# Tagged BDDs: Combining Reduction Rules from Different Decision Diagram Types

Tom van Dijk
Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria
Email: tom.vandijk@jku.at

Robert Wille
Institute for Integrated Circuits
Johannes Kepler University Linz, Austria
Email: robert.wille@jku.at

Robert Meolic
Faculty of EE and CS
University of Maribor, Slovenia
Email: robert.meolic@um.si

***Abstract*—Binary decision diagrams are fundamental data structures in discrete mathematics, electrical engineering and computer science. Many different variations of binary decision diagrams exist, in particular variations that employ different reduction rules. For some applications, such as on-the-fly state space exploration, multiple reduction rules are beneficial to minimize the size of the involved graphs. We propose tagged binary decision diagrams, an edge-based approach that allows to use two reduction rules simultaneously. Experimental evaluations demonstrate that on-the-fly state space exploration is an order of magnitude faster using tagged binary decision diagrams compared to traditional binary decision diagrams.**

## I. Introduction

Binary decision diagrams are fundamental data structures which emerged in the '80s as a means to efficiently represent Boolean functions. They now find applications in many areas including logic synthesis, model checking, verification, automated reasoning, reachability analysis, and combinatorics. Section 7.1.4 in Knuth's encyclopedia [17] and a recent survey paper by Minato [24] provide an accessible history of the research activities into binary decision diagrams.

Many variations of binary decision diagrams have been proposed. They differ, e.g., in the type of leaves, decomposition rules, and reduction rules. The existence of these variants motivates dedicated application areas. Original binary decision diagrams (BDDs) are, e.g., applied in model checking and logic synthesis [9], [8], [21], multi-terminal binary decision diagrams (MTBDDs) are, e.g., used to compute properties of probabilistic models [2], [14], [3] and zero-suppressed binary decision diagrams (ZBDDs) are exploited to compactly represent subsets and sparse matrices [23].

For some applications, it could be beneficial to combine characteristics of different binary decision diagram types. In this paper, we consider on-the-fly state space exploration such as conducted in [5], [16]. The efficiency of on-the-fly state space exploration relies on a compact representation of the involved Boolean functions, which initially include a large number of Boolean variables set to 0 (suitably represented by ZBDDs) but, over time, are extended by partial assignments to variables which become redundant (suitably represented by BDDs). Thus far, no solution combines the reduction rules from both BDDs *and* ZBDDs at the same time. Existing methods use either original BDDs or ZBDDs only and thus do not exploit the full potential of these reduction rules.

In the literature, different decomposition rules have been combined using a node-based approach, where nodes store the information about the applied decomposition rule [13]. However, combining BDDs and ZBDDs using this method would not be efficient, as it limits node sharing. Hence, we propose *tagged binary decision diagrams* (TBDDs), an edge-based approach that allows the simultaneous use of two reduction rules. To distinguish which rule is used to remove nodes, we introduce tags on every edge in the graph. We adapt several algorithms to handle both reduction rules and this leads to a more compact representation. We evaluate the proposed TBDDs for on-the-fly state space exploration and observe that they improve upon BDDs by an order of magnitude.

The rest of this paper is structured as follows. We cover the preliminaries in Section II. Section III motivates the new binary decision diagram type by reviewing on-the-fly symbolic state space exploration – an application where reduction rules from both BDDs and ZBDDs are beneficial. We describe the concepts of TBDDs in Section IV. In Section V, we present several algorithms that construct and manipulate TBDDs and discuss how they are used in the considered application. The benefits of TBDDs are illustrated and evaluated in Section VI. Finally, we reflect upon the obtained results in Section VII.

## II. Preliminaries

This paper proposes a type of binary decision diagrams that exploits characteristics of both traditional binary decision diagrams and zero-suppressed binary decision diagrams. We briefly review both types in the following.

### A. Binary Decision Diagrams

A *binary decision diagram* [7] is a rooted directed acyclic graph. There are only two distinct leaves labeled with 0 and 1. Each internal node $v$ is labeled with $\mathrm{var}(v)$ and has two outgoing edges to successors denoted by $\mathrm{low}(v)$ (the *low* successor) and $\mathrm{high}(v)$ (the *high* successor). The edge leading to the root is called a *top edge*. A binary decision diagram is called *ordered* if each variable is encountered at most once on each path from the root to a leaf and the variables are encountered in the same order on all such paths. The *size* of a binary decision diagram is defined by the number of its nodes.

If binary decision diagrams are used to represent Boolean functions, then labels in nodes are Boolean variables and the

ISBN: 978-0-9835678-7-5. 

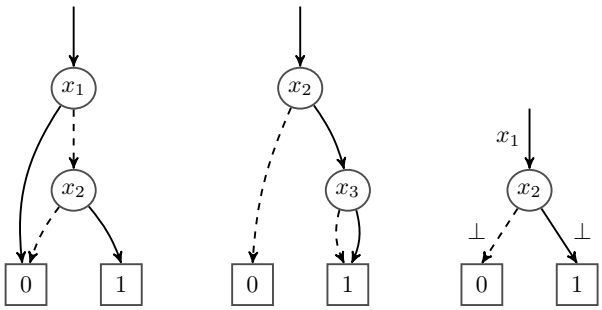
leaves 0 and 1 represent the Boolean constants. The Boolean function represented by the graph with root $v$ is recursively determined using the *decomposition rule* which is based on Shannon's expansion theorem [27], i.e.,

$$f(v) = \overline{\mathrm{var}(v)} \cdot f(\mathrm{low}(v)) + \mathrm{var}(v) \cdot f(\mathrm{high}(v)).$$

Consequently, Boolean functions $f(\mathrm{low}(v))$ and $f(\mathrm{high}(v))$ coincide with $f(v)|_{\mathrm{var}(v)=0}$ and $f(v)|_{\mathrm{var}(v)=1}$, respectively.

The efficiency of binary decision diagrams is achieved by minimizing the structure. This is done by employing several *reduction rules*. The primary rule is that a binary decision diagram may not contain isomorphic subgraphs, i.e., *equivalent nodes* with the same $\mathrm{var}(v)$, $\mathrm{low}(v)$ and $\mathrm{high}(v)$. Traditional BDDs obey an additional rule, which removes so-called *redundant nodes*, i.e., nodes where $\mathrm{low}(v) = \mathrm{high}(v)$ [7]. With ZBDDs, this rule is substituted by one where a node $v$ is called redundant iff $\mathrm{high}(v) = 0$. In both approaches, a decision diagram is *reduced* if it contains neither equivalent nodes nor redundant nodes. Both reduced BDDs and reduced ZBDDs are a canonical representation of Boolean functions and combinatorial sets.

**Example 1.** *Fig. 1 shows three binary decision diagrams representing the Boolean function $f(x_1, x_2, x_3) = \overline{x_1}x_2$. We use dashed lines for low edges and solid lines for high edges. TBDDs are proposed in Section IV.*




(a) BDD (b) ZBDD (c) TBDD

Fig. 1: Different binary decision diagrams representing the same Boolean function.

### B. Complemented Edges

Modern binary decision diagram packages often use complemented edges [6], [25] as an additional mechanism to minimize the graph size. A complemented edge modifies the interpretation of the subgraph to which it points. In this section, we write $\neg v$ to denote that edge pointing to $v$ is complemented, i.e., marked with a complement bit.

Complemented edges on BDDs are interpreted as follows. A complemented edge to 0 is interpreted as 1 (or vice-versa), and a complemented edge to an internal node is interpreted by toggling the complement bit on both successors, i.e.,

$$f(\neg v) := \overline{\mathrm{var}(v)} \cdot f(\neg\,\mathrm{low}(v)) + \mathrm{var}(v) \cdot f(\neg\,\mathrm{high}(v)).$$

It follows that $f(\neg v)$ (complemented edge) is equal to $\overline{f(v)}$ (negation) for BDDs and therefore that computing the negation of a formula is free with BDDs.

With ZBDDs, complemented edges cannot be used such that they coincide with negation. Nevertheless, complemented edges can be used to increase node sharing. Since forwarding the complement bit to the high successor counteracts the "zero-suppressing" property of ZBDDs, the complement bit is only forwarded to the low successor:

$$f(\neg v) := \overline{\mathrm{var}(v)} \cdot f(\neg\,\mathrm{low}(v)) + \mathrm{var}(v) \cdot f(\mathrm{high}(v))$$

Complemented edges allow multiple representations of the same function by toggling the complement bit on the incoming edge and both successors (for BDDs), or on the incoming edge and the low successor (for ZBDDs). A well-known method to ensure a canonical representation is to forbid the complement bit on the low successor, which works well for both BDDs and ZBDDs [6], [23].

### C. Generalizing Reduction Rules

The reduction rules of BDDs and ZBDDs target two *patterns* of nodes that are removed. For BDDs these are nodes with equivalent successors, whereas for ZBDDs these are nodes whose high successors are 0. In general, given a graph in which no variable is skipped, the reduction rule removes all nodes that match a given pattern. Given the successors of a node $v$ as a pair $(\mathrm{low}(v), \mathrm{high}(v))$, we can identify the following *simple* reduction patterns involving only one node:

| (1) | (2) | (3) | (4) | (5) |
|---|---|---|---|---|
| $(k,k) \Rightarrow k$ | $(k,0) \Rightarrow k$ | $(k,1) \Rightarrow k$ | $(0,k) \Rightarrow k$ | $(1,k) \Rightarrow k$ |

Pattern 1 is the rule for BDDs, while pattern 2 is the rule for ZBDDs. If we also allow patterns involving negation, we get 12 patterns [22]. This can be generalized further for patterns that involve multiple nodes, such as representations of $x = x'$. However, this is beyond the scope of this paper.

## III. Motivation

BDDs and ZBDDs offer compact representations for Boolean functions. As they use different reduction rules, their size significantly depends on the kind of Boolean functions they represent. BDDs are particularly suited for functions where adjacent input assignments have the same outcome, whereas ZBDDs are more compact for functions that often evaluate to 1 when many variables are set to 0. This motivates dedicated application areas for either type. BDDs are heavily applied, e.g., in symbolic model checking, while ZBDDs find particular application, e.g., to represent sparse matrices and subsets.

However, applications exist where characteristics from BDDs *and* ZBDDs could both be beneficial. This section reviews such an application. Afterwards, we illustrate how using characteristics from both types could advance the state-of-the-art. This provides the main motivation for the novel binary decision diagram type proposed in Section IV which combines reduction rules of BDDs and ZBDDs.

ISBN: 978-0-9835678-7-5. 

### A. On-the-fly State Space Exploration

Binary decision diagrams are an important data structure in symbolic model checking, which creates models of complex systems to verify that they function according to certain properties or a given specification. Systems are modeled as a set of states $S$ and a transition relation $T \subseteq S \times S$. We encode these sets using their characteristic functions $\mathbf{S}$ and $\mathbf{T}$ (such that $S = \{\, s \mid \mathbf{S}(s) \,\}$ and $T = \{\, (s, s') \mid \mathbf{T}(s, s') \,\}$), which we represent by binary decision diagrams. Since algorithms on binary decision diagrams effectively operate on sets of states rather than individual states, they have successfully been applied to verify systems with a large number of states [9], [8], [28] (infeasible for explicit-state model checking).

A central role in model checking algorithms, such as verifying properties expressed in the modal $\mu$-calculus [18], [4], LTL [10] or CTL [8], [19] is state space exploration. Here, all reachable states in (a part of) the state space are determined starting from a given set of initial states. This is typically conducted by computing the successor states according to the transition relation until a fixed point has been reached.

The model checking toolset LTSMIN [5], [16], [20] offers a framework where transition relations are updated on-the-fly as the model is explored. Initially, LTSMIN does not have knowledge of the transitions in the system. The transition system is explored by learning new transitions via a language-independent interface called PINS,which connects various input languages to model checking algorithms [16]. In PINS, the states of a system are represented by vectors of integers.

### B. Utilizing Characteristics from BDDs and ZBDDs

One of the challenges in on-the-fly state space exploration is that the number of bits required to encode the state space is not known in advance. As LTSMIN uses integers as the fundamental data type, these integers are encoded with, e.g., 16 or 32 bits per integer. However, state variables often hold only few values in the reachable state space, so most bits are always set to 0. Hence, ZBDDs can more effectively represent the state space, as nodes for variables set to 0 are then eliminated [15]. On the other hand, the set of reachable states often includes adjacent states, where some variable $x$ can be either True or False. Such sets are effectively represented with traditional BDDs. Hence, both the reduction rules of ZBDDs and traditional BDDs would significantly reduce the size of the binary decision diagrams.

**Example 2.** *In Fig. 2 we consider one integer variable of a possibly much larger system, encoded using 8 bits. In this example, we have discovered that the variable may hold values* $\{0, 2, 4, 6\}$. *Using ZBDDs would allow for a compact representation for variables* $x_0, x_1, x_2, x_3, x_4, x_7$ *(since these variables would have their high edge to* 0*). In contrast, using BDDs would allow for a compact representation for variables* $x_5, x_6$ *(since these variables would have identical successors). Hence, both the reduction rules of ZBDDs and ordinary BDDs would significantly reduce the size of the representation.*

| $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| | | | A | | B | | C |

Fig. 2: The BDD representing the four states would have nodes with the high edge to 0 for the variables in regions "A" and "C" and with identical successors in region "B".

## IV. Tagged binary decision diagrams

This section introduces a novel binary decision diagram type which addresses the drawback discussed in the previous section. The general idea is that nodes are removed according to two different reduction rules. To distinguish which rule has been applied between two adjacent nodes in the graph, a variable label is used as a *tag* on every edge (note that edge tags have also been used for other purposes, e.g., in [25]). Missing variables *before* the tag are removed according to the first rule, while variables missing *starting from* the tag are removed according to the second rule. Both reduction rules are maximally applied. Overall, this combines the benefits from both decision diagram types and thus leads to a more compact representation. In the following, the concepts are described first. Afterwards, the prototype implementation is described in Section V, while their efficiency is experimentally evaluated in Section VI.

### A. Definition

A *tagged binary decision diagram* (TBDD) is a rooted directed acyclic graph. There are two leaves that are labeled with 0 and 1. Each internal node $v$ is labeled with $\mathrm{var}(v)$ and has two outgoing edges to successors denoted by $\mathrm{low}(v)$ (the *low* successor) and $\mathrm{high}(v)$ (the *high* successor). Like ordinary binary decision diagrams, variables are encountered along each directed path according to a fixed variable ordering and equivalent nodes are forbidden. In addition, edges are labeled with a *tag*, which can be a variable label or $\perp$. Edges to internal nodes are always labeled with a variable label, while edges to leaves may also be labeled with $\perp$. The variable used as a variable label must always be *after* the variable of the source node and *before or equal to* the variable of the target node, if the target node is not a leaf.

TBDDs admit two reduction rules that forbid certain nodes. Examples of these rules are given in Section II-C. In the remainder of this paper, we use the reduction rule of BDDs as the first rule and the reduction rule of ZBDDs as the second rule. Both rules are maximally applied. The tag determines how missing nodes are treated. Informally, the tag $x_{\text{tag}}$ means that all skipped variables *before* $x_{\text{tag}}$ were removed due to the first rule and that all skipped variables *starting from* $x_{\text{tag}}$ were removed due to the second rule. The tag $\perp$ on an edge to a leaf means that all remaining variables were removed due to the first reduction rule.

ISBN: 978-0-9835678-7-5. 

**Example 3.** *Fig. 3a shows a fragment of a TBDD with the variable ordering $x_i < x_j \leq x_k$. Nodes with variable $x_m$ such that $x_i < x_m < x_j$ were removed by the first reduction rule, while nodes with variable $x_n$ such that $x_j \leq x_n < x_k$ were removed by the second reduction rule. Fig. 3b shows a concrete TBDD where the first reduction rule is those of BDDs and the second reduction rule is those of ZBDDs. For the variables $\{x_0, \ldots, x_8\}$, this TBDD represents $\overline{x_1}x_2\overline{x_3}x_4\overline{x_6}x_7\overline{x_8} \vee \overline{x_1}x_2x_3\overline{x_5}\overline{x_6} \vee \overline{x_1}x_2x_3\overline{x_5}x_6\overline{x_8}$. This expression is obtained by looking at each path from the root to leaf 1 and assigning False to all variables that are skipped and equal to or greater than the tag.*

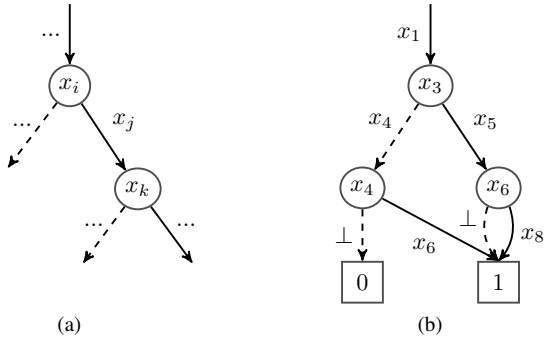



Fig. 3: Examples of tagged binary decision diagrams.

To interpret a BDD as a Boolean function, the variable domain is not explicitly needed, while for ZBDDs and for TBDDs the represented Boolean function depends on the variable domain. In the following, for a given edge and variable domain *Dom* the notations $Dom_1$ and $Dom_2$ are used, where $Dom_1$ denotes the variables in *Dom* before the tag and $Dom_2$ denotes the variables in *Dom* starting from the tag and before the variable of the target node if it is not a leaf. If the tag is $\perp$, then $Dom_1 = Dom$ and $Dom_2 = \emptyset$. For the two reduction rules, we can deduce interpretation functions $Interp_1$ and $Interp_2$. These functions map a set of variables and a Boolean formula to the Boolean formula of the interpretation. An edge to a node or leaf $v$ is interpreted as $Interp_1(Dom_1, Interp_2(Dom_2, f(v)))$.

The interpretation function of the reduction rule of BDDs is simply $Interp(Dom, f) := f$, while the interpretation function of the reduction rule of ZBDDs is $Interp(Dom, f) := \bigwedge_{x \in Dom} \overline{x} \wedge f$. An edge to a node or leaf $v$ in a TBDD using these two rules is interpreted as $\bigwedge_{x \in Dom_2} \overline{x} \wedge f(v)$. Furthermore, edges to the leaf 0 can only have tag $\perp$. Finally, we introduce complemented edges similarly as for ZBDDs. This means that we cannot use complemented edges to compute negation, but at least we can sometimes reuse nodes.

### B. Reduction Rules During Construction

The two reduction rules are maximally applied. However, a special case occurs when nodes that are eliminated by the two rules alternate. In this case, either every last node matching the second rule or every first node matching the first rule must be kept. In the following, we maximally apply the second rule (rule of ZBDDs) and thus some nodes of the first rule (rule of BDDs) are kept to denote these alternating sequences.

**Example 4.** *The binary decision diagram in Fig. 4 has two nodes that have the high edge to 0 and a node with identical successors. Applying the reduction rules removes all nodes with the high edge to 0. The remaining node cannot be eliminated, because it would no longer be possible to distinguish which rule was used to eliminate which node.*

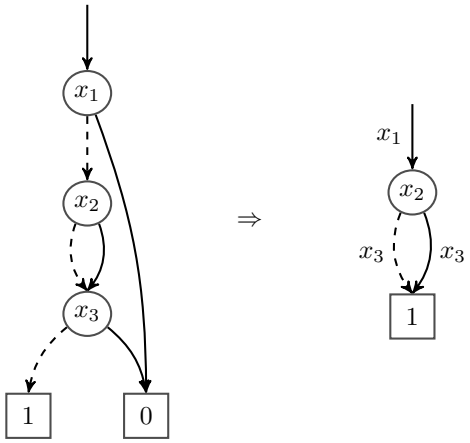



Fig. 4: A TBDD where not all nodes can be eliminated.

Binary decision diagrams are reduced from bottom to top during their construction. Whenever a node is constructed that matches one of the reduction rules, the rule is applied, in the case of the rule of BDDs by simply returning the original low (or high) successor, and in case of the rule of ZBDDs by returning the low successor with an updated tag. However, if we deduce that the low successor already has skipped variables according to the rule of BDDs, then we keep an extra node.

Fig. 5 illustrates the respective reduction rules for TBDDs. The bottom-up reduction is started with a graph in which no variable is skipped. All edges are initially labeled with the same variable as the node that they point to, or $\perp$ if they point to a leaf node. Rules 1 and 2a are applied in the most cases. For the special case where nodes that match both reduction rules alternate, rule 2b ensures that an extra node is created. As we maximally apply the rule of ZBDDs, rule 2b creates a node that matches the rule of BDDs; if we would maximally apply the rule of BDDs, then rule 2b would be modified to accomplish this.

### C. Canonical Representation

TBDDs are, like BDDs and ZBDDs, a canonical representation of Boolean functions, under the condition that the reduction rules are maximally applied. In general, this removes all nodes of the two chosen patterns, except exactly those nodes that are required when rule applications alternate (as discussed above). In the considered case (combining BDDs and ZBDDs), the canonicity can be simply derived by observing the bijection between BDDs, ZBDDs, and TBDDs, i.e., every reduced TBDD matches with a unique reduced BDD (ZBDD)

ISBN: 978-0-9835678-7-5. 

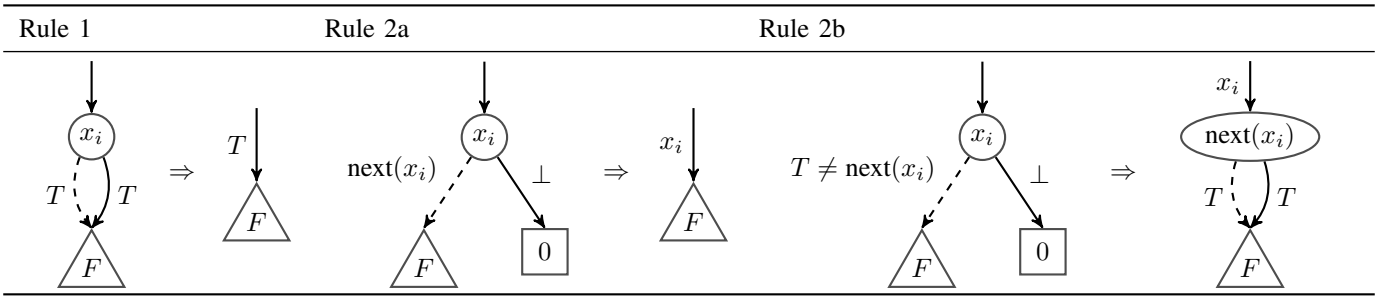



Fig. 5: The reduction rules of a TBDD for bottom-up reduction. Here $T$ stands for any tag and $F$ for any reduced TBDD; next($x$) is the next variable in the variable domain, or $\perp$ if $x$ is the last variable.

with a deterministic conversion procedure, and vice versa. The conversion procedure can be defined as a two step procedure such that the first step is a creation of a graph in which no variable is skipped. Therefore, since BDDs (and ZBDDs) are a canonical representation, so are TBDDs.

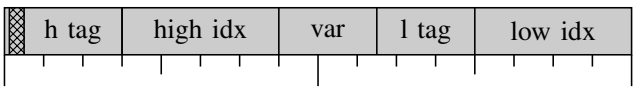
## V. Implementation

Algorithms on TBDDs are not trivial, since they have to deal with various special cases introduced by the two reduction rules. This section first describes the representation of nodes and edges of TBDDs followed by the algorithm `FoaNode` that constructs TBDDs for a given function by finding or adding nodes. Afterwards, the algorithms `ite` and `exists` are described which provide basic operations on TBDDs. Finally, we briefly discuss a more specialized algorithm that is used for the on-the-fly state space exploration.

### A. Representation of Nodes and Edges

We implemented TBDDs in the parallel decision diagram package Sylvan [11], [12]. As Sylvan allocates 16 bytes per node (for performance), we design the internal structure of a TBDD node to fit these constraints. In binary decision diagram implementations, nodes are stored in a *unique table*. This ensures that no equivalent nodes are created. We use 32 bits to store the index of a node in the unique table. This is sufficient to store up to $2^{32}$ nodes, i.e., 96 gigabytes of nodes, excluding overhead costs. For the variable labels and the tags on the edges, we allocate 20 bits. This allows up to 1,048,576 variables; we reserve the highest value to encode $\perp$.

In our implementation, we use a 64-bit integer to encode a tagged edge to a TBDD node. The lowest 32 bits represent the location of the node in the table, and the highest bit stores the complement bit [6]. The TBDD 0 is reserved for the leaf 0 (False), with the complemented edge to 0 for True. The 31 remaining bits provide space to store the 20-bit tag. A TBDD node in memory stores the variable label (20 bits), the low edge index (32 bits), the high edge index (32 bits), the low edge tag (20 bits), the high edge tag (20 bits) and the complement bit of the high edge (1 bit, the first bit below) as follows:




### B. Constructing TBDD Nodes

The core function to obtain nodes is `FoaNode` (find or add a node), which applies the reduction rules presented in Fig. 5 and creates a new node if necessary. This function is given the variable $x_i$ and tagged edges *L* (low) and *H* (high), as well as the next variable $x_i'$ (which can be $\perp$) which is necessary to apply the reduction rules of Fig. 5. The function's result is a tagged edge to a TBDD node. The `FoaNode` algorithm is shown as Alg. 1. It first applies Rule 1 (line 2). If there is a complement on the low edge, we apply the rule that forbids complemented edges on the low edge (line 3). We then apply Rules 2a and 2b (lines 4–6) by comparing the tag on the low edge to the next domain variable. If they are the same, Rule 2a is applied by returning an edge to the given node with the tag according to Rule 2a (line 5). Otherwise, an extra node is created (line 6) and returned with the appropriate tag (line 8). If no reduction rule can be applied, a TBDD node is created via the unique table (line 7) and returned with the appropriate tag (line 8). Note that `find-or-insert` is provided by the unique table and it does not create a new TBDD node if the requested node already exists.

```
1 def FoaNode(x_i, L, H, x_i'):
2     if L = H : return L
3     if comp(L) : return ¬FoaNode(x_i, ¬L, H, x_i')
4     if H = 0 :
5         if tag(L) = x_i' : return settag(L, x_i)
6         else: node ← find-or-insert({x_i', L, L})
7     else:  node ← find-or-insert({x_i, L, H})
8     return settag(node, x_i)
```

**Algorithm 1:** The `FoaNode` method that constructs a TBDD node and applies the reduction rules.

### C. Basic Operations

This section describes several basic TBDD operations. All operations use an operation cache to store subresults, like virtually all binary decision diagram operations. We assume that the reader is familiar with this technique and omit it here. We also assume that the involved TBDDs are interpreted in the same variable domain as the result. This variable domain

ISBN: 978-0-9835678-7-5. 

```
1  def cofactors(F, x_i, x'_i):
2      if F = 0 : return 0, 0
3      elif x_i < tag(F) : return F, F
4      elif F = 1 : return F, 0
5      elif x_i < var(F) :
6          F_0 ← low(F)
7          if F_0 ∉ {0,1} ∧ var(F_0) = x'_i ∧ low(F_0) = high(F_0) :
8              F_0 ← low(F_0)
9          return F_0, 0
10     else: return low(F), high(F)
```

**Algorithm 2:** Fragment used by TBDD algorithms to compute the cofactors $F_0$ and $F_1$, given a domain variable $x_i \leq \text{var}(F)$ and the next domain variable $x'_i$ (or $\perp$ if $x_i$ is the last).

```
1  def ite(F, G, H):
2      if F = 1 : return G
3      if F = 0 : return H
4      if G = H : return G
5      t ← min(tag(F), tag(G), tag(H))
6      if tag(F) = tag(G) = tag(H) :
7          v ← min(var(F), var(G), var(H))
8      else: v ← t
9      F_0, F_1 ← cofactors(F, v, next(v))
10     G_0, G_1 ← cofactors(G, v, next(v))
11     H_0, H_1 ← cofactors(H, v, next(v))
12     r ← FoaNode(v, ite(F_0, G_0, H_0), ite(F_1, G_1,
         H_1), next(v))
13     if t ≠ v : r ← FoaNode(t, r, 0, v)
14     return r
```

**Algorithm 3:** The implementation of the algorithm `ite`.

is given as an additional parameter in the operations, but we omit it here in the interest of clarity and brevity.

Many binary decision diagram operations first find a *pivot variable*, typically the topmost variable of the root nodes of the parameters, then recursively compute the subresults of the operation on the cofactors of the parameters obtained by setting the pivot variable to False or True, and finally compute the result by creating a node with `FoaNode`.

Computing the cofactors is straightforward and summarized in Alg. 2. Note that this is *not* a "generic cofactor" operation; just a helper method for the recursion step. This method returns $F|_{x_i=0}$ and $F|_{x_i=1}$ for a given TBDD $F$ with top variable $x_i$ and also compensates for the application of Rule 2b. We use `low` and `high` to obtain the low and high successor of $F$, where `low` also applies any complement on $F$ to the low edge. After checking whether $F$ is an edge to 0 (line 2), we check whether Rule 1 was applied (line 3). If not, we check if Rule 2 was applied to leaf 1 (line 4). If not, then we check if Rule 2 was applied to the node (line 5) and if so, we check if the node is a redundant node inserted by Rule 2b (line 7) and return the appropriate result (lines 6–9). Finally, at line 10 it is established that no reduction rule was used and we return the successors of the TBDD.

```
1  def exists(F, x⃗):
2      if F = 0 ∨ (F = 1 ∧ tag(F) = ⊥) : return F
3      if x⃗ = ∅ : return F
4      while var(x⃗) < tag(F) :
5          x⃗ ← x⃗ \ {var(x⃗)}
6          if x⃗ = ∅ : return F
7      v ← min(var(F), var(x⃗))
8      F_0, F_1 ← cofactors(F, v, next(v))
9      x⃗' ← x⃗ \ {v}
10     if v < var(F) : res ← exists(F_0, x⃗')
11     elif v = var(x⃗) : res ← or(exists(F_0, x⃗'),
         exists(F_1, x⃗'))
12     else: res ← FoaNode(v, exists(F_0, x⃗'),
         exists(F_1, x⃗'), next(v))
13     if tag(F) < v : res ← FoaNode(tag(F), res, 0, v)
14     return res
```

**Algorithm 4:** The implementation of the algorithm `exists`.

See Alg. 3 for the implementation of the well-known `if-then-else` operation. Given three TBDDs representing $f$, $g$ and $h$, this algorithm computes "if $f$ then $g$ else $h$". We use `tag` and `var` to obtain the tag of an edge and the variable of the root node. The algorithm first tries to apply the trivial cases (lines 2–4). We compute the topmost tag (line 5) and then we determine the pivot variable $v$. Variables that are in $Dom_1$ of all three parameters are in $Dom_1$ of the result. Variables that are in $Dom_2$ of *all* three parameters are in $Dom_2$ of the result. We perform recursion on the first variable that is not in $Dom_1$ of all parameters or in $Dom_2$ of all parameters. This variable is equal to the lowest variable if all tags are the same, or the lowest tag if this is not the case. The cofactors are computed at lines 9–11 and the recursion is performed at line 12, where also the result is computed. If there are variables in $Dom_2$ of the result (which is true if $t \neq v$) then we use `FoaNode` in a special way to compute the result. This ensures the correct application of Rule 2b when the result has variables in $Dom_1$.

We also implement existential quantification as in Alg. 4. This operation existentially quantifies all given variables $\vec{x}$ from the input TBDD $F$. We check for the trivial cases at lines 2–3. We then skip all variables in $\vec{x}$ that are in $Dom_1$ of $F$ (lines 4–6). Variables that are in $Dom_2$ and before $\text{var}(\vec{x})$ will be in $Dom_2$ of the result. As the pivot variable we select the first variable of $F$ and $\vec{x}$ (line 7). We obtain the cofactors and if the chosen pivot variable is in $\vec{x}$ we remove it in $\vec{x}'$ that we use for the recursion (lines 8–9). Now there are three cases. If the pivot variable is in $Dom_2$, then we just compute the result based recursively on $F_0$, as $F_1$ is 0 (line 10). Otherwise, the pivot variable is the variable of the TBDD node. Now either it is also in $\vec{x}$ and we perform recursion as usual and compute the disjunction of the two results, which is how existential quantification is computed (line 11), or it is not in $\vec{x}$ and we perform recursion as usual and compute the node of the result (line 12). Finally, we update $Dom_2$ of the result as discussed above (line 13) and return the result (line 14).

ISBN: 978-0-9835678-7-5. 

As is clear from the above discussions, we must consider for all variables whether they are in $Dom_1$ or $Dom_2$ of the parameters. This makes these algorithms more complex, but overall improves the efficiency by exploiting the reductions.

### D. Computing Successors

Besides "typical" decision diagram operations, we additionally implemented an operation dedicated for the LTSMIN application motivated in Section III, namely `relnext` which applies a transition relation to a set in order to compute the successors (combined with variable renaming). An additional challenge is that LTSMIN partitions the transitions into different transition groups. This has the advantage that each transition group only affects a part of the state vector. As a consequence, the `relnext` operation cannot assume that both parameters are defined on the same variable domain; rather, it handles various special cases introduced by the difference in the variable domains. In the interest of space, we cannot treat `relnext` here but refer the interested reader to the publicly available source code (see next section).

## VI. Experimental Evaluation

This section evaluates the ideas proposed in this paper. The evaluation is performed based on the BEEM database of models [26]. In Sec. VI-A, we study the impact of different reduction rules on the size of the graphs at each iteration in state space exploration. In Sec. VI-B, we evaluate the performance of state space exploration using either BDDs or TBDDs. We use LTSMIN to perform on-the-fly state space exploration, using the FORCE algorithm for selecting a variable reordering [1] and using the standard breadth-first-search strategy to explore the state space. All experimental data and the scripts required to reproduce them are available online via http://fmv.jku.at/tbdd.

### A. Impact of Reduction Rules

We modify LTSMIN to write the BDDs of the explored states and of all transition relations to disk at every iteration of the on-the-fly state space exploration. We compute the number of nodes required for the explored states and the number of nodes required for all transition relations, when removing no redundant nodes, when removing nodes according to the five rules from Sec. II-C (without using complemented edges) and when using the proposed TBDD type.

Since computing these sizes costs a lot of time, we only perform this analysis on a subset of the full benchmark set. Our hypothesis in Sec. III was that, as the state space exploration progresses, the contribution from the $(k, k) \Rightarrow k$ rule would increase. We found that this is not the case for the models that we checked. See the top row of Fig. 6 for a representative model (`at.1`). The top lines in the graphs represent rules 3 and 5 and the case where no rule is applied. They appear to have no significant effect. The bottom lines represent rule 2 (ZBDD, dots) and the TBDD representation (long dashes). These rules have the most significant effect. The middle lines represent rule 1 (BDD, dashes) and rule 4 (dots and dashes) Rules 1 and 4 have a greater impact for the transition relation than for the set of explored states.

### B. On-the-fly State Space Exploration

We look at the performance of on-the-fly state space exploration on the BEEM benchmark database, using either standard BDDs (16 bits per state variable) or TBDDs. We set the timeout for the experiments to $1200$ seconds, i.e., $20$ minutes. The experiments are run on a 48-core AMD Opteron machine. We set the number of cores to either $1$ or $48$, so we can also measure the effect of the parallelism of Sylvan.

For $219$ models, no experiment timed out. For these models, the results are summarized by the following table, where the time is the sum of all models, and the number of nodes is the size of the BDD that represents the set of visited states.

| | BDD | TBDD |
|---|---|---|
| Time $1$ core | 24504 sec. | 6453 sec. |
| Time $48$ cores | 14672 sec. | 1075 sec. |
| #Nodes in the visited set | 59,503,837 | 5,922,973 |

See further Fig. 6 for a comparison of the time and the number of nodes for the models, using just $1$ worker. We see that TBDDs are approximately an order of magnitude faster and smaller. Finally, the highest parallel speedup we obtain when using TBDDs was $32.4\times$ (with $48$ cores) for the model `rushhour.3`.

## VII. Conclusions

BDDs and ZBDDs offer compact representations of Boolean functions which is crucial for many applications. Due to the different reduction rules, their effectiveness strongly depends on the type of functions. The proposed TBDDs simultaneously apply reduction rules of both types which in general results in smaller graphs for a broader set of applications.

We studied the benefits of TBDDs for on-the-fly state space exploration using the BEEM database of models. We obtained a significant improvement compared to ordinary BDDs, although our analysis showed that the contribution of the reduction rule of BDDs was less than expected. Our implementation of TBDDs also resulted in a similarly high parallel scalability as has been obtained for BDDs in the past.

We expect that improvements and tuning of the TBDD operations may result in a better performance, especially considering that these operations are complex and therefore give various opportunities to be optimized. Furthermore, we believe that looking at other benchmark models and application areas would be insightful. Tagged binary decision diagrams can also be applied using other reduction rules, like those mentioned in this paper, or rules involving multiple nodes, or simply by exchanging the order of the rules that we applied here. Furthermore, implementation of other operations like dynamic variable reordering may be challenging, as the interaction between the reduction rules of TBDDs and the popular sifting algorithm is probably complex. These are several ideas that may inspire future research on TBDDs.

## Acknowledgment

The authors acknowledge the support of FWF, NFN Grant S11408-N23 (RiSE) and of the COST Action IC1405.

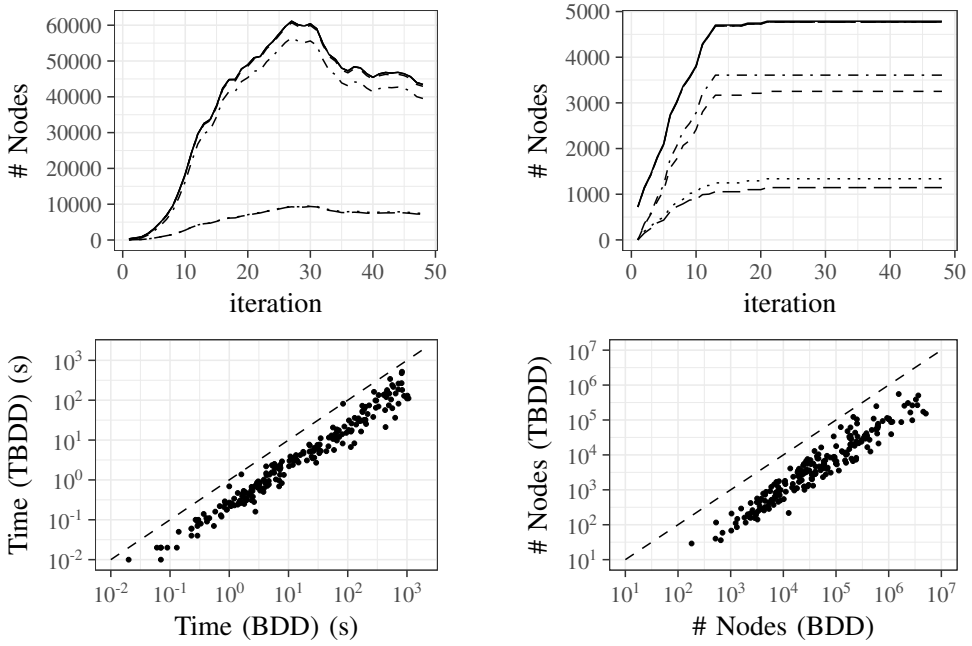



Fig. 6: Results of experimental evaluation. The first row shows the evolution of graph sizes of the set of explored states (left) and the transition relations (right) per iteration (see Sec. VI-A). The second row compares BDDs with TBDDs.

## REFERENCES


[1] Aloul, F.A., Markov, I.L., Sakallah, K.A.: FORCE: a fast and easy-to-implement variable-ordering heuristic. In: VLSI 2003. pp. 116–119. ACM (2003)

[2] Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. Formal Methods in System Design 10(2/3), 171–206 (1997)

[3] Baier, C., Clarke, E.M., Hartonas-Garmhausen, V., Kwiatkowska, M.Z., Ryan, M.: Symbolic model checking for probabilistic processes. In: Automata, Languages and Programming 1997. LNCS, vol. 1256, pp. 430–440. Springer (1997)

[4] Biere, A.: $\mu$cke - Efficient $\mu$-Calculus Model Checking. In: CAV 1997. pp. 468–471. LNCS, Springer (1997)

[5] Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and Symbolic Reachability. In: CAV. LNCS, vol. 6174, pp. 354–359. Springer (2010)

[6] Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: DAC. pp. 40–45 (1990)

[7] Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers C-35(8), 677–691 (8 1986)

[8] Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., Dill, D.L.: Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13(4), 401–424 (4 1994)

[9] 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)

[10] Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. Formal Methods in System Design 10(1), 47–71 (1997)

[11] van Dijk, T.: Sylvan: Multi-core Decision Diagrams. Ph.D. thesis, University of Twente (7 2016)

[12] van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. International Journal on Software Tools for Technology Transfer pp. 1–22 (2016)

[13] Drechsler, R., Becker, B.: Ordered Kronecker functional decision diagrams-a data structure for representation and manipulation of Boolean functions. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17(10), 965–973 (10 1998)

[14] Fujita, M., McGeer, P.C., Yang, J.C.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods in System Design 10(2/3), 149–169 (1997)

[15] Haji Ghasemi, M.: Symbolic model checking using Zero-suppressed Decision Diagrams. Master's thesis, University of Twente, Dept. of C.S. (11 2014)

[16] Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: High-Performance Language-Independent Model Checking. In: TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer (2015)

[17] Knuth, D.E.: The Art of Computer Programming, vol. 4A. Addison-Wesley, Upper Saddle River, New Jersey (2011)

[18] Kozen, D.: Results on the propositional mu-calculus. Theor. Comput. Sci. 27, 333–354 (1983)

[19] Marrero, W.: Using BDDs to decide CTL. In: TACAS 2005. pp. 222–236 (2005)

[20] Meijer, J., Kant, G., Blom, S., van de Pol, J.: Read, Write and Copy Dependencies for Symbolic Model Checking. In: HVC. pp. 204–219 (2014)

[21] Meinel, C., Somenzi, F., Theobald, T.: Linear sifting of decision diagrams and its application in synthesis. IEEE Trans. on CAD 19(5), 521–533 (2000)

[22] Meolic, R., Brezočnik, Z.: Universal decomposition rule for OBDD, OFDD, and 0-sup-BDD. Tech. rep., University of Maribor (2016), https://dk.um.si/IzpisGradiva.php?id=64403

[23] Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proceedings of the 30th international Design Automation Conference. pp. 272–277. DAC '93, ACM, New York, NY, USA (1993)

[24] Minato, S.: Techniques of BDD/ZDD: Brief History and Recent Activity. IEICE Transactions 96-D(7), 1419–1429 (2013)

[25] Minato, S., Ishiura, N., Yajima, S.: Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In: Proceedings of the 27th ACM/IEEE Design Automation Conference. pp. 52–57. DAC '90, ACM, New York, NY, USA (1990)

[26] Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: SPIN. pp. 263–267. Springer-Verlag, Berlin, Heidelberg (2007)

[27] Shannon, C.E.: A Symbolic Analysis of Relay and Switching Circuits. Transactions of the American Institute of Electrical Engineers 57(12), 713–723 (12 1938)

[28] Yoneda, T., Hatori, H., Takahara, A., Minato, S.: BDDs vs. Zero-Suppressed BDDs: for CTL Symbolic Model Checking of Petri Nets. In: FMCAD. pp. 435–449 (1996)


ISBN: 978-0-9835678-7-5. 