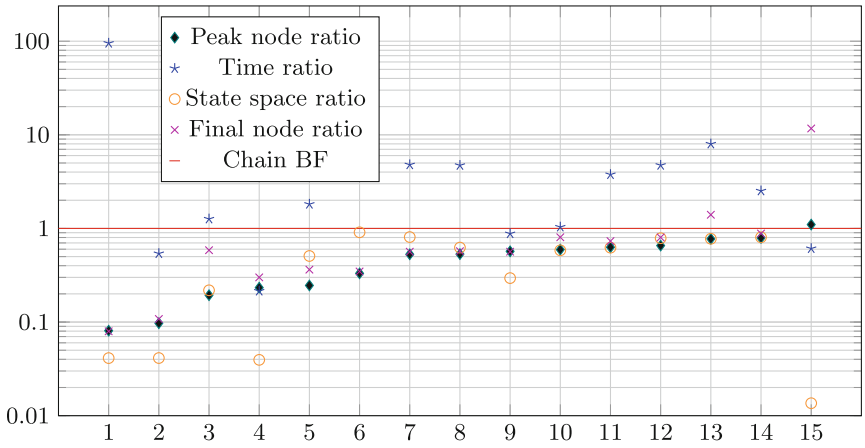


Fig. 15. Runtime, peak node, state space, and final node ratios for models where ChBFUA generates a partial state space (sorted by increasing peak node ratio).

$|\mathcal{S}_{\text{ChBFUA}}|$ is the size of the state space generated by ChBFUA), for models where ChBFUA generates $\mathcal{S}_{part}$. The final number of nodes for most of these models (except model 13 and 15) is greater than T_{max}. The state space ratio is always less than one, since the ChBFUA does generate the complete state space. The final number of nodes generated by ChBFUA in most models is less than the final number of nodes for ChBF, however ChBFUA encodes only a portion of the entire state space. In some models, e.g., 15, the algorithm detects a partial state space faster (time ratio less than one), but the final node ratio is greater than one, indicating that the algorithm is unable to merge nodes to obtain a denser MDD. In these cases, a self-adjusting heuristic could be beneficial.

7 Conclusions and Future Work

We presented a new algorithm for MDDs under-approximation that uses a more precise density than in previously-proposed techniques for BDDs. We demonstrated the soundness of our approach by applying it to the symbolic Petri net state-space generation, where it can compute the entire state space, or possibly a subset of it, with lower memory requirements, at the price of longer runtimes.

Further work is needed towards reducing the number of user-provided parameters. Specifically, we envision a self-adjusting heuristic that automatically chooses and updates (upward or downward) the percentage ψ parameter and the minimum threshold for under-approximation, by self-monitoring the algorithm's own performance (in practical applications, the maximum threshold would instead be likely set to a large value dictated by the amount of available RAM).

Finally, it is worth investigating whether our approach can be adapted to compute an over-approximation. Simply substituting a highest-density node with terminal **1** would result in an over-approximation but, for the monotonically-increasing fixpoint algorithm we use for state-space generation, an unreachable state added by an over-approximation call would never be removed; this is in contrast to a reachable state removed by an under-approximation call, which can always in principle be added back.

References

1. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Comput. **45**(9), 993–1002 (1996). https://doi.org/10.1109/12.537122
2. Bryant: graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
3. Ciardo, G., Jones, R., Marmorstein, R., Miner, A., Siminiceanu, R.: SMART: stochastic model-checking analyzer for reliability and timing. In: Proceedings International Conference on Dependable Systems and Networks, pp. 545- (2002). https://doi.org/10.1109/DSN.2002.1028976
4. Ciardo, G., Marmorstein, R., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. Int. J. Softw. Tools Technol. Transfer **8**(1), 4–25 (2006)
5. Hosseini, S.: memory constrained algorithms for multi-valued decision diagrams. Master's thesis, Iowa State University (2021). https://www.proquest.com/dissertations-theses/memory-constrained-algorithms-multi-valued/docview/2628162662/se-2
6. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. **1**(1), 121–141 (1979). https://doi.org/10.1145/357062.357071
7. Pastor, E., Roig, O., Cortadella, J., Badia, R.M.: Petri net analysis using Boolean manipulation. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 416–435. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58152-9_23

8. Ravi, K., McMillan, K.L., Shiple, T.R., Somenzi, F.: Approximation and decomposition of binary decision diagrams. In: Proceedings of the 35th Annual Design Automation Conference, pp. 445–450. DAC 1998, Association for Computing Machinery, New York, NY, USA (1998). https://doi.org/10.1145/277044.277168
9. Ravi, K., Somenzi, F.: High-density reachability analysis. In: Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, pp. 154–158. ICCAD 1995, IEEE Computer Society, USA (1995)