Model-checking with Edge-valued Decision Diagrams

Pierre Roux[1], Radu I. Siminiceanu[2]

[1] École Normale Supérieure de Lyon, France pierre.roux@ens-lyon.fr
[2] National Institute of Aerospace, Hampton, Virginia 23666, USA. radu@nianet.org

Abstract. We describe an algebra of Edge-Valued Decision Diagrams (EVMDDs) to encode arithmetic functions and its implementation in a model checking library along with state-of-the-art algorithms for building the transition relation and the state space of discrete state systems.

We provide efficient algorithms for manipulating EVMDDs and give upper bounds of the theoretical time complexity of these algorithms for all basic arithmetic and relational operators. We also demonstrate that the time complexity of the generic recursive algorithm for applying a binary operator on EVMDDs is no worse than that of Multi-Terminal Decision Diagrams.

We have implemented a new symbolic model checker with the intention to represent in one formalism the best techniques available at the moment across a spectrum of existing tools: EVMDDs for encoding arithmetic expressions, identity-reduced MDDs for representing the transition relation, and the saturation algorithm for reachability analysis. We compare our new symbolic model checking EVMDD library with the widely used CUDD package and show that, in many cases, our tool is several orders of magnitude faster than CUDD.

1 Foreword

This is the report of an internship I (Pierre Roux) made at National Institute of Aerospace from January 26th to April 16th 2009 under the direction of Radu Siminiceanu. This was part of my first year of master (M1).

2 Introduction

Binary decision diagrams (BDD) [4] have revolutionized the reachability analysis and model checking technology. Arithmetic decision diagrams [3], also called Multi Terminal Binary Decision Diagrams (MTBDD) [9] are the natural extension of regular BDDs to arithmetic functions. They take advantage of the symbolic encoding scheme of BDDs, but functions with large co-domains do not usually have a very compact representation because there are less chances for suffixes to be shared.

Edge-valued decision diagrams have been previously introduced, but only scarcely used. An early version, the edge valued binary decision diagrams (EVBDD) [14,13], is particularly useful when representing both arithmetic and logic functions, which is the case for discrete state model checking. However, EVBDD have only been applied to rather obscure applications: computing the probability spectrum and the Reed-Muller spectrum of (pseudo)-Boolean functions.

Binary Moment Diagrams [5] were designed to overcome the limitations of BDDs/EVBDDs when encoding multiplier functions. However, their efficiency seems to be limited only to this particular type of functions. A new canonization rule for edge-valued decision diagrams enabling them to encode functions in $\mathbb{Z} \cup \{+\infty\}$ was introduced in [7] along with an extension to multi-way diagrams (MDD) [12], but, again, this was applied to a very specific task, of finding minimum length counterexamples for safety properties. Later, EVMDDs have been also used for partial reachability analysis.

In this paper we first present a theoretical comparison between EVMDDs and MTMDDs for building the transition relation of discrete state systems before dealing with an implementation in a model checker along with state-of-the-art algorithms for state space construction.

3 Background

3.1 Discrete-state Systems

A discrete–state model is a triple (S, S_0, T), where the discrete set S is the *potential state space* of the model; the set $S_0 \subseteq S$ contains the *initial states*; and $T : S \to 2^S$ is the *transition function* specifying which states can be reached from a given state in one step, which we extend to sets: $T(X) = \bigcup_{i \in X} T(i)$.

We consider structured systems modeled as a collection of K *submodels*. A (global) system state i is then a K-tuple $(i_K, \ldots, i_1)$, where i_k is the *local state* for submodel k, for $K \geq k \geq 1$, and S is given by $S_K \times \cdots \times S_1$, the cross–product of K local state spaces S_k, which we identify with $\{0, \ldots, n_k - 1\}$ since we assume that S is finite. The *(reachable) state space* $R \subseteq S$ is the smallest set containing S_0 and closed with respect to T, i.e.

$$R = S_0 \cup T(S_0) \cup T(T(S_0) \cup \cdots = T^*(S_0).$$

Thus, R is least fixpoint of function $X \mapsto S_0 \cup T(X)$.

3.2 Symbolic State–space Generation: Breadth–first vs. Saturation

The traditional approach to generate the reachable states of a system is based on a breadth–first traversal, as derived from classical fixed–point theory, and applies a monolithic T (even when encoded as $\bigcup_{e \in E} T_e$), after d iterations, the currently–known state space contains exactly all states whose distance from any state in S_0 is at most d. However, recent advances have shown that non–BFS, guided, or chaotic exploration can result in a better iteration strategy.

An example is the *saturation* algorithm introduced in [6], which exhaustively *fires* (explores) all events of E_k in an MDD node at level k[3], thereby greedily bringing the node to its final "saturated" form.

3.3 Decision Diagrams

We assume implicitly that the decision diagrams are *ordered*, i.e. the variables labeling nodes along any path from the root must follow the order $x_K, \ldots, x_1$. Ordered DDs can be either *reduced* (no duplicate nodes and no node with all edges pointing to the same node, but edges possibly spanning multiple levels) or *quasi–reduced* (no duplicate nodes, and all edges spanning exactly one level), either form being *canonical*.

We adopt the extension of BDDs to integer variables, i.e., multi–valued decision diagrams (MDDs) [12]. MDDs are often more naturally suited than BDDs to represent the state space of arbitrary discrete systems, since no binary encoding must be used to represent the local states for level k when $n_k > 2$. An even more important reason to use MDDs is that they allow us to better exploit the *event locality* present in systems exhibiting a globally–asynchronous locally–synchronous behavior.

4 EVMDDs

4.1 Definition

Definition 1. *An EVMDD, or* EVMDD edge*, on a group $(G, *)$, is a pair $A = \langle v, n \rangle$ where $v \in G$ will also be noted $A.\mathrm{val}$ and n, also noted $A.\mathrm{node}$, is a node.*

A node *n is either the unique terminal node $\langle 0, e \rangle$ where e is the identity element of G or a pair $\langle k, p \rangle$ where $1 \leq k \leq K$ and p is an array of edges of size n_k. The first element of the pair will be denoted $k = n.\mathrm{level}$ and, when relevant, the element of index i in the array will be denoted by $n[i]$.*

[3] T is then encoded as a disjunction $\bigcup_{e \in E} T_e$ of events e and E is then divided in $\bigcup_{1 \leq k \leq K} E_k$ with each E_k grouping events not depending on submodels above k and not affecting them.

Additionally, the notation $n[i_k, \ldots, i_{k'}]$ is used as a shortcut for $n[i_k] \ldots [i_{k'}]$.node.

Definition 2. *An* ordered *EVMDD is an EVMDD in which every node n satisfies*

$$\forall i \in S_{n.\text{level}} \,.\, n[i].\text{node.level} < n.\text{level}$$

As already mentioned, we only consider ordered EVMDDs, the canonicity of unordered EVMDD being significantly more complex to establish, as well as their manipulation being a lot more challenging.

Example 1. Graphs are a convenient representation for EVMDDs. For ordered EVMDDs, the graph directed by node levels is acyclic. Graphically, the terminal node is represented by a circle at bottom and internal nodes are drawn above according to their level, with all edges pointing down to children nodes. Examples of graph representation of EVMDDs are given in Figure 1.

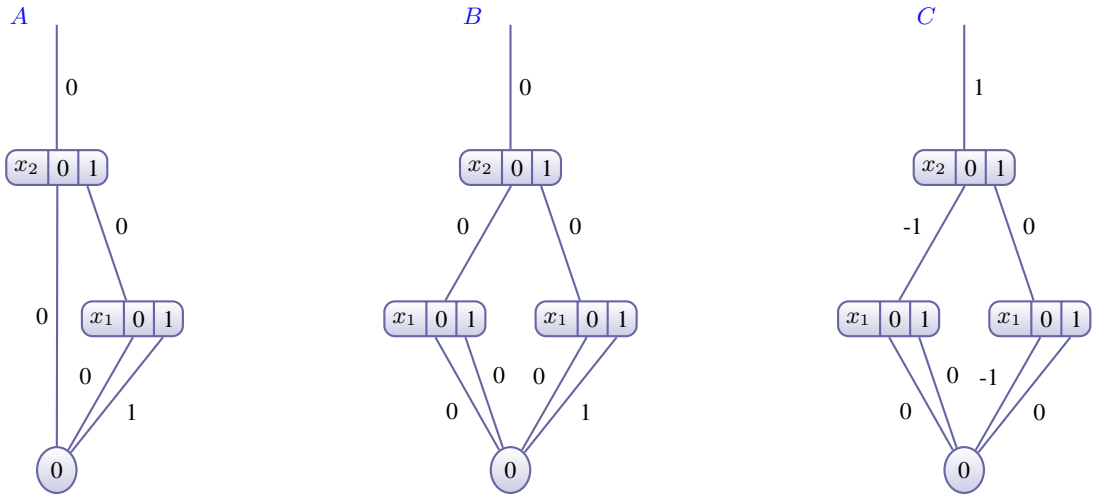

Fig. 1. Three EVMDDs on group $(\mathbb{Z}, +)$ representing the same function $f : \{0,1\}^2 \to \mathbb{Z}, (x_2, x_1) \mapsto x_2 \times x_1$.

Definition 3. *Given a node n with n.level $= k$ and $i_k, \ldots, i_{k'} \in S_k \times \ldots \times S_{k'}$, we define $n(i_k, \ldots, i_{k'})$*

$$n(i_k, \ldots, i_{k'}) = \begin{cases} n[i_k].\text{val} & \text{if } n[i_k].\text{node.level} < k' \\ n[i_k].\text{val} * n[i_k].\text{node}(i_{n[i_k].\text{node.level}}, \ldots, i_{k'}) & \text{if } n[i_k].\text{node.level} \geq k' \end{cases}$$

This allows the definition of the *represented function* for any EVMDD A as

$$\begin{aligned} f : \quad & S \to G \\ & (i_K, \ldots, i_1) \mapsto A.\text{val} * A.\text{node}(i_{A.\text{node.level}}, \ldots, i_1) \end{aligned}$$

In other words, $n(i_k, \ldots, i_{k'})$ is the repetitive application of law $*$ on edge values along the path going down from node n and following directions given by $(i_k, \ldots, i_{k'})$. Hence $f(i_K, \ldots, i_1)$ is $*$ on a path from the root to the terminal node of A.

In this setup, every EVMDD A represents a function $f : S \to G$. The reciprocal is also true: given a function f, an EVMDD A representing f can be built by setting the values of all edges of A to the identity element e of G, except those pointing to the terminal node, which take proper values $f(i_k, \ldots, i_1)$ according to the incoming path leading to it.

Example 2. In Figure 1, the EVMDD B is built from $f : \{0,1\}^2 \to \mathbb{Z}, (x_2, x_1) \mapsto x_2 \times x_1$. according to the method explained above.

Definition 4. *A* reduced *EVMDD contains no duplicate or redundant nodes. A redundant node n has all outgoing edges identical*

$$\forall i, j \in S_{n.\text{level}} \,.\, n[i] = n[j]$$

Definition 5. *A* quasi-reduced *EVMDD contains no duplicate nodes and all internal nodes n have all the descendants on the level below*

$$\forall i \in S_{n.\text{level}} \,.\, n[i].\text{node.level} = n.\text{level} - 1$$

From any EVMDD A, we can build a reduced EVMDD representing the same function by just deleting nodes with all children identical and redirecting the incoming edges to its unique descendant. Similarly, from any EVMDD A, a quasi reduced EVMDD can be built by adding nodes with all children identical on edges spanning multiple levels.

Example 3. In Figure 1 on the preceding page A is reduced, whereas B and C are quasi-reduced.

For the sake of simplicity we will only consider quasi-reduced EVMDDs in the following discussion. However, proofs and algorithms for the reduced version are very similar, only slightly more evolved in order to deal with edges skipping levels. We will turn back to reduced EVMDDs only for implementation since they are never larger in size than their quasi-reduced counterpart, hence could have more efficient algorithms.

As can be seen in the previous example, even when restricting to quasi-reduced diagrams, the EVMDD representation of a function f may not be uniquely defined.

Definition 6. *A* canonical node *is either the terminal node or a node n such that $n[0].\text{val} = e$.*

A canonical EVMDD *is an EVMDD in which all nodes are canonical.*

It can be proved that for every function f, there exists a unique canonical EVMDD representing it [7].

In the following, EVMDDs are assumed to be canonical.

4.2 Extensions

EVMDDs can be used even when the algebraic structure is not a group. [7] offers a canonization rule for $\mathbb{N} \cup \{+\infty\}$ and $(\mathbb{Z}, \times)$ can be handled with the canonization rule "$\gcd\{n[i].\text{val} \mid i \in S_{n.\text{level}}\} = 1$ and $(n[0].\text{val}, \ldots, n[n_{n.\text{level}}].\text{val}) \geq_{\text{lex}} 0$".

It is also interesting to notice that EVMDDs are just a generalization of binary decision diagrams with complemented edges, as presented for example in [11]. Indeed, they are edge-valued diagrams on $G = \mathbb{Z}/2\mathbb{Z}$ and complemented (respectively not complemented) edges corresponding to value 1 (respectively 0).

5 EVMDDs compared to MTMDDs

MTBDDs are commonly used in model checking to build the transition relation of discrete-state systems. In this section we show that EVMDDs are at least as suited to that purpose and oftentimes significantly better. In the remainder of this section, we pick, often without loss of generality, $(G, *) = (\mathbb{Z}, +)$.

5.1 Space Complexity

As stated in section 2.2.6 of [13].

Theorem 1. *For any function f, the number of nodes of the EVMDD representing f is at most the number of nodes of the MTMDD representing the same function f.*

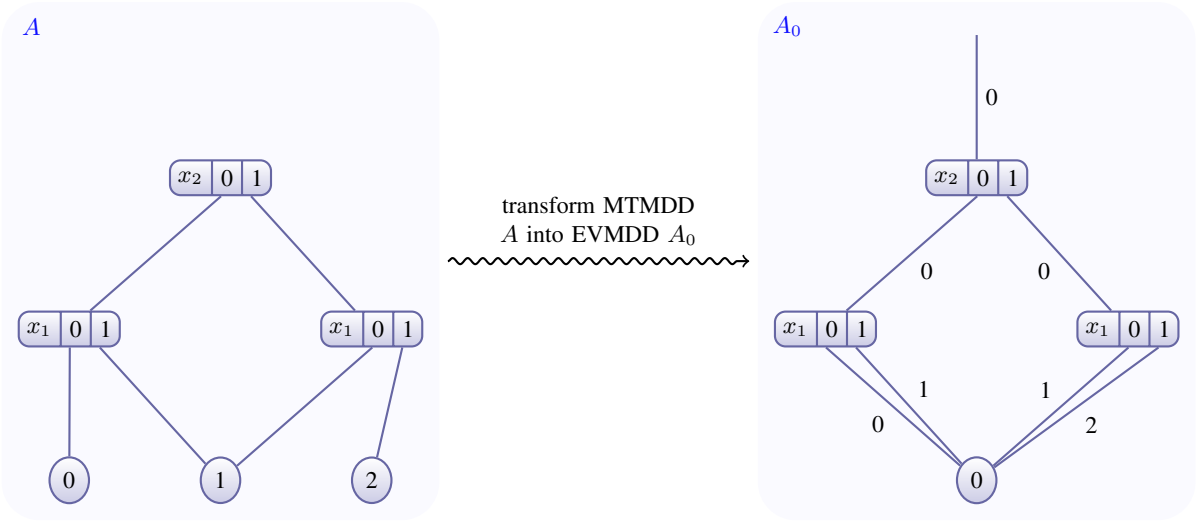

Fig. 2. Building the EVMDD A_0 (right) from MTMDD A (left)

Proof. Let A be the MTMDD representing f. From A we construct the EVMDD (not in canonical form) A_0 by replacing each edge from level 1 to a terminal with value v with an edge with value v to the unique terminal node 0 and associating value 0 to all other edges (see Figure 2 for an example)[4]. Then, we iteratively compute the EVMDD A_k, for each k from 1 to n, through the following process:

- for each node n at level k, subtract $n[0].\mathrm{val}$ from all outgoing edges and add this value to all incoming edges;
- merge all duplicate nodes at level k (by duplicate nodes we mean two nodes having edges x_i holding same value and pointing to same children for each i in the range of variable x_k).

See Figure 3 for an example.

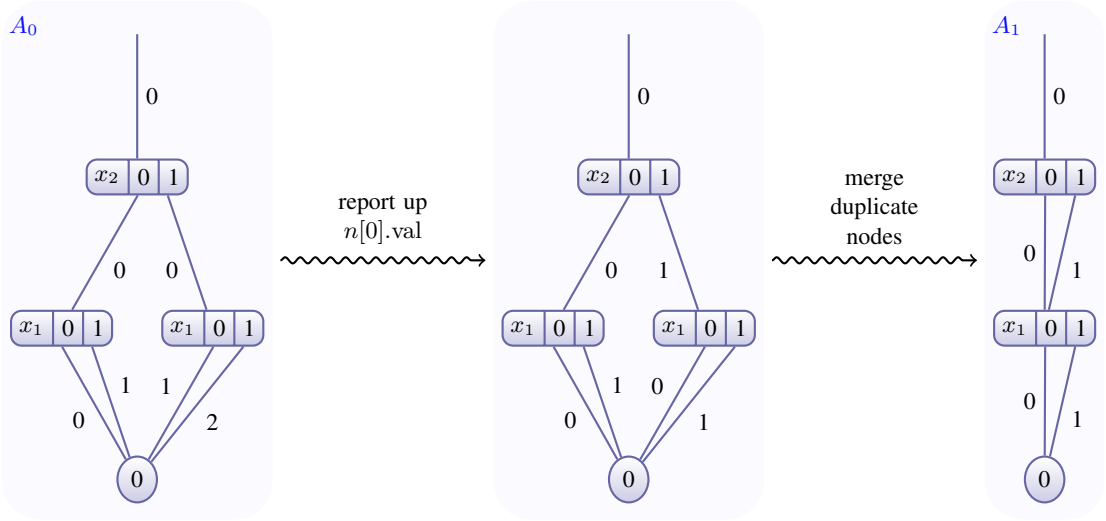

Fig. 3. Constructing EVMDD A_1 (right) from EVMDD A_0 (left)

To prove that A_k and A represent the same function, it is sufficient to see that A and A_0 represent the same function and that the iterative transformation preserves the sum of values on any path $(i_K, \ldots, i_1)$

[4] This process is similar to the one used in section 4.1 on page 2 to prove that every function can be represented by an EVMDD.

from the root of the diagram to the unique terminal node (plus the value of the root's incoming edge)

$$A_k.\text{val} + A_k.\text{node}(i_K, \ldots, i_1) = A_{k-1}.\text{val} + A_{k-1}.\text{node}(i_K, \ldots, i_1)$$

Since A_K is in canonical form and since for each k, the number of nodes of A_k is at most the number of nodes of A_{k-1}, we can conclude that the size of an EVMDD is never larger than that of the corresponding MTMDD. □

This doesn't prove that EVMDDs always require less memory than MTMDDs since they need extra space to store the edge values, but no worse than up to a small factor [5]. On the other hand, EVMDDs can be exponentially better than MTMDD in some cases. For example, the function

$$\begin{aligned} \{0, B-1\}^K &\to \mathbb{Z} \\ (i_K, \ldots, i_1) &\mapsto \sum_{k=1}^{K} i_k B^{k-1} \end{aligned}$$

where ($B \geq 2$), requires $\dfrac{B^{K+1}-1}{B-1}$ nodes in its MTMDD representation[6] whereas it can be represented as an EVMDD with only $K+1$ nodes[7].

5.2 Time complexity

What makes decision diagrams a useful data structure for symbolic model-checking is not only their space efficiency but the ability to efficiently compute common operations.

Section 2 of [9] gives an algorithm to compute any binary operation on MTBDDs. The *apply* algorithm can be easily generalized to MDDs for any n-ary operator $\square_n$. It computes its result in time $\mathrm{O}\left(\prod_{i=1}^{n} |f_i|\right)$, where $|f_i|$ is the size of MTMDD representing operand i (in nodes).

Section 2.2 of [13] gives the equivalent algorithm 1 on the facing page for edge-valued decision diagrams. This is the binary version, but n-ary one is very similar for any $n \in \mathbb{N}^*$.

As stated in section 2.2.6 of [13]

Theorem 2. *The number of recursive calls of the above* apply *algorithm is the same for MTMDDs and for EVMDDs representing the same functions.*

Lemma 1. *Two paths* $(i_K, \ldots, i_{k+1})$ *and* $(j_K, \ldots, j_{k+1})$ *lead to the same node in* A

$$A.\text{node}[i_K, \ldots, i_{k+1}] = A.\text{node}[j_K, \ldots, j_{k+1}]$$

if and only if they lead to the same node in A_{k-1}

$$A_{k-1}.\text{node}[i_K, \ldots, i_{k+1}] = A_{k-1}[j_K, \ldots, j_{k+1}]$$

Proof. A and A_{k-1} are identical from level $k+1$ to level K. □

Lemma 2. *Two paths* $(i_K, \ldots, i_{k+1})$ *and* $(j_K, \ldots, j_{k+1})$ *lead to the same node in* A_K *with the same value*

$$\begin{aligned} & A_K.\text{node}[i_K, \ldots, i_{k+1}] = A_K.\text{node}[j_K, \ldots, j_{k+1}] \\ \textit{and} \quad & A_K.\text{node}(i_K, \ldots, i_{k+1}) = A_K.\text{node}(j_K, \ldots, j_{k+1}) \end{aligned}$$

if and only if they lead to the same node in A_k *with the same value*

$$\begin{aligned} & A_k.\text{node}[i_K, \ldots, i_{k+1}] = A_k.\text{node}[j_K, \ldots, j_{k+1}] \\ \textit{and} \quad & A_k.\text{node}(i_K, \ldots, i_{k+1}) = A_k.\text{node}(j_K, \ldots, j_{k+1}) \end{aligned}$$

[5] Usually 2, assuming that edge values are as big as node pointers.

[6] Since all terminal values are distinct.

[7] More generally, any linear function requires only one node per level in its EVMDD representation.

Algorithm 1 Computes any binary operation $\square_2$ on EVMDDs $\langle v, n\rangle$ and $\langle v', n'\rangle$

```
apply(□₂ : edge * edge → edge, ⟨v, n⟩ : edge, ⟨v', n'⟩ : edge) : edge
  k ← n.level  // = n'.level since EVMDDs are quasi-reduced

  // base case
  if k = 0 then
    return ⟨v □₂ v', t⟩  // t is the unique terminal node

  // lookup in cache
  if CacheFind(□₂, ⟨v, n⟩, ⟨v', n'⟩, ⟨m, r⟩) then
    return ⟨m, r⟩

  r ← NewNode(k)
  for i = 0 to n_k − 1 do
    r[i] ← apply(□₂, ⟨v + n[i].val, n[i].node⟩, ⟨v' + n'[i].val, n'[i].node⟩)
  m ← r[0].val
  for i = 0 to n_k − 1 do
    r[i].val ← r[i].val − m

  // check if a node identical to r already exists
  r ← FindOrAdd(r)

  // save result in cache
  CacheInsert(□₂, ⟨v, n⟩, ⟨v', n'⟩, ⟨m, r⟩)

  return ⟨m, r⟩
```

Proof. First, let us prove by induction on l, from k to K, that:

$$\begin{aligned} & A_k.\text{node}[i_K, \ldots, i_{k+1}] = A_l.\text{node}[i_K, \ldots, i_{k+1}] \\ \text{and} \quad & A_k.\text{val} + A_k.\text{node}(i_K, \ldots, i_{k+1}) = A_l.\text{val} + A_l.\text{node}(i_K, \ldots, i_{k+1}) \end{aligned}$$

- for $l = k$, the property trivially holds;
- if the property holds for a value of l between k and $K-1$, it still holds for $l+1$, since the computation of A_{l+1} from A_l can only merge duplicate nodes at level $l+1 \geq k+1$. □

Proof (Theorem 2). We show the proof for the unary version of the algorithm. Proofs for other versions are similar, only a bit more verbose.

If we do not take into consideration the caches, the two algorithms are obviously equivalent so what we need to prove is that a cache hit occurs in the EVMDD *apply* algorithm with diagram A_K if and only if it occurs in the MTMDD algorithm with diagram A. In other words, we have to prove for every two paths $(i_K, \ldots, i_{k+1})$ and $(j_K, \ldots, j_{k+1})$ that they lead to the same node in A

$$A.\text{node}[i_K, \ldots, i_{k+1}] = A.\text{node}[j_K, \ldots, j_{k+1}]$$

if and only if they reach the same node in A_K with the same value:

$$\begin{aligned} & A_K.\text{node}[i_K, \ldots, i_{k+1}] = A_K.\text{node}[j_K, \ldots, j_{k+1}] \\ \text{and} \quad & A_K.\text{node}(i_K, \ldots, i_{k+1}) = A_K.\text{node}(j_K, \ldots, j_{k+1}) \end{aligned}$$

Therefore, from lemmas 1 on the preceding page and 2 on the facing page it remains to prove that two paths $(i_K, \ldots, i_{k+1})$ and $(j_K, \ldots, j_{k+1})$ lead to the same node in A_{k-1}

$$A_{k-1}.\text{node}[i_K, \ldots, i_{k+1}] = A_{k-1}.\text{node}[j_K, \ldots, j_{k+1}] \tag{1}$$

if and only if they reach the same node in A_k with the same value:

$$\begin{aligned} & A_k.\text{node}[i_K, \ldots, i_{k+1}] = A_k.\text{node}[j_K, \ldots, j_{k+1}] \\ \text{and} \quad & A_k.\text{node}(i_K, \ldots, i_{k+1}) = A_k.\text{node}(j_K, \ldots, j_{k+1}) \end{aligned} \tag{2}$$

– Let us prove that (1) implies (2). The first part of (2) is obvious: if two paths lead to the same node before merging, this still holds after merging. Second part is slightly more complex. Indeed, $A_k.\text{node}(i_K, \ldots, i_{k+1})$ is just the value attached to edge i_{k+1} of node $A_k.\text{node}[i_K, \ldots, i_{k+2}]$ since all other edges hold value 0. This value is the one reported up when constructing A_k from A_{k-1} for node $A_k.\text{node}[i_K, \ldots, i_{k+1}] = A_k.\text{node}[j_K, \ldots, j_{k+1}]$ so it is the same for $A_k.\text{node}(j_K, \ldots, j_{k+1})$, which is the second part of (2).
– In the other direction: (2) implies (1).
If (2) holds then $A_{k-1}.\text{node}[i_K, \ldots, i_{k+1}]$ and $A_{k-1}.\text{node}[j_K, \ldots, j_{k+1}]$ are identical (for if they were not, then (2) does not hold on A_k computed from A_{k-1}). Hence (1), since A_{k-1} doesn't contain any duplicate at level k. □

In conclusion, EVMDD computations are at least not slower than the MTMDD equivalent. However, particular operators $\square_n$ may enable much better algorithms on EVMDDs.

Addition of Constant: EVMDD $+c$ Adding a constant c to an EVMDD $\langle v, n\rangle$ is just computing $\langle c + v, n\rangle$, which can be done in constant time.

Multiplication with Scalar: EVMDD $\times c$ As stated in section 2.2.2 of [13], computing $f \times c$ is just multiplying each edge value of EVMDD representing f by c, which can be done in time O $(|f|)$.

Addition: EVMDD + EVMDD As stated in section 2.2.2 of [13], addition satisfies the property

$$\langle v, n\rangle + \langle v', n'\rangle = (\langle 0, n\rangle + \langle 0, n'\rangle) + (v + v')$$

allowing to cache only edges with value 0, therefore leading to a slightly modified *apply*, algorithm 2 The

Algorithm 2 Computes sum of EVMDDs $\langle v, n\rangle$ and $\langle v', n'\rangle$

```
plus(⟨v, n⟩ : edge, ⟨v', n'⟩ : edge) : edge
  k ← n.level  // = n'.level since EVMDDs are quasi-reduced

  // base case
  if k = 0 then
    return ⟨v + v', t⟩  // t is the unique terminal node

  // lookup in cache
  if CacheFind(+, ⟨0, n⟩, ⟨0, n'⟩, r) then
    return ⟨v + v', r⟩

  r ← NewNode(k)
  for i = 0 to n_k − 1 do
    r[i] ← plus(n[i], n'[i])

  // check if a node identical to r already exists
  r ← FindOrAdd(r)

  // save result in cache
  CacheInsert(+, ⟨v, n⟩, ⟨v', n'⟩, ⟨m, r⟩)

  return ⟨v + v', r⟩
```

complexity of the algorithm to compute $f + g$ is O $(|f|\ |g|)$.

It's also interesting to notice that this algorithm offers a simple upper bound to the size of the result of an addition

$$|f + g| \leq |f|\ |g|$$

Remainder and Euclidean Division: EVMDD $\%c$ and EVMDD $/c$ As stated in section 2.2.4 of [13], there is no need to cache values equal modulo c, hence the complexity of these algorithms is $\mathrm{O}(|f|c)$.

Minimum and Maximum Per section 2.2.3 of [13], $\max\{f(x) \mid x \in S\}$ and $\min\{f(x) \mid x \in S\}$ can be easily computed by traversing the graph and caching the result for each node, hence the complexity is $\mathrm{O}(|f|)$.

Multiplication: EVMDD $\times$ EVMDD As stated in section 2.2.5 of [13], the result of a multiplication can have an EVMDD representation of exponential size in terms of the operands. For example, let S be $\{0,1\}^K$, $f : (x_K, \ldots, x_1) \mapsto \sum_{k=2}^{K} x_k 2^{k-2}$ and $g : (x_K, \ldots, x_1) \mapsto x_1$, f and g have both an EVMDD representation with $K+1$ nodes whereas fg needs 2^K nodes (here again, we see the importance of variable ordering, putting x_1 at top allows fg to be represented with $2K$ nodes). Therefore, we cannot expect to find an algorithm with better worst-case complexity. However, the following equation

$$\langle v, n\rangle \times \langle v', n'\rangle = vv' + v\langle 0, n'\rangle + v'\langle 0, n\rangle + \langle 0, n\rangle \times \langle 0, n'\rangle$$

suggests the alternative algorithm 3. The first product is an integer multiplication done in constant time.

Algorithm 3 Computes product of EVMDDs $\langle v, n\rangle$ and $\langle v', n'\rangle$

```
times(⟨v, n⟩ : edge, ⟨v', n'⟩ : edge) : edge
  k ← n.level  // = n'.level since EVMDDs are quasi-reduced

  // base case
  if k = 0 then
    return ⟨v × v', t⟩  // t is the unique terminal node

  // lookup in cache
  if CacheFind(×, ⟨0, n⟩, ⟨0, n'⟩, r) then
    return plusc(plus(plus(timesc(⟨0, n⟩, v'), timesc(⟨0, n'⟩, v)), r), v × v')

  r ← NewNode(k)
  for i = 0 to n_k − 1 do
    r[i] ← times(n[i], n'[i])

  // check if a node identical to r already exists
  r ← FindOrAdd(r)

  // save result in cache
  CacheInsert(×, ⟨v, n⟩, ⟨v', n'⟩, ⟨m, r⟩)

  return plusc(plus(plus(timesc(⟨0, n⟩, v'), timesc(⟨0, n'⟩, v)), r), v × v')
```

The next two are multiplications by a constant done in $\mathrm{O}(|f|)$ and $\mathrm{O}(|g|)$, respectively. The last one is done through recursive calls. The first addition takes constant time, the second one takes $\mathrm{O}(|f|\ |g|)$ and produce a result of size at most $|f|\ |g|$, hence a cost of $\mathrm{O}(|f|\ |g|\ |fg|)$ for the last addition. The function `times` is called $\mathrm{O}(|f|\ |g|)$ times, hence a final complexity of $\mathrm{O}\left(|f|^2\ |g|^2\ |fg|\right)$.

Although we were unable to theoretically compare this algorithm to the generic algorithm 1 on page 7, it seems to perform far better in practice when the size of the result is moderate.

Relational Operators: EVMDD $<$ c Relational operators ($<, >, \leq, \geq, =$ and $\neq$) are used in building transition relations. Unfortunately, the corresponding operations remain something expensive. But, as stated in section 2.2.3 of [13], $f < g$ can be computed as $f - g < 0$ and the operation "less than constant" can be greatly improved as shown in algorithm 4 through the use of min and max, which are easy to compute from section 5.2 on the previous page.

Algorithm 4 Computes $\langle v, n\rangle < c$ for EVMDD $\langle v, n\rangle$ and integer c

```
lt(⟨v, n⟩ : edge, c : int) : edge
  k ← n.level

  // base cases
  if c − v ≤ min(n) then
    return ⟨0, t⟩  // t is the unique terminal node
  if c − v > max(n) then
    return ⟨1, t⟩  // t is the unique terminal node

  // lookup in cache
  if CacheFind(<, ⟨0, n⟩, c − v, ⟨m, r⟩) then
    return ⟨m, r⟩

  r ← NewNode(k)
  for i = 0 to n_k − 1 do
    r[i] ← lt(n[i], c − v)
  m ← r[0].val
  for i = 0 to n_k − 1 do
    r[i].val ← r[i].val − m

  // check if a node identical to r already exists
  r ← FindOrAdd(r)

  // save result in cache
  CacheInsert(<, ⟨0, n⟩, c − v, ⟨m, r⟩)

  return ⟨m, r⟩
```

We also developed algorithm 5 on the facing page, which initially was deemed as "optimized". The idea behind this algorithm is that for a call $\mathrm{lt}((\_, p), \_)$ returning (lb, ub, q), the interval $[lb, ub]$ is the set $\{c \mid \forall x.\ p(x) < c \Leftrightarrow q(x) = 1\}$. In other words, $[lb, ub]$ is the exact interval for which the relation $p < .$ is given by q.

The later algorithm is theoretically better than the former in that more cache hits occur. It can even be viewed as optimal in the sense that it never creates duplicate nodes from a same node of the input EVMDD. However, in practice, this advantage appears quite small in most cases. Moreover, in order to implemented it, it requires an ordered cache, such as a search tree, instead of a simple hash table and the resulting log factor seems to overcome any advantage.

Algorithms for other relational operators analogous to $<$.

5.3 Versatility

The advantages of EVMDDs over MTMDDs come at a price of a slightly reduced versatility. Both EVMDDs and MTMDDs can be used to represent functions more general than arithmetic functions over integers. For MTMDDs, the only restriction is that the domain and co-domain of the encoded function have to be finite. However EVMDDs, despite the extensions of section 4.2 on page 4, also require some algebraic structure. For example, they cannot be directly used with floats since the sum of two floating point values has to be rounded to be represented as a floating point value. In computer arithmetic, [15] encodes floating point functions through integer EVMDDs of their binary (bit-level) representation. Even though space efficient, floating point computations on this representation remain expensive.

Another problem affecting EVMDDs but not MTMDDs is *false overflow*. Indeed, computers do not operate over $\mathbb{Z}$ proper, but on $\mathbb{Z}/2^{32}\mathbb{Z}$ (for 32bit platforms, or the equivalent structure for the appropriate size of integers – 16, 32, 64 bits). Integer overflow is possible when adding edge values, even when the represented function may fit within 32 bits. An example of such an internal overflow can be seen in figure 4 on the facing page. Since $\mathbb{Z}/2^{32}\mathbb{Z}$ is still an additive group, these overflows are harmless for addition and multiplication by a constant. The general algorithm from section 5.2 on page 6 is not affected either, as it applies operations only on leaf nodes, but this observation no longer holds on the other algorithms of

Algorithm 5 Computes $\langle v, n\rangle < c$ for EVMDD $\langle v, n\rangle$ and integer c

lt($\langle v, n\rangle$: edge, c : int) : int $*$ int $*$ edge
  $k \leftarrow n$.level

  // base cases
  if $c - v \leq \min(n)$ **then**
    return $(-\infty, \min(n), \langle 0, t\rangle)$ *// t is the unique terminal node*
  if $c - v > \max(n)$ **then**
    return $(\max(n) + 1, +\infty, \langle 1, t\rangle)$ *// t is the unique terminal node*

  // lookup in cache for an interval $[lb, ub]$ containing $c - v$
  if CacheFind($<$, $\langle 0, n\rangle$, $c - v$, $(lb, ub, \langle m, r\rangle)$) **then**
    return $(lb, ub, \langle m, r\rangle)$

  $r \leftarrow$ NewNode(k)
  $lb \leftarrow -\infty$
  $ub \leftarrow +\infty$
  for $i = 0$ **to** $n_k - 1$ **do**
    $lb', ub', r[i] \leftarrow$ lt($n[i]$, $c - v$)
    $lb \leftarrow \max(lb, lb' + n[i].\text{val})$
    $ub \leftarrow \min(ub, ub' + n[i].\text{val})$
  $m \leftarrow r[0].\text{val}$
  for $i = 0$ **to** $n_k - 1$ **do**
    $r[i].\text{val} \leftarrow r[i].\text{val} - m$

  // check if a node identical to r already exists
  $r \leftarrow$ FindOrAdd(r)

  // save result in cache
  CacheInsert($<$, $\langle 0, n\rangle$, $[lb, ub]$, $(lb, ub, \langle m, r\rangle)$)

  return $(lb, ub, \langle m, r\rangle)$

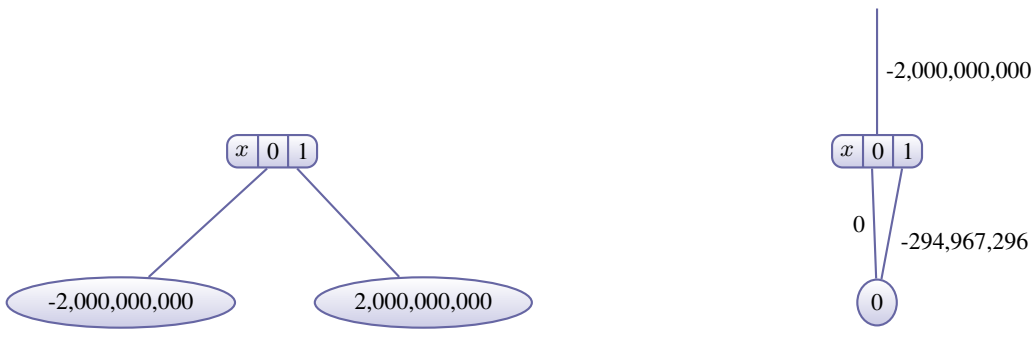

Fig. 4. MTMDD (left) and EVMDD (right) encoding the same function over 32-bit integers, $f : x \in \{0, 1\} \mapsto 2,000,000,000 \times (2x - 1)$.

this section. A necessary and sufficient condition on a function f for not having false overflows in the corresponding EVMDD is

$$\forall k \in \{1, \ldots, K\} \,.\, \forall (i_K, \ldots, i_{k+1}) \in S_K \times \ldots \times S_{k+1} \,.\, \forall i_k \in S_k \,.$$
$$f(i_K, \ldots, i_{k+1}, i_k, 0, \ldots, 0) - f(i_K, \ldots, i_{k+1}, 0, 0, \ldots, 0) \text{ does not overflow}$$

Despite these drawbacks, EVMDDs are very adequate for encoding discrete-state transition relations.

6 Implementation

Symbolic model checkers such as (Nu)SMV[1] or SAL[2] are based on the library CUDD[10] which offers an efficient implementation of BDDs and MTBDDs. For state space generation, they use a plain breadth first search (BFS) algorithm.

Our goal was to implement a new symbolic model checking library featuring EVMDDs for transition relation construction and saturation[6] for state space generation. We also developed a basic model checking front-end to test the library and compare it to CUDD. Binaries for both and model checker sources are available at `http://research.nianet.org/~radu/evmdd/`.

In this section, we discuss some implementation details before presenting experimental results.

6.1 Memory Management

It is well known that model checking is highly memory intensive, making memory management a critical issue when implementing a decision diagrams library.

The usual addressing scheme of BDD nodes is through pointers. Here, with MDD, nodes at different levels with different variable ranges are of different sizes. For that reason, we chose an index-based addressing scheme $(level, index)$, allowing to allocate nodes independently at each level.

Decision diagrams algorithms rely heavily on several cache structures. They are usually implemented as lossy hash tables: upon collision, older entries are simply discarded. We avoided the loss of information by choosing dynamically resizing hash tables with chaining. The non-lossy caches give slightly faster algorithms at the price of some additional memory.

Different diagrams often share subgraphs, making explicit freeing of disconnected sub-diagrams very difficult. Moreover, symbolic model checking algorithms often produce a large number of temporary nodes which rapidly become disconnected (also called garbage). The easiest way to reclaim memory is to use some garbage collection procedure. The most commonly used technique is via reference counting. This is perfectly suited since diagrams are DAGs, hence avoiding circular reference, the main drawback of reference counting. Instead, we chose to use a simple *mark & sweep* algorithm that proved to be both simple and efficient, because it requires minimal book-keeping. We keep reference counting only as an interface with library users.

6.2 Encoding the Transition Relation

We represent the transition relation T as a disjunction $\bigcup_{e \in E} T_e$ of *events* e. Each event T_e is represented as an MDD of high $2K$, with level $2k$ encoding variable x_k before the transition and level $2k-1$ same variable x'_k but after the transition[8].

This disjunction representation is well suited for globally–asynchronous locally–synchronous systems, where each event encodes some local transition. However, we could end up with many events that are just the identity relation for most variables, The numerous "identity patterns" in the MDD are very expensive to deal with, both in terms of memory usage and computation time. To avoid this problem, we chose yet

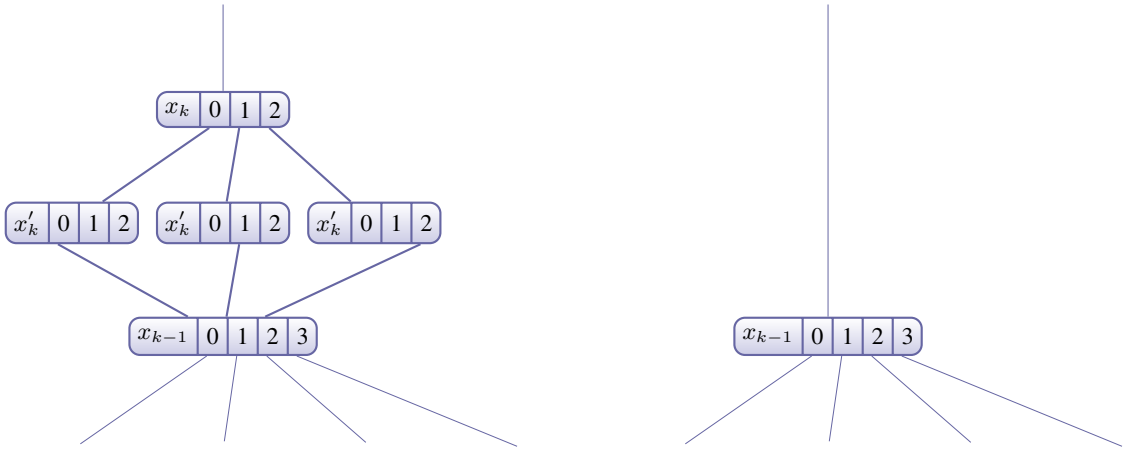

Fig. 5. An identity pattern in a reduced MDD (left) and its fully-identity reduced equivalent (right).

another reduction rule of nodes, different than the two already presented in definitions 4 on page 4 and 5 on page 4: the *full-identity reduction* from [8].

An example of such an identity pattern and its full-identity reduction is shown in Figure 5. In this figure, edges leading to terminal 0 are omitted for clarity.

Table 1 shows the execution time with standard reduction and full-identity reduction rules for the classical model of dining philosophers.

Model size	standard (sec)	full-identity (sec)
10	0.02	0.00
50	0.36	0.02
100	1.68	0.04
200	7.85	0.07

Model size	standard (sec)	full-identity (sec)
400	36.72	0.20
500	—	0.27
1000	—	0.91
7000	—	37.47

Table 1. Execution time for building the transition relation using standard and fully-identity reduced MDDs ("—" means "out of memory").

6.3 State Space Construction

For state space construction, we use the Saturation algorithm [6] instead of the classical breadth first search exploration. This heuristic often gives spectacular improvements when building the state spaces of globally–asynchronous locally–synchronous systems. This is certainly the major source of improvement of our implementation over existing BDD libraries.

As advised in [8], we chose to merge all events e with same topmost affected level[9] in the same MDD. The cost of the unions slows down the generation of the transition relation but generally makes the state space construction up to several times faster.

[8] That is, we consider T_e as a part of $S \times S = S_K \times S_K \times \ldots \times S_1 \times S_1$. It is interesting to notice, that T_e does not need to be a function $S \to S$, allowing to model non deterministic behaviors.

[9] i.e. same value of $\max\{k \mid \exists i, j \in S_k \,.\, i \neq j \wedge (i,j) \in T_e\}$

6.4 Experimental Results

Our new model checker comprises 7K lines of ANSI-C code for the library and 4K lines for the simple model checker that provides a common interface to our library and CUDD. Table 2 shows execution times for finding deadlocks on a suite of classical models. The search for deadlocks in a deadlock-free system equates to building state space. Programs to generate all models can be found in the `examples` directory of our simple model checker source distribution, available at `http://research.nianet.org/~radu/evmdd/`.

We collected the results on a Linux machine with Intel Core 2 processor, 1.2GHz, 1.5GB of memory.

Model size	Reachable states	Deadlock states	CUDD (in s)	EVMDD (in s)
Dining philosophers				
100	4×10^{62}	2	5.15	0.04
200	2×10^{125}	2	1493.36	0.10
1000	9×10^{626}	2	—	1.10
5000	6×10^{3134}	2	—	20.48
10000	4×10^{6269}	2	—	77.77
16000	2×10^{10031}	2	—	196.84
Round robin mutual exclusion protocol				
20	4×10^{7}	0	0.47	0.04
40	9×10^{13}	0	4.04	0.46
60	1×10^{20}	0	15.59	1.54
100	2×10^{32}	0	599.16	7.42
200	7×10^{62}	0	—	75.94
Slotted ring protocol				
10	8×10^{9}	0	1.07	0.01
15	1×10^{15}	0	13.00	0.03
20	2×10^{20}	0	1804.48	0.04
100	2×10^{105}	0	—	3.74
200	8×10^{211}	0	—	29.73
300	3×10^{318}	0	—	111.29
500	5×10^{531}	0	—	505.04

Model size	Reachable states	Deadlock states	CUDD (in s)	EVMDD (in s)
Kanban assembly line				
10	1×10^{9}	0	0.84	0.01
15	4×10^{10}	0	86.58	0.03
20	8×10^{11}	0	811.89	0.04
100	1×10^{19}	0	—	3.25
200	3×10^{22}	0	—	32.31
400	6×10^{25}	0	—	697.90
Knights problem				
5	6×10^{7}	0	921.95	0.30
7	1×10^{15}	0	—	3.73
9	8×10^{24}	0	—	47.80
Randomized leader election protocol				
5	1×10^{5}	5×10^{3}	0.90	0.32
6	2×10^{6}	3×10^{4}	4.62	1.20
7	2×10^{7}	2×10^{5}	23.87	3.69
8	3×10^{8}	2×10^{6}	176.16	10.04
9	5×10^{9}	1×10^{7}	810.36	24.86
10	6×10^{10}	1×10^{8}	—	85.54
11	9×10^{11}	8×10^{8}	—	423.41

Table 2. Execution times for finding deadlocks using our library or CUDD ("—" means "more than an hour").

Note that using other existing tools, such as NuSMV or SAL on these models, we get execution times of the same order of magnitude as with our model checker using CUDD.

Compared to the first implementation of saturation algorithm [6] in the tool SMART, our new implementation is always several (up to a few dozens) times faster. This is due to both the encoding of the transition relation and our simple C implementation in comparison to the object-oriented C++ version.

These models feature a transition relation without much arithmetic. Actual interest of using EVMDDs for building transition relation still needs to be seriously investigated.

7 Conclusions and Future Work

We studied the advantages of the EVMDD data structure over the widely used MTBDDs for the construction of transition relation of finite state systems and implemented them in a library, along with state-of-the-art algorithms for state space generation. We obtained execution times several orders of magnitude faster than the CUDD library and classical algorithms, with a reduced memory usage enabling to handle extremely large systems. Future work should focus primarily on integrating our library into the SAL model checker.

Our results show that, as old fashioned as it may seem, symbolic model checking remains an efficient technique for analyzing globally–asynchronous locally–synchronous systems and significant improvements are still possible.

8 About the Internship

This internship was an excellent opportunity to discover symbolic model checking, since I used to know almost nothing about model checking before it.

I would like to thank Radu Siminiceanu a lot for being more than an attentive internship director. Thanks also to Gilles Dowek for his appreciated welcoming at the beginning of the internship, when I was still suffering the effects of jet lag. I won't forget some of the people doing research in room 105 at NIA: Alwyn Goodloe, Paolo Gradassi, Heber Herencia Zapana, Mark van der Steen, Taylor Spalt and David Billingsley who make me discovering the environment of Hampton, Washington and both sites of the unique National Air and Space Museum. Thanks again to César Muñoz and the Langley Formal Methods team from NASA Langley Research Center for sharing many lunches with us, to Daniel Hirschkoff and Jean-Christophe Filliâtre for sending me there and finally to NIA administrative staff for his happy efficiency.

References

1. NuSMV model checker. http://nusmv.irst.itc.it/.
2. SAL model checker. http://sal.csl.sri.com/.
3. R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *ICCAD*, pp. 188–191, 1993.
4. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
5. Randal E. Bryant and Yirng-An Chen. Verification of arithmetic functions with binary moment diagrams. Techn. Rep. CS-94-160, CMU, 1994.
6. Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In Tiziana Margaria and Wang Yi, eds., *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2031, pp. 328–342, Genova, Italy, April 2001. Springer.
7. Gianfranco Ciardo and Radu Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In Mark D. Aagaard and John W. O'Leary, eds., *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 2517, pp. 256–273, Portland, OR, USA, November 2002. Springer.
8. Ciardo, G. and Yu, J. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In *Proc. CHARME*, October 2005.
9. Edmund Clarke, M. Fujita, and X. Zhao. Application of multi-terminal binary decision diagrams. Techn. rep. IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design, 1995.
10. Fabio Somenzi. CUDD library, Colorado University Decision Diagram. http://vlsi.colorado.edu/~fabio/CUDD/.
11. K. S. Brace and R. L. Rudell and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th Design Automation Conference*, pp. 40–45, June 1990.
12. T. Kam, T. Villa, R.K. Brayton, and Alberto Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
13. Y.-T. Lai, M. Pedram, and B. K. Vrudhula. Formal verification using edge-valued binary decision diagrams. *IEEE Transactions on Computers*, 45:247–255, 1996.
14. Yung-Te Lai and Sarma Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *DAC*, pp. 608–613, 1992.
15. S. Nagayama, and T. Sasao, and J. T. Butler. Floating-point numerical function generators using EVMDDs for monotone elementary functions. In *39th International Symposium on Multiple-Valued Logic (ISMVL 2009)*, pp. 349–355, 2009.