Factored Edge-Valued Binary Decision Diagrams [1]

Paul Tafertshofer and Massoud Pedram
University of Southern California
Department of Electrical Engineering- Systems
Los Angeles, CA 90089

[1]This research was supported in part by NSF under contract No. MIP-9211668. The first author is presently with the Institute of Electronic Design Automation, Technical University of Munich, Germany.

List of Figures

List of Tables

Abstract

Factored Edge-Valued Binary Decision Diagrams form an extension to Edge-Valued Binary Decision Diagrams. By associating both an additive and a multiplicative weight with the edges, FEVBDDs can be used to represent a wider range of functions concisely. As a result, the computational complexity for certain operations can be significantly reduced compared to EVBDDs. Additionally, the introduction of multiplicative edge weights allows us to directly represent the so-called complement edges which are used in OBDDs, thus providing a one to one mapping of all OBDDs to FEVBDDs. Applications such as integer linear programming and logic verification that have been proposed for EVBDDs also benefit from the extension. We also present a complete matrix package based on FEVBDDs and apply the package to the problem of solving the Chapman-Kolmogorov equations.

Keywords: Ordered Binary Decision Diagrams, Pseudo-Boolean Functions, Affine Property, Logic Verification, Integer Linear Programming, Matrix Operations.

1 Introduction

Over the past decade a drastic increase in the integration of VLSI chips has taken place. Consequently, the complexity of the circuit designs has risen dramatically so that today's circuit designers rely more and more on sophisticated computer-aided design (CAD) tools. The goal of CAD tools is to automatically transform a description in the algorithmic or behavioral domains to one in the physical domain, i.e. down to a layout mask for chip production. We divide this process into four different levels: *system, behavioral, logic* and *layout*.

At the logic level, the behavior of the circuit is described by boolean functions. The efficiency of the algorithms applied in this level depends largely on the chosen data structure. Originally, representations such as the sum of products form or factored form representations were predominant. Today, the most popular data structure for boolean functions is the Ordered Binary Decision Diagram (OBDD) which provides a compact and canonical representation. In the wake of the successful introduction of the concept of function graphs by OBDDs, various other function graphs have been proposed which are not constrained to boolean functions but can be used to denote arithmetic functions. These function graphs have been used for state reduction in finite state machines and logic verification of higher-level specifications. Additionally, they have been applied to problems outside CAD, such as integer linear programming and matrix representation.

Since the introduction of OBDDs by R. E. Bryant [5], several different forms of function graphs have been proposed. Functional Decision Diagrams (FDD) have been presented as an alternative to OBDDs for representing boolean functions [3]. Ordered Kronecker Functional Decision Diagrams (OKFDD) have been introduced in [10] as a generalization of OBDDs and FDDs. Multi-Terminal Binary Decision Diagrams (MTBDD) [9] have been proposed to represent integer valued functions and extended to functions on finite sets [2]. Edge-Valued Binary Decision Diagrams (EVBDD) [12][13][14] provide a more compact means of representing such functions. Recently Binary Moment Diagrams (BMD and *BMD) [7] were introduced which permit efficient word-level verification of arithmetic functions (including multipliers of up to 62-bit word size).

This paper presents Factored Edge-Valued Binary Decision Diagrams (FEVBDD) as an extension to EVBDDs. By associating both an additive and a multiplicative weight with the edges, FEVBDDs can be used to represent a wider range of functions concisely. As a result, the computational complexity for certain operations can be significantly reduced compared to EVBDDs. Additionally, the introduction of multiplicative edge weights allows us to directly represent the complement edges which are used in OBDDs. This paper also describes uses of FEVBDDs in applications such as integer linear programming, logic verification and matrix representation and manipulation.

2 Review of Edge-Valued Binary Decision Diagrams

Edge-Valued Binary Decision Diagrams, which were proposed by Lai, et al. [12][13][14] offer a direct extension to the concept of OBDDs. By associating a so-called edge value ev to every then-edge of the OBDD they are capable of representing pseudo-boolean functions such as integer valued functions. Their application has proven successful in such areas as formal verification and integer linear programming, spectral transformation, and function decomposition.

Definition 2.1 *An EVBDD is a tuple $\langle c, \mathbf{f} \rangle$ where c is a constant value and* **f** *is a rooted, directed acyclic graph $(V \cup T, E)$ consisting of two types of vertices.*

- *A nonterminal vertex* $\mathbf{f} \in V$ *is represented by a quadruple $\langle variable(\mathbf{f}), child_t(\mathbf{f}), child_e(\mathbf{f}), ev \rangle$, where $variable(\mathbf{f}) \in \{x_0, \ldots, x_{n-1}\}$ is a binary variable.*

- *The single terminal vertex $\mathbf{f} \in T$ with value 0 is denoted by* **0**.

There is no nonterminal vertex **f** *such that $child_t(\mathbf{f}) = child_e(\mathbf{f})$ and $ev = 0$, and there are no two nonterminal vertices* **f** *and* **g** *such that $\mathbf{f} = \mathbf{g}$. Furthermore, there exists an index function $index(x) \in \{0, \ldots, n-1\}$ such that the following holds for every nonterminal vertex. If $child_t(\mathbf{f})$ is also nonterminal, then we must have $index(variable(\mathbf{f})) < index(variable(child_t(\mathbf{f})))$. If $child_e(\mathbf{f})$ is nonterminal, then we must have $index(variable(\mathbf{f})) < index(variable(child_e(\mathbf{f})))$.*

Definition 2.2 *An EVBDD $\langle c, \mathbf{f} \rangle$ denotes the arithmetic function $c + f : \{0,1\}^n \rightarrow integer$ where f is the function f denoted by $\mathbf{f} = \langle x, \mathbf{f}_t, \mathbf{f}_e, ev \rangle$. The terminal node* **0** *represents the constant function $f = 0$, and $\langle x, \mathbf{f}_t, \mathbf{f}_e, ev \rangle$ denotes the arithmetic function $f = x \cdot (ev + f_t) + (1 - x) \cdot f_e$.*

Definitions (2.1), (2.2) provide a graphical representation of pseudo-boolean functions. As a consequence integer variables have to be encoded in binary as in $X = \sum_{i=0}^{n-1} x_i \cdot 2^i$ where X is a n-bit integer variable. It has been shown that EVBDDs form a canonical representation of arithmetic functions.

Definition 2.3 *Given an EVBDD $\langle c, \mathbf{f} \rangle$ representing $f(x_0, \ldots, x_{n-1})$ and a function Φ that for each variable x assigns a value $\Phi(x)$ equal to either 0 or 1, the function EVBDDeval is defined as*

$$\mathit{EVBDDeval}(\langle c, \mathbf{f} \rangle, \Phi) = \begin{cases} c & \mathbf{f} \textit{ is the terminal node } \mathbf{0} \\ \mathit{EVBDDeval}(\langle c + ev, child_t(\mathbf{f}) \rangle, \Phi) & \Phi(variable(\mathbf{f})) = 1 \\ \mathit{EVBDDeval}(\langle c, child_e(\mathbf{f}) \rangle, \Phi) & \Phi(variable(\mathbf{f})) = 0 \end{cases}$$

boolean	arithmetic
$\neg x$	$1 - x$
$x \vee y$	$x + y - xy$
$x \wedge y$	xy
$x \oplus y$	$x + y - 2xy$

Table 1: Arithmetic equivalents of boolean functions

Boolean functions can be represented in EVBDDs by using the integers 0 and 1 to denote the boolean values $true$ and $false$. Boolean operations are implemented through arithmetic operations as shown in Table 1. A method has been described by Lai, et al. that converts any OBDD representation of a boolean function to its corresponding EVBDD representation. It can be proven that both function graphs OBDD **v** and EVBDD $\langle c, \mathbf{v}' \rangle$ denoting the same function f share the same topology except that the terminal node **1** is absent from the EVBDD and the edges connected to it are redirected to the single terminal node **0**. Additionally, it was shown that boolean operations executed on EVBDDs have the same time complexity $O(|f| \cdot |g|)$ as boolean operations on OBDDs. The concept of complement edges can not be realized in EVBDDs.

As has been done for OBDDs, a generic operation *apply* can be defined that implements arbitrary arithmetic operations on the EVBDD representations $\langle c_f, \mathbf{f} \rangle$, and $\langle c_g, \mathbf{g} \rangle$ of two arithmetic functions f and g. In general, the time complexity of such an operation on two EVBDDs $\langle c_f, \mathbf{f} \rangle$, and $\langle c_g, \mathbf{g} \rangle$ is $O(\|\langle c_f, \mathbf{f} \rangle\| \cdot \|\langle c_g, \mathbf{g} \rangle\|) = O(|\langle c_f, \mathbf{f}' \rangle| \cdot |\langle c_g, \mathbf{g}' \rangle|$ where $\langle c_f, \mathbf{f}' \rangle$, and $\langle c_g, \mathbf{g}' \rangle$ denote the flattened EVBDDs of $\langle c_f, \mathbf{f} \rangle$ $\langle c_g, \mathbf{g} \rangle$, respectively. A flattened EVBDD is defined in exactly the same manner as an MTBDD. For operations such as addition, subtraction, scalar-multiplication, etc. the time complexity of *apply* can be drastically reduced by exploiting certain properties. A scalar multiplication $c \cdot (c_f + f)$ can be done with time complexity $O(|\langle c_f, \mathbf{f} \rangle|)$ by simply multiplying all edge values by c. All operations op, such as addition, that fulfill the *additive property*

$$\langle c_f, \mathbf{f} \rangle op \langle c_g, \mathbf{g} \rangle = \langle c_f op c_g, \mathbf{f}\, op \mathbf{g} \rangle \qquad (1)$$

have the reduced time complexity $O(|\langle c_f, \mathbf{f} \rangle| \cdot |\langle c_g, \mathbf{g} \rangle|)$.

Based on EVBDDs, the concept of structured EVBDDs (SEVBDDs) has been developed in [14]. SEVBDDs allow the modeling of conditional expressions and vectors. Their main use lies in the field of formal verification.

3 Factored Edge-Valued Binary Decision Diagrams

Factored Edge-Valued Binary Decision Diagrams (FEVBDD) are an extension to EVBDDs. By associating both an additive and a multiplicative weight with the true-edges[1] FEVBDDs offer a more compact representation of linear functions, since common subfunctions differing only by an affine transformation can now be expressed by a single subgraph. Additionally, they allow the notion of complement edges to be transferred from OBDDs to FEVBDDs.

Definition 3.1 *An FEVBDD is a tuple $\langle c, w, \mathbf{f}, rule\rangle$ where c and w are constant values, **f** is a rooted, directed acyclic graph $(V \cup T, E)$ consisting of two types of vertices, and rule is the set of weight normalizing rules applied to the graph.*

- *A nonterminal vertex $\mathbf{f} \in V$ is represented by a 6-tuple[2] $\langle$variable$(\mathbf{f}), child_t(\mathbf{f}), child_e(\mathbf{f}), ev, w_t, w_e\rangle$, where variable$(\mathbf{f}) \in \{x_0, \ldots, x_{n-1}\}$ is a binary variable.*
- *The single terminal vertex $\mathbf{f} \in T$ with value 0 is denoted by **0**. By definition all branches leading to **0** have an associated weight $w = 0$.*

*There is no nonterminal vertex **f** such that $child_t(\mathbf{f}) = child_e(\mathbf{f})$, $ev = 0$, and $w_e = w_t = 1$, and there are no two nonterminal vertices **f** and **g** such that $\mathbf{f} = \mathbf{g}$. Furthermore, there exists an index function index$(x) \in \{0, \ldots, n-1\}$ such that the following holds for every nonterminal vertex. If $child_t(\mathbf{f})$ is also nonterminal, then we must have index(variable$(\mathbf{f})) <$ index(variable$(child_t(\mathbf{f})))$. If $child_e(\mathbf{f})$ is nonterminal, then we must have index(variable$(\mathbf{f})) <$ index(variable$(child_e(\mathbf{f})))$.*

Definition 3.2 *A FEVBDD $\langle c_f, w_f, \mathbf{f}, rule_f\rangle$ denotes the arithmetic function $c_f + w_f \cdot f$ where f is the function f denoted by $\mathbf{f} = \langle x, \mathbf{f}_t, \mathbf{f}_e, ev, w_t, w_e\rangle$. The terminal node **0** represents the constant function $f = 0$, and $\langle x, \mathbf{f}_t, \mathbf{f}_e, ev, w_t, w_e\rangle$ denotes the arithmetic function $f = x \cdot (ev + w_t \cdot f_t) + (1 - x) \cdot w_e \cdot f_e$.*

Definition 3.3 *Given a FEVBDD $\langle c_f, w_f, \mathbf{f}, rule_f\rangle$ representing $f(x_0, \ldots, x_{n-1})$ and a function $\mathbf{\Phi}$ that for each variable x assigns a value $\mathbf{\Phi}(x)$ equal to either 0 or 1, the function FEVBDDeval is defined as:*

FEVBDDeval$(\langle c_f, w_f, \mathbf{f}, rule_f\rangle, \mathbf{\Phi}) =$

[1]The GCD rule requires also a multiplicative weight to be associated with the else-edges.

[2]If we use the rational rule it holds that $w_e = 1$ for all nodes. Thus we can represent a nonterminal vertex by a 5-tuple $\langle$*variable*$(\mathbf{f}), child_t(\mathbf{f}), child_e(\mathbf{f}), ev, w_t\rangle$.

$$= \begin{cases} c_f & \textbf{f} \textit{ is the terminal node } \mathbf{0} \\ c_f + w_f \cdot \textit{FEVBDDeval}(\langle ev, w_t, child_t(\mathbf{f}), rule\rangle, \Phi) & \Phi(variable(\mathbf{f})) = 1 \\ c_f + w_f \cdot \textit{FEVBDDeval}(\langle 0, w_e, child_e(\mathbf{f}), rule\rangle, \Phi) & \Phi(variable(\mathbf{f})) = 0 \end{cases}$$

Figure 1 goes here.

As an example, we construct the various function graphs based on the different decompositions of function f given in its tabular form in Figure 1.

$$\begin{aligned} f(x,y,z) &= 15\ (1-x)\ (1-y)\ (1-z) + 6\ (1-x)\ (1-y)\ z + \\ &\quad 5\ (1-x)\ y\ (1-z) + 2\ (1-x)\ y\ z + \\ &\quad 13\ x\ (1-y)\ (1-z) + 7\ x\ (1-y)\ z + \\ &\quad 5\ x\ y\ (1-z) + 2\ x\ y\ z \qquad (2) \\ &= 15 + x(-2 + y(-8 + z(-3)) + (1-y)(z(-6))) + \qquad (3) \\ &\quad (1-x)(y(-10 + z(-3)) + (1-y)(z(-9))) \\ &= 15 - 9(x(\frac{2}{9} + \frac{2}{3}(y(\frac{4}{3} + \frac{1}{2}z) + (1-y)z)) + \qquad (4) \\ &\quad (1-x)(y(\frac{10}{9} + \frac{1}{3}z) + (1-y)z)) \end{aligned}$$

Equation (2) is in a form that directly corresponds to the function decomposition for MTBDDs or ADDs and the tabular form. Equations (3) and (4) reflect the structure of the decomposition rules for EVBDDs and FEVBDDs, respectively. The different function graphs are shown in Figure 1.

Figure 2 goes here.

Figure 3 goes here.

Figure 4 goes here.

Representations of signed integers based on FEVBDDs are presented in Figure 2 and representations of word-level sum and product are given in Figures 3 and 4.

Lemma 3.1 *Given two FEVBDDs $\langle c_f, w_f, \mathbf{f}, rule_f \rangle$ and $\langle c_g, w_g, \mathbf{g}, rule_g \rangle$, which have been generated using the same weight normalizing rule and with f and g being non-isomorphic, it holds that there exists an assignment $\mathbf{\Phi} \in \{0,1\}^n$ such that $c_f + w_f \cdot f \neq c_g + w_g \cdot g$ for this assignment.*

Proof:

Case 1: if $c_f \neq c_g$ then let $\mathbf{\Phi} = \mathbf{0}$; it follows that *FEVBDDeval*$(\langle c_f, w_f, \mathbf{f}, rule_f \rangle, \mathbf{\Phi}) = c_f \neq$ *FEVBDDeval*$(\langle c_g, w_g, \mathbf{g}, rule_g \rangle, \mathbf{\Phi}) = c_g$.

Case 2: $c_f = c_g$ and $w_f = w_g$; by the definition of non-isomorphism it holds that $\exists \mathbf{\Phi}$ such that *FEVBDDeval*$(\langle 0, 1, \mathbf{f}, rule \rangle, \mathbf{\Phi}) \neq$ *FEVBDDeval*$(\langle 0, 1, \mathbf{g}, rule \rangle, \mathbf{\Phi})$. Consequently, we have that $c_f + w_f \cdot f \neq c_g + w_g \cdot g$ for this assignment $\mathbf{\Phi}$.

Case 3: $c_f = c_g$ and $w_f \neq w_g$; we assume that it holds that f and g are non-isomorphic and that $c_f + w_f \cdot f = c_g + w_g \cdot g$ for all assignments $\mathbf{\Phi}$. This implies that $w_f \cdot f = w_g \cdot g$ or $f = \frac{w_g}{w_f} \cdot g$. Consequently, f and g are isomorphic which contradicts the original assumption. Thus, it holds that $\exists \mathbf{\Phi}$ such that $c_f + w_f \cdot f \neq c_g + w_g \cdot g$. □

Theorem 3.1 *Two FEVBDDs $\langle c_f, w_f, \mathbf{f}, rule_f \rangle$ and $\langle c_g, w_g, \mathbf{g}, rule_g \rangle$ that have been generated using the same weight normalizing rule, i.e. $rule_f = rule_g$, denote the same function, i.e. $\forall \mathbf{\Phi} \in \{0,1\}^n$, FEVBDDeval$(\langle c_f, w_f, \mathbf{f}, rule_f \rangle, \mathbf{\Phi}) =$ FEVBDDeval$(\langle c_g, w_g, \mathbf{g}, rule_g \rangle, \mathbf{\Phi})$, if and only if $c_f = c_g$, $w_f = w_g$, and* **f** *and* **g** *are isomorphic.*

Proof:
Sufficiency: If $c_f = c_g$ and $w_f = w_g$ and **f** and **g** are isomorphic, then $\forall \mathbf{\Phi}$, *FEVBDDeval*$(\langle c_f, w_f, \mathbf{f}, rule_f \rangle, \mathbf{\Phi}) =$ *FEVBDDeval*$(\langle c_g, w_g, \mathbf{g}, rule_g \rangle, \mathbf{\Phi})$ follows directly from the definitions of isomorphism and *FEVBDDeval*.
Necessity: If $c_f \neq c_g$ then let $\mathbf{\Phi} = \mathbf{0}$; it holds that *FEVBDDeval*$(\langle c_f, w_f, \mathbf{f}, rule_f \rangle, \mathbf{\Phi}) = c_f \neq c_g =$ *FEVBDDeval*$(\langle c_g, w_g, \mathbf{g}, rule_g \rangle, \mathbf{\Phi})$. If $c_f = c_g$ and $w_f \neq w_g$ then let $\mathbf{\Phi}$ be an arbitrary assignment such that *FEVBDDeval*$(\langle 0, 1, \mathbf{f}, rule \rangle, \mathbf{\Phi}) \neq 0$ and *FEVBDDeval*$(\langle 0, 1, \mathbf{g}, rule \rangle, \mathbf{\Phi}) \neq 0$; it holds that *FEVBDDeval*$(\langle c_f, w_f, \mathbf{f}, rule_f \rangle, \mathbf{\Phi}) = c_f + w_f \cdot$ *FEVBDDeval*$(\langle 0, 1, \mathbf{f}, rule \rangle, \mathbf{\Phi})$ and

FEVBDDeval$(\langle c_g, w_g, \mathrm{g}, rule_g\rangle, \Phi) = c_g + w_g \cdot$ *FEVBDDeval*$(\langle 0, 1, \mathrm{g}, rule\rangle, \Phi)$. If f and g are isomorphic then it holds by the definition of isomorphism and *FEVBDDeval* that *FEVBDDeval*$(\langle 0, 1, \mathrm{f}, rule\rangle, \Phi) =$ *FEVBDDeval*$(\langle 0, 1, \mathrm{g}, rule\rangle, \Phi) = val$. It follows that $c_f + w_f \cdot val \neq c_g + w_g \cdot val$. If f and g are non-isomorphic lemma 3.1 holds. Now we have to prove the lemma for the last condition **f** being isomorphic to **g**. We need to show that if **f** and **g** are not isomorphic, then $\exists \Phi \in \{0,1\}^n$ such that *FEVBDDeval*$(\langle 0, 1, \mathrm{f}, rule\rangle, \Phi) \neq$ *FEVBDDeval*$(\langle 0, 1, \mathrm{g}, rule\rangle, \Phi)$. Without loss of generality, we assume *index*(*variable*$(\mathrm{f})) \leq$ *index*(*variable*$(\mathrm{g}))$. Let $k = n -$ *index*(*variable*$(\mathrm{f}))$, we will prove the lemma by induction on k.

Base: If $k = 0$, both **f** and **g** are terminal nodes. Furthermore, $\mathrm{f} = \mathrm{g} = \mathbf{0}$. Thus, f and g are isomorphic.

Induction hypothesis: Assume the above holds for $n -$ *index*(*variable*$(\mathrm{f})) < k$.

Induction: We show that the hypothesis holds for $n -$ *index*(*variable*$(\mathrm{f})) = k$. Let $\mathrm{f} = \langle x_{n-k}, \mathrm{f_t}, \mathrm{f_e}, ev_f, w_{t_f}, w_{e_f}\rangle$.

Case 1: $n -$ *index*(*variable*$(\mathrm{g})) = k$, i.e. $\mathrm{g} = \langle x_{n-k}, \mathrm{g_t}, \mathrm{g_e}, ev_g, w_{t_g}, w_{e_g}\rangle$.

If $ev_f \neq ev_g$ then let $\Phi(x_{n-k}) = 1$ and $\Phi(x_i) = 0, \forall i \neq n-k$. Then it holds that *FEVBDDeval*$(\langle 0, 1, \mathrm{f}, rule\rangle, \Phi) = ev_f \neq ev_g =$ *FEVBDDeval*$(\langle 0, 1, \mathrm{g}, rule\rangle, \Phi)$. If $ev_f = ev_g$ and $w_{t_f} \neq w_{t_g}$ then let Φ be an arbitrary assignment such that $\Phi(x_{n-k}) = 1$ and *FEVBDDeval*$(\langle 0, 1, \mathrm{f}, rule\rangle, \Phi) \neq 0$ and *FEVBDDeval*$(\langle 0, 1, \mathrm{f}, rule\rangle, \Phi) \neq 0$. Then it holds that *FEVBDDeval*$(\langle 0, 1, \mathrm{f}, rule\rangle, \Phi) = ev_f + w_{t_f} \cdot$ *FEVBDDeval*$(\langle 0, 1, \mathrm{f_t}, rule\rangle, \Phi)$ and *FEVBDDeval*$(\langle 0, 1, \mathrm{g}, rule\rangle, \Phi) = ev_g + w_{t_g} \cdot$ *FEVBDDeval*$(\langle 0, 1, \mathrm{g_t}, rule\rangle, \Phi)$. If $\mathrm{f_t}$ and $\mathrm{g_t}$ are isomorphic it holds that *FEVBDDeval*$(\langle 0, 1, \mathrm{f}, rule\rangle, \Phi) =$ *FEVBDDeval*$(\langle 0, 1, \mathrm{g}, rule\rangle, \Phi) = val$. Thus we have that $ev_f + w_{t_f} \cdot val \neq ev_g + w_{t_g} \cdot val$. If $\mathrm{f_t}$ and $\mathrm{g_t}$ are nonisomorphic then lemma 3.1 is applicable. Almost the identical prove can be given for $ev_f = ev_g$ and $w_{e_f} \neq w_{e_g}$. If $ev_f = ev_g$, $w_{t_f} = w_{t_g}$, and $w_{e_f} = w_{e_g}$ either $\mathrm{f_t}$ and $\mathrm{g_t}$, or $\mathrm{f_e}$ and $\mathrm{g_e}$ are nonisomorphic.

Subcase 1: If $\mathrm{f_t}$ and $\mathrm{g_t}$ are nonisomorphic, then from $n -$ *index*(*variable*$(\mathrm{f_t})) < k$,$n -$ *index*(*variable*$(\mathrm{g_t})) < k$, and the induction hypothesis, we see that there exists some Φ such that *FEVBDDeval*$(\langle 0, 1, \mathrm{f_t}, rule\rangle, \Phi) \neq$ *FEVBDDeval*$(\langle 0, 1, \mathrm{g_t}, rule\rangle, \Phi)$. Now let Φ' be defined as $\Phi'(x_{n-k}) = 1$ and $\Phi'(x_i) = \Phi(x_i), \forall i \neq n-k$, then *FEVBDDeval*$(\langle 0, 1, \mathrm{f}, rule\rangle, \Phi') = ev_f + w_{t_f} \cdot$ *FEVBDDeval*$(\langle 0, 1, \mathrm{f_t}, rule\rangle, \Phi') \neq ev_g + w_{t_g} \cdot$ *FEVBDDeval*$(\langle 0, 1, \mathrm{g_t}, rule\rangle, \Phi') =$ *FEVBDDeval*$(\langle 0, 1, \mathrm{g}, rule\rangle, \Phi')$.

Subcase 2: Otherwise $\mathbf{f_e}$ and $\mathbf{g_e}$ are nonisomorphic, then by similar arguments, letting $\Phi'(x_{n-k}) = 0$ and $\Phi'(x_i) = \Phi(x_i), \forall i \neq n-k$, will result in *FEVBDDeval*$(\langle 0, 1, \mathbf{f}, rule\rangle, \Phi') \neq$ *FEVBDDeval*$(\langle 0, 1, \mathbf{g}, rule\rangle, \Phi')$.

Case 2 : $n -$ *index(variable(***g***))* $< k$

By definition of a reduced FEVBDD, we cannot have $ev_f = 0$, $w_{t_f} = w_{e_f} = 1$ and $\mathbf{f_t}$ being isomorphic to $\mathbf{f_e}$. If $ev_f \neq 0$, let $\Phi(x_{n-k}) = 1$ and $\Phi(x_i) = 0$, $\forall i \neq n-k$, then *FEVBDDeval*$(\langle 0, 1, \mathbf{f}, rule\rangle, \Phi) = ev_f \neq 0 =$ *FEVBDDeval*$(\langle 0, 1, \mathbf{g}, rule\rangle, \Phi)$, since $\mathbf{g}$ is independent of the first $n-k$ bits. If $ev_f = 0$ and $w_{t_f} \neq w_{e_f}$ then let Φ be an assignment such that $\Phi(x_{n-k}) = 1$ and $\Phi(x_i) = 0$, $\forall((i \neq n-k) \wedge (i < n -$ *index*$($*variable*$(\mathbf{g}))))$. Furthermore, let Φ be such that *FEVBDDeval*$(\langle 0, 1, \mathbf{f}, rule\rangle, \Phi) = val_f \neq 0$ and *FEVBDDeval*$(\langle 0, 1, \mathbf{g}, rule\rangle, \Phi) = val_g \neq 0$. If the corresponding subgraph of $\mathbf{f}$ with top-variable x_{n-k} and $\mathbf{g}$ are isomorphic then it holds that $val_f = w_{t_f} \cdot val_g$. If the graphs are non-isomorphic we can apply the same reasoning as we did in the proof of lemma 3.1. Otherwise, $\mathbf{f_t}$ and $\mathbf{f_e}$ are non-isomorphic and at least one of them is not isomorphic to $\mathbf{g}$. If $\mathbf{f_t}$ and $\mathbf{g}$ are non-isomorphic, then by induction hypothesis, there exists an assignment Φ such that *FEVBDDeval*$(\langle 0, 1, \mathbf{f_t}, rule\rangle, \Phi) \neq$ *FEVBDDeval*$(\langle 0, 1, \mathbf{g}, rule\rangle, \Phi)$. Now, let $\Phi'(x_{n-k}) = 1$ and $\Phi'(x_i) = \Phi(x_i), \forall i \neq n-k$. It holds that *FEVBDDeval*$(\langle 0, 1, \mathbf{f_t}, rule\rangle, \Phi') \neq$ *FEVBDDeval*$(\langle 0, 1, \mathbf{g}, rule\rangle, \Phi')$. □

As shown above, FEVBDDs form a canonical representation of a function only for specific weight normalizing rules that uniquely determine how the node weight of a new node is computed based on its both descendants. We propose two basic rules that can be used to guarantee canonicity for FEVBDDs. Given two FEVBDDs $\langle c_t, w_t, \mathbf{t}, rule_t\rangle$ and $\langle c_e, w_e, \mathbf{e}, rule_e\rangle$ with $rule_t = rule_e = rule$ the node weight w of $\langle c, w, \mathbf{f}, rule\rangle$ is computed as follows

1. GCD rule:

$$\begin{aligned} w &= \text{gcd}(c_t - c_e, w_t, w_e) \\ &= \text{gcd}(c_t - c_e, \text{gcd}(w_t, w_e)) \\ sign(w) &= \begin{cases} sign(w_e) & \text{if } w_e \neq 0 \\ sign(w_t) & \text{if } w_t \neq 0 \wedge w_e = 0 \\ sign(c_t - c_e) & \text{if } w_t = w_e = 0 \end{cases} \end{aligned}$$

2. RATIONAL rule:

$$w = \begin{cases} w_e & \text{if } w_e \neq 0 \\ w_t & \text{if } w_t \neq 0 \wedge w_e = 0 \\ (c_t - c_e) & \text{if } w_t = w_e = 0 \end{cases}$$

```
make_new_node($x_i$,$\langle c_T, w_T, \mathbf{T}, rule_T\rangle$,$\langle c_E, w_E, \mathbf{E}, rule_E\rangle$)
{
    if{$\langle c_T, w_T, \mathbf{T}, rule_T\rangle = \langle c_E, w_E, \mathbf{E}, rule_E\rangle$}
        return($\langle c_T, w_T, \mathbf{T}, rule_T\rangle$);
    /* compute the new weights */
    $c = c_E$;
    $ev = c_T - c_E$;
    $w = \textit{norm\_weight}(ev, w_T, w_E)$;
    $w_t = w_T/w$;
    $w_e = w_E/w$;
    /* guarantee uniqueness */
    $\langle c_h, w_h, \mathbf{h}, rule_h\rangle$=find_or_add($x_i, c, w, ev, w_t, w_e, \mathbf{T}, \mathbf{E}$);
    return $\langle c_h, w_h, \mathbf{h}, rule_h\rangle$ ;
}
```

Table 2: *Make_New_Node*

These weight normalizing rules (cf. Table 3) are applied whenever a new node is generated using the *make_new_node* routine. (cf. Table 2). This routine enforces both the canonicity of the function graph as well as its uniqueness.

The routine *find_or_add* preserves the uniqueness of all nodes. Before a new node is actually created a quick hash table lookup is performed and, if the node is already a member of the table, the stored node with its unique ID is returned. Otherwise, a new node entry in the hash table is created and the new node with its unique ID is returned. Thus it is guaranteed that every node is stored only once in the hash table.

Although the GCD rule requires a multiplicative weight to be associated with both the true- and the else-edges, there are some cases where it might be the rule of choice. If the function range is purely integer the GCD rule avoids dealing with fractions. This is particularly valuable, since all arithmetic operations on fractions are significantly more time consuming than the built in hardware routines for integers. Furthermore, the restriction to integers by use of the GCD rule brings a clear advantage in memory efficiency. Even though we need to store an additional weight, the memory consumption per node is less than when using the rational rule which requires the use of fractions. This is because every fraction is internally represented as one integer for the numerator and one

```
norm_weight($ev, w_T, w_E$)
{
    switch(mode) {
    case 'GCD':
        if($w_E \neq 0$)
            sign = $sign(w_E)$;
        else if($w_T \neq 0$)
            sign = $sign(w_T)$;
        else sign = $sign(ev)$;
        return(sign · gcd$(ev, w_T, w_E)$) ;
    break ;
    case 'RATIONAL':
        if($w_E \neq 0$)
            return $w_E$;
        else if($w_T \neq 0$)
            return $w_T$;
        else return $ev$;
    break;
}
```

Table 3: *Norm_Weight*

for the denominator. Of course, as soon as the application requires the use of fractions the rational rule should be preferred. Nevertheless, the GCD rule is still applicable since we define:

$$\text{gcd}(\frac{u}{u'}, \frac{v}{v'}) = \frac{\text{gcd}(u, v)}{\text{gcd}(u', v')}$$

3.1 Operations

As has been done for OBDDs [5] and EVBDDs [14], we provide a generic algorithm *apply* that implements arbitrary arithmetic operations on two FEVBDDs (cf. Table 4). *Apply* takes two FEVBDDs $\langle c_f, w_f, \mathbf{f}, rule_f \rangle$ and $\langle c_g, w_g, \mathbf{g}, rule_g \rangle$, as well as an operation *op* as its arguments. Both FEVBDDs have to be based on the same weight normalizing rule. The algorithm recursively branches at the top variable, i.e. the variable with the least index in **f** or **g** until it reaches a terminal case. Terminal cases depend on the operation *op*; as an example, for *op*='+' we have the terminal case $\langle c_f, w_f, \mathbf{f}, rule_f \rangle + \langle c, 0, \mathbf{0}, rule \rangle$.

The computational efficiency of this algorithm can be improved significantly by taking advantage of a computation cache. Before the recursive process is started, a quick lookup in the computation cache is performed and if successful, then the result of *op* is returned immediately without further computation. The entries of the cache are uniquely identified by a key consisting of the operands $\langle c_f, w_f, \mathbf{f}, rule_f \rangle$ and $\langle c_g, w_g, \mathbf{g}, rule_g \rangle$ and the operation *op*. Whenever a new result is computed it is stored in the computation cache. In general the complexity of operations performed by *apply* is $O(\|\langle c_f, w_f, \mathbf{f}, rule_f \rangle\| \cdot \|\langle c_g, w_g, \mathbf{g}, rule_g \rangle\|)$.

As mentioned before we can further improve the computational complexity of *apply* by making use of properties of specific operations. We adapt the concept of an *additive property* proposed for EVBDDs by Lai, et al., [14] and extend it to the so-called *affine property* for FEVBDDs.

Definition 3.4 *An operator op applied to* $\langle c_f, w_f, \mathbf{f}, rule_f \rangle$ *and* $\langle c_g, w_g, \mathbf{g}, rule_g \rangle$ *is said to satisfy the affine property if*

$$(c_f + w_f \cdot f)\mathit{op}(c_g + w_g \cdot g) = (c_f \ \mathit{op} \ c_g) + w \cdot ((w_f' \cdot f)\mathit{op}(w_g' \cdot g))$$

The factor w *is defined as* $w = \text{gcd}(w_f, w_g)$ *and can be of arbitrary value.*[3]

[3]Similar to the rational rule we can alternatively define the *affine property* as follows:

$$(c_f + w_f \cdot f)\mathit{op}(c_g + w_g \cdot g) = (c_f \ \mathit{op} \ c_g) + w_f \cdot (f\mathit{op}(\frac{w_g}{w_f} \cdot g)).$$

All the benefits of the *affine property* remain the same.

```
apply($\langle c_f, w_f, \mathbf{f}, rule_f\rangle$,$\langle c_g, w_g, \mathbf{g}, rule_g\rangle$,op) {
    /* check for a terminal case */
    if(terminal_case($\langle c_f, w_f, \mathbf{f}, rule_f\rangle$,$\langle c_g, w_g, \mathbf{g}, rule_g\rangle$,op))
        return ($\langle c_f, w_f, \mathbf{f}, rule_f\rangle$op$\langle c_g, w_g, \mathbf{g}, rule_g\rangle$);
    /* is the result of op already available in the computation cache */
    if(comp_table_lookup($\langle c_f, w_f, \mathbf{f}, rule_f\rangle$,$\langle c_g, w_g, \mathbf{g}, rule_g\rangle$,op,$\langle c_{ans}, w_{ans}, \mathbf{ans}, rule_{ans}\rangle$))
        return ($\langle c_{ans}, w_{ans}, \mathbf{ans}, rule_{ans}\rangle$);
    /* perform the recursive computation of op*/
    if($index(\mathbf{f}) \geq index(\mathbf{g})$) {
        $\langle c_{g_t}, w_{g_t}, \mathbf{g_t}, rule_{g_t}\rangle = \langle c_g + w_g \cdot ev_g, w_g \cdot w_{t_g}, child_{\mathbf{t}}(\mathbf{g}), rule\rangle$;
        $\langle c_{g_e}, w_{g_e}, \mathbf{g_e}, rule_{g_e}\rangle = \langle c_g, w_g \cdot w_{e_g}, child_{\mathbf{e}}(\mathbf{g}), rule\rangle$;
        $var = variable(\mathbf{g})$;
    }
    else {
        $\langle c_{g_t}, w_{g_t}, \mathbf{g_t}, rule_{g_t}\rangle = \langle c_{g_e}, w_{g_e}, \mathbf{g_e}, rule_{g_e}\rangle = \langle c_g, w_g, \mathbf{g}, rule_g\rangle$;
        $var = variable(\mathbf{f})$;
    }
    if($index(\mathbf{f}) \leq index(\mathbf{g})$) {
        $\langle c_{f_t}, w_{f_t}, \mathbf{f_t}, rule_{f_t}\rangle = \langle c_f + w_f \cdot ev_f, w_f \cdot w_{t_f}, child_{\mathbf{t}}(\mathbf{f}), rule\rangle$;
        $\langle c_{f_e}, w_{f_e}, \mathbf{f_e}, rule_{f_e}\rangle = \langle c_f, w_f \cdot w_{e_f}, child_{\mathbf{e}}(\mathbf{f}), rule\rangle$;
    }
    else {
        $\langle c_{f_t}, w_{f_t}, \mathbf{f_t}, rule_{f_t}\rangle = \langle c_{f_e}, w_{f_e}, \mathbf{f_e}, rule_{f_e}\rangle = \langle c_f, w_f, \mathbf{f}, rule_f\rangle$;
    }
    $\langle c_{h_t}, w_{h_t}, \mathbf{h_t}, rule_{h_t}\rangle = apply(\langle c_{f_t}, w_{f_t}, \mathbf{f_t}, rule_{f_t}\rangle, \langle c_{g_t}, w_{g_t}, \mathbf{g_t}, rule_{g_t}\rangle, op)$;
    $\langle c_{h_e}, w_{h_e}, \mathbf{h_e}, rule_{h_e}\rangle = apply(\langle c_{f_e}, w_{f_e}, \mathbf{f_e}, rule_{f_e}\rangle, \langle c_{g_e}, w_{g_e}, \mathbf{g_e}, rule_{g_e}\rangle, op)$;
    if($\langle c_{h_t}, w_{h_t}, \mathbf{h_t}, rule_{h_t}\rangle = \langle c_{h_e}, w_{h_e}, \mathbf{h_e}, rule_{h_e}\rangle$ )
        return($\langle c_{h_t}, w_{h_t}, \mathbf{h_t}, rule_{h_t}\rangle$);
    $\langle c_h, w_h, \mathbf{h}, rule_h\rangle = make\_new\_node(var, \langle c_{h_t}, w_{h_t}, \mathbf{h_t}, rule_{h_t}\rangle, \langle c_{h_e}, w_{h_e}, \mathbf{h_e}, rule_{h_e}\rangle)$;
    /* store the result in the computation cache */
    comp_table_insert($\langle c_f, w_f, \mathbf{f}, rule_f\rangle$,$\langle c_g, w_g, \mathbf{g}, rule_g\rangle$,op,$\langle c_h, w_h, \mathbf{h}, rule_h\rangle$);
    return ($\langle c_h, w_h, \mathbf{h}, rule_h\rangle$);
}
```

Table 4: *Apply*

Operations that satisfy the affine property are addition, subtraction, scalar multiplication and logical bit shifting. The main advantage of the affine property lies in reducing the computational complexity of *apply*. Since we can separately compute the parts of the result generated by the constants c_f and c_g and by the two subgraphs $\langle 0, w_f, \mathbf{f}, rule \rangle$ and $\langle 0, w_g, \mathbf{g}, rule \rangle$, the hit ratio of the computation cache can be drastically increased by separating the influence of the constants and always storing only the results for $c = 0$. This concept is applied to every recursion step so that the constant value is never passed down to the next recursion level. Unfortunately, we still have to pass the multiplicative weights w_f and w_g since they cannot be separated from the functions f and g. To achieve a further improvement in the hit ratio, we extract the common divisor w from w_f and w_g and promote only w'_f and w'_g. This is an advantage in such cases as reducing the problem of performing $(7 + 8 \cdot f)$*op*$(4 + 6 \cdot g)$ to the already computed problem $(0 + 4 \cdot f)$*op*$(0 + 3 \cdot g)$. Since we cannot quantify the influence of the GCD extraction the worst case computational complexity for operations satisfying the affine property is given as $O(|\langle c_{f'}, \mathbf{f}' \rangle| \cdot |\langle c_{g'}, \mathbf{g}' \rangle|)$ where $\langle c_{f'}, \mathbf{f}' \rangle$ and $\langle c_{g'}, \mathbf{g}' \rangle$ denote the EVBDDs corresponding to the FEVBDDs $\langle c_f, w_f, \mathbf{f}, rule_f \rangle$ and $\langle c_g, w_g, \mathbf{g}, rule_g \rangle$, respectively.

Scalar multiplication and logical-bit shifting offer a better computational complexity since they can be computed in time independent of the size of the function graph. Scalar multiplication only requires the weights of the root node to be multiplied. In the case of EVBDDs we have to multiply every edge weight with the scalar; a task of complexity $O(|\mathbf{f}|)$.

Since multiplication does not satisfy the affine property we are basically required to use the original version of *apply*. For the multiplication of two functions that both have a high percentage of reconverging branches, the following approach tends to improve the cache efficiency:

$$\begin{aligned} \langle c_f, w_f, \mathbf{f}, rule_f \rangle \cdot \langle c_g, w_g, \mathbf{g}, rule_g \rangle &= \langle c_f \cdot c_g, 0, \mathbf{0}, rule \rangle + \langle 0, c_f \cdot w_g, \mathbf{g}, rule \rangle + \\ &\quad \langle 0, c_g \cdot w_f, \mathbf{f}, rule \rangle + \langle 0, w_f \cdot w_g, \mathbf{f} \cdot \mathbf{g}, rule \rangle \\ &= \langle c_h, w_h, \mathbf{h}, rule_h \rangle \end{aligned}$$

We now have only $O(|\langle c_f, w_f, \mathbf{f}, rule_f \rangle| \cdot |\langle c_g, w_g, \mathbf{g}, rule_g \rangle|)$ calls to *multiply* but every call requires three calls to *apply* for adding the separate terms. The first addition is not costly since the first term is always a constant, however, the second and third addition are potentially costly.

In addition to the *additive property*, two further properties – the *bounding property* and the *domain-reducing property* – have been introduced by Lai, et al. [14] [12]. As has been done for the *additive property*, these properties can be easily adapted to FEVBDDs.

3.2 Representation of Boolean Functions

Boolean Functions are represented in FEVBDDs by encoding the boolean values *true* and *false* as integers 1 and 0, respectively. All the basic boolean operations can be easily represented using only arithmetic operations. Thus we can easily represent any boolean function using FEVBDDs. Although we could implement the boolean operations based on their corresponding arithmetic functions, it is by far better in terms of computational complexity to directly use *apply* for boolean operations. All we need to do is to provide the necessary terminal cases for *apply*($\langle c_f, w_f, \mathbf{f}, rule_f\rangle$, $\langle c_g, w_g, \mathbf{g}, rule_g\rangle$, *boolean_op*). In the case of the boolean conjunction operation for example the terminal cases are:

1. if($\langle c_f, w_f, \mathbf{f}, rule_f\rangle = \langle 0, 0, \mathbf{0}, rule\rangle$ or $\langle c_g, w_g, \mathbf{g}, rule_g\rangle = \langle 0, 0, \mathbf{0}, rule\rangle$) return $\langle 0, 0, \mathbf{0}, rule\rangle$
2. if($\langle c_f, w_f, \mathbf{f}, rule_f\rangle = \langle 1, 0, \mathbf{0}, rule\rangle$) return $\langle c_g, w_g, \mathbf{g}, rule_g\rangle$
3. if($\langle c_g, w_g, \mathbf{g}, rule_g\rangle = \langle 1, 0, \mathbf{0}, rule\rangle$) return $\langle c_f, w_f, \mathbf{f}, rule_f\rangle$

To convert a boolean function from its OBDD to its FEVBDD representation we can adapt the algorithm suggested by Lai in [14]. Additionally, the concept of multiplicative weights allows us to directly represent the so called complement edges, so that we need to take care of this case in the algorithm:

1. convert the terminal node **0** to $\langle 0, 0, \mathbf{0}, rule\rangle$ and **1** to $\langle 1, 0, \mathbf{0}, rule\rangle$.
2. for each nonterminal node $\langle x_i, \mathbf{t}, \mathbf{e}\rangle$ in the OBDD such that **t** and **e** have already been converted to FEVBDDs as $\langle c_{t'}, w_{t'}, \mathbf{t}', rule_{t'}\rangle$ and $\langle c_{e'}, w_{e'}, \mathbf{e}', rule_{e'}\rangle$, the following conversion rules are applied:
3. if the branch leading from node $\langle x_i, \mathbf{t}, \mathbf{e}\rangle$ to **t** or **e** is a complement edge we have to perform the complementation by computing $1 - t$ or $1 - e$, respectively. This is achieved by multiplying both weights c_t (c_e) and w_t (w_e) by -1 and later adding 1 to c_t (c_e). The four basic conversion rules are listed below:
   - $\langle x_i, \langle 0, 1, \mathbf{t}', rule\rangle, \langle 0, 1, \mathbf{e}', rule\rangle\rangle \rightarrow \langle 0, 1, \langle x_i, \mathbf{t}', \mathbf{e}', 0, 1, 1\rangle\rangle$
   - $\langle x_i, \langle 0, 1, \mathbf{t}', rule\rangle, \langle 1, 1, \mathbf{e}', rule\rangle\rangle \rightarrow \langle 1, 1, \langle x_i, \mathbf{t}', \mathbf{e}', -1, 1, 1\rangle\rangle$
   - $\langle x_i, \langle 1, 1, \mathbf{t}', rule\rangle, \langle 0, 1, \mathbf{e}', rule\rangle\rangle \rightarrow \langle 0, 1, \langle x_i, \mathbf{t}', \mathbf{e}', 1, 1, 1\rangle\rangle$
   - $\langle x_i, \langle 1, 1, \mathbf{t}', rule\rangle, \langle 1, 1, \mathbf{e}', rule\rangle\rangle \rightarrow \langle 1, 1, \langle x_i, \mathbf{t}', \mathbf{e}', 0, 1, 1\rangle\rangle$

The above conversion rules are not complete in the case of FEVBDDs since we can now also have variations in the multiplicative weights which can either be $+1$ or -1. These cases however are handled exactly according to the norm weighting rule that has been presented before, so that we do not explicitly list them here.

As it has been done for EVBDDs [14], it can be shown that the following theorems hold.

Theorem 3.2 *Given an OBDD representation* **v** *of a boolean function with complement edges being allowed and an FEVBDD* $\langle c_v, w_v, \mathbf{v}, rule_v \rangle$*, then* **v** *and* **v′** *have the same topology except that the terminal node* **1** *is absent from the FEVBDD* **v′** *and the edges connected to it are redirected to the terminal node* **0**.

Theorem 3.3 *Given two OBDDs* **f** *and* **g** *with complement edges being allowed and the corresponding FEVBDDs* $\langle c_{f'}, w_{f'}, \mathbf{f}', rule_{f'} \rangle$ *and* $\langle c_{g'}, w_{g'}, \mathbf{g}', rule_{g'} \rangle$*, the time complexity of boolean operations on FEVBDDs (using apply) is* $O(|\mathbf{f}| \cdot |\mathbf{g}|) = O(|\mathbf{f}'| \cdot |\mathbf{g}'|)$.

An example of a FEVBDD representing a boolean function with complement edges is given in Figure 5. This FEVBDD represents the four output functions of a 3-bit adder. It has the same topology (except for the terminal edges) as the corresponding OBDD depicted in the same figure. As it is shown in this example, FEVBDDs successfully extend the use of EVBDDs to represent boolean functions as they inherently offer a way to represent complement edges. Furthermore, the boolean operation 'not' can now be performed in constant time since it only requires manipulation of the weights of the root node.

Figure 5 goes here.

3.3 Logic Verification

The purpose of logic verification is to formally prove that the actual implementation satisfies the conditions defined by the specification. This is done by formally showing the equivalence between the combinational circuit, i.e. the description of the design and the specification of the intended behaviour.

In general, the implementation is represented by an array of boolean functions $\mathbf{f}_b$ and the specification is given by a word-level function f_w. In order to transform the bit-level representation to the word-level we can use any encoding function to encode the binary input signals to the circuit.

The set of input signals is partitioned into several subsets of binary signals $\mathrm{x}^0, \ldots, \mathrm{x}^n$ and every array x^i is then encoded using an encoding function *encode*$_i$ that provides a word-level interpretation of the binary input signals. Common encoding functions are signed-integer, one's-complement and two's-complement. The corresponding FEVBDDs are shown in Figure 2. Thus, the implementation can be described by an array of boolean functions $\mathbf{f}_b(\mathrm{x}^0, \ldots, \mathrm{x}^n)$. The specification is given as a word-level function $f_w(X_1, \ldots, X_n)$. Verification is then done by proving the equivalence between an encoding of the binary output signals of the circuit, i.e. the array of boolean functions, and the word-level function of the encoded input signals:

$$\mathit{encode}_{out}(\mathbf{f}_b(\mathrm{x}^0, \ldots, \mathrm{x}^n)) = f_w(\mathit{encode}_0(\mathrm{x}^0), \ldots, \mathit{encode}_n(\mathrm{x}^n))$$

This strategy for logic verification was first proposed by Lai, et al., using EVBDDs [12][14]. Since FEVBDDs can describe both bit-level and word-level functions, they can be successfully applied to logic verification.

Although all word-level operations can be represented by FEVBDDs, the space complexity of certain operations becomes exponential so that their application is limited to small word-length.

Both EVBDD and FEVBDD representations of word-level multiplication are exponential; FEVBDDs however offer significant savings in memory consumption over EVBDDs. As can be seen in Figure 4 for word-level multiplication of two three-bit integers, the EVBDD contains 28 internal nodes whereas the FEVBDD representation requires only 10 nodes. In general, the EVBDD denoting the multiplication of two n-bit integers has $(n+1)(2^n-1)$ internal nodes. The corresponding FEVBDD contains only $n+(2^n-1)$ internal nodes and the ratio of EVBDD nodes to FEVBDD nodes is $\frac{n+1}{1+\frac{n}{2^n-1}}$. As can be seen from this ratio, the savings in the number of nodes in the FEVBDD representation are of order n. As an example, a 16-bit multiplier requires 1,114,095 EVBDD nodes but only 65,551 FEVBDD nodes. Even if we take into account that a FEVBDD node requires 20 bytes versus only 12 bytes per EVBDD node, the savings remain significant (EVBDD:13.3 Mbyte, FEVBDD: 1.3 Mbyte).

As has been done for EVBDDs [14], FEVBDDs can also be extended to structured FEVBDDs which allow the modeling of conditional expressions and vectors.

3.4 Integer Linear Programming

An algorithm FGILP for solving Integer Linear Programming (ILP) problems based on EVBDDs has been proposed by Lai, et al. in [15]. FGILP realizes an ILP solver based on function graphs, which uses a mixed branch-and-bound/implicit-enumeration strategy. It has been shown that this approach can successfully compete with other branch-and-bound strategies that require the solution of the corresponding Linear Programming problems. The latter strategy is the one most widely applied in commercial programs.

An ILP problem can be formulated as follows:

$$\textit{minimize} \quad \sum_{i=1}^{n} c_i x_i \tag{5}$$

$$\text{subject to} \quad \sum_{i=1}^{n} a_{ij} x_i \leq b_j, 1 \leq j \leq m, \tag{6}$$

$$\text{with } x_i \text{ integer}$$

Since both EVBDDs and FEVBDDs allow only binary decision variables, the encodings shown in Figure 2 have to be applied. A 32-bit integer, for example, can be represented by an EVBDD or FEVBDD with 32 nodes. Since FEVBDDs form an extension of EVBDDs we can also apply FEVBDDs to solve ILP problems. We expect a reduction in the memory requirement for FGILP when using FEVBDDs. This is due to the fact that different multiples of the integer variables x_i appear in equations (5) and (6). If we use EVBDDs to represent these multiples of x_i, we have to build an EVBDD for every different coefficient a_{ij} since scalar multiplication on EVBDDs is performed by multiplying all edge weights with the factor. If we use FEVBDDs, however, we only have to store the FEVBDD representing x_i once. Multiples of x_i can be easily realized by associating the corresponding multiplicative edge weights with dangling incoming edges leading to x_i. As an example, storing 6x, 7x and 5x requires 96 nodes if we use EVBDDs but only 32 nodes if we apply FEVBDDs.

3.5 Implementation of Arbitrary Precision Arithmetic

The introduction of multiplicative weights in combination with the *RATIONAL rule* for weight normalizing makes it necessary to extend the value range of the edge weights from the integer domain to the rational domain. This is done in a way such that any future expansion to other domains such as the complex domain can be easily achieved. All operations on edge weights are accessed through a standardized interface that invokes the specified function and then executes the requested operation depending on the current mode. Thus, the FEVBDD code remains fully independent of the selected domain. By changing to another mode we can easily switch from the integer domain to the rational domain, for example. This means we can still use the fast routines for single precision integers when necessary.

Multiple precision integers are realized as arrays of integers and the arithmetic operations are implemented based on the algorithms for multiple precision arithmetic given by Knuth in [11]. Multiple precision fractions are implemented as arrays of two multiple precision integers where one integer represents the numerator and the other one the denominator. It is enforced by the package that the numerator and denominator remain relative prime and only the numerator can be signed. This is achieved by computing the greatest common divisor (GCD) of numerator and denominator and dividing both the numerator and denominator by the GCD. This operation is performed whenever an input is given. Internally the data is guaranteed to remain in the normalized form as this form is strictly enforced by all operations. Thus, a rational value is always uniquely represented by the numerator and denominator.

The GCD can be computed very fast by Euclid's algorithm or the binary GCD algorithm [11]. For multi-precision fractions we use the binary GCD algorithm since it works very fast for integers of multiple word length. It only relies on subtraction and right shifting and does not require division operations. For single word precision fractions we employ the classical version of Euclid's algorithm since division can be executed very efficiently for single word integers. The basic arithmetic operations for fractions are realized as follows:

- multiplication:
$$U \cdot V = \frac{u}{u'} \cdot \frac{v}{v'} = \frac{\frac{u}{d_1} \cdot \frac{v}{d_2}}{\frac{u'}{d_2} \cdot \frac{v'}{d_1}}$$
where $d_1 = \text{gcd}(u, v')$ and $d_2 = \text{gcd}(u', v)$

- division:
$$\frac{U}{V} = \frac{\frac{u}{u'}}{\frac{v}{v'}} = \frac{u}{u'} \cdot \frac{v'}{v} = \frac{\frac{u}{d_1} \cdot \frac{v'}{d_2}}{\frac{u'}{d_2} \cdot \frac{v}{d_1}}$$
where $d_1 = \text{gcd}(u, v)$ and $d_2 = \text{gcd}(u', v')$

- addition:
$$U + V = \frac{u}{u'} + \frac{v}{v'} = \begin{cases} \frac{u \cdot v' + v \cdot u'}{u' \cdot v'} & \text{if } d_1 = 1 \\ \frac{\frac{t}{d_2}}{\frac{u'}{d_1} \cdot \frac{v'}{d_2}} & \text{if } d_1 > 1 \end{cases}$$
where $d_1 = \text{gcd}(u', v')$ and if $d_1 > 1$: $t = u \cdot \frac{v'}{d_1} + v \cdot \frac{u'}{d_1}$, $d_2 = \text{gcd}(t, d_1)$

3.5.1 Symbolic Operations and Finite Fields

FEVBDDs are not constrained to integer valued functions. As one can already see in the use of a rational rule, we can easily represent functions with rational function values. Complex values are also feasible; additionally, we can use symbolic computation. Even though the value ranges can be extended by using rational or complex edge weights, the decision variables still have to be binary. Thus, if we want to represent linear functions containing variables from the above value ranges, we have to encode them binarily such as it has been done for integers. Generally this approach leads to a means to represent any function on finite fields by FEVBDDs as it has been proposed for ADDs [2]. In this case the FEVBDD $\langle c_f, w_f, \mathbf{f}, rule_f \rangle$ more generally represents the function

$$c_f \oplus w_f \odot (ITE(x, (ev \oplus w_t \odot f_t), (w_e \oplus f_e)))$$

where $\oplus$ and $\odot$ denote operations on the finite field. The ITE operator acts as a switch that either selects the subfunction denoted by the true- or else-edge. Contrary to the ADD approach we can exploit relationships between the subgraphs.

4 Matrix Representation and Manipulation

Matrices have been successfully represented using MTBDDs [8] [9] and ADDs [2] and implementations of the basic matrix operations such as addition and multiplication have been given. A popular class of matrices that can be efficiently represented by MTBDDs and EVBDDs is the class of Walsh matrices which can be generated by a recursive rule.

4.1 Representation of Matrices

The basic idea in using function graphs to represent matrices is to encode both the row and column position of the matrix elements using binary variables. An 8 $\times$ 8 matrix, for example, requires 3 binary variables for the rows and another 3 for the columns. Basically, we can view the problem of representing a $m \times n$ matrix as representing a function from the finite set $D = \{0, \ldots, m-1\} \times \{0, \ldots, n-1\}$ of all element positions to the finite set R of its elements. The binary variables giving the row position are called *row designators* $x \in \{x_0, \ldots, x_{m-1}\}$, the ones denoting the column position are called *column designators* $y \in \{y_0, \ldots, y_{n-1}\}$. For the imposed variable ordering row and column designators are mixed together such that the order is $\{x_0, y_0, \ldots, x_{m-1}, y_{n-1}\}$. Because of this chosen variable ordering subtrees in the function graph directly correspond to submatrices in the given matrix, as can be seen in Figure 6. Based on this correspondence the pseudo-boolean function denoting the matrix M can be given easily:

$$\begin{array}{rcllllll} f_M & = & f_{M_{\overline{xy}}} & (1-x) & (1-y) & + & f_{M_{\overline{x}y}} & (1-x) \quad y \quad + \\ & & f_{M_{x\overline{y}}} & x & (1-y) & + & f_{M_{xy}} & x \quad y \end{array}$$

Figure 6 goes here.

Furthermore, this ordering allows matrices to be be represented compactly if they have submatrices that are identical (MTBDDs) or can be transformed into each other by an affine transformation[4] (FEVBDDs). Since the concept of square matrices, i.e. vertical size m = horizontal size n, helps to keep many algorithms efficient and simple we will from now on only consider square matrices with $size = \max(m, n)$. To make non-square matrices square we can easily pad them with rows or columns filled with zeros. This does not significantly increase our memory consumption for storing the matrix since the padded blocks are uniform and can therefore be represented by only a few nodes.

As it has already been mentioned, MTBDDs only offer a compact and memory efficient representation of matrices that feature identical subblocks. They require a different terminal node for each distinct matrix element. FEVBDDs can do far better than that. The concept of FEVBDDs allows two subblocks to be represented by the same subgraph if they differ only by an affine transformation of their elements. We will now introduce a special class of matrices that can always be represented by a FEVBDD of linear size. For this class of matrices the sizes of the corresponding MTBDD, EVBDD and *BMD are likely to be exponential.

Definition 4.1 *A recursively-affine matrix is recursively generated using the following rules:*

1. *we begin with a* 1×1 *matrix* $M_0 = [c_0]$ *where* c_0 *is a integer or rational constant value*
2. *in every recursion step a new matrix* M_{n+1} *is created based on the previous result* M_n *such that:*

$$M_{n+1} = \begin{bmatrix} k_0 + w_0 \cdot M_n & k_1 + w_1 \cdot M_n \\ k_2 + w_2 \cdot M_n & k_3 + w_3 \cdot M_n \end{bmatrix}$$

with $k_0, \ldots, k_3$ *and* $w_0, \ldots, w_3$ *being arbitrary integer or rational numbers.*

[4]An affine transformation is a transformation of the form $y \rightarrow a \cdot y + b$.

Figure 7 shows the general structure of the FEVBDD that corresponds to a recursion step in building up a recursively affine matrix. In every recursion step a structure as shown in Figure 7 is added to the already constructed FEVBDD.

Figure 7 goes here.

As can be seen from Figure 7 we only need $3 \cdot \lceil \log_2(n) \rceil$ nodes to represent a recursively-affine matrix of size n x n. As an example of a recursively affine matrix we build in Figure 8 the FEVBDD for the matrix M given below:

$$M = \begin{bmatrix} 3 & 10 & 14 & 35 \\ 9 & 5 & 32 & 20 \\ 12 & 26 & 22 & 64 \\ 24 & 16 & 58 & 34 \end{bmatrix}$$

Figure 8 goes here.

An important class of matrices that belongs to the family of recursively-affine matrices is the set of Walsh matrices in the Hadamard ordering [17]. These matrices can be used to compute spectral transforms of boolean functions. They are recursively defined as follows:

$$\begin{aligned} H_1^h &= [1] \\ H_{n+1}^h &= \begin{bmatrix} H_n^h & H_n^h \\ H_n^h & -H_n^h \end{bmatrix} \end{aligned}$$

Figure 9 shows both the FEVBDD and EVBDD representations of the Walsh matrix H_3^h. As can be seen in Figure 9, the size of the FEVBDD representation is $2 \cdot n$ where n denotes the order of the Walsh matrix. The size of the EVBDD representation is $4 \cdot n - 2$

Figure 9 goes here.

Generally speaking, employing function graphs such as MTBDDs or FEVBDDs to represent sparse matrices offers the following advantages:

1. In comparison with normal sparse data structures, function graphs do provide a uniform $\log_2(N)$ access time, where N is the number of real elements being stored in the function graph (for example, all non-zero elements of a sparse matrix)

2. Function graphs may not be able to beat sparse-matrix data structures in terms of worst space complexity. However, recombination of isomorphic subgraphs may give a considerable practical advantage to function graphs over other data structures. This is particularly valid for FEVBDDs since the same subgraph can represent all the matrices that can be generated by an affine transformation of the matrix represented by the subgraph.

4.2 Operations

Operations on matrices can be divided into two major groups. The first group comprises termwise operations such as scalar multiplication, addition, etc. The second group is formed by matrix multiplication, matrix transpose and matrix inversion. Termwise operations are easily implemented based on function graphs. We can simply use *apply* to compute all termwise operations on matrices. This is obviously possible since *apply*(*op*) performs the operation *op* on every single function value, i.e. it works in a termwise manner. Matrix specific operations such as transposition require their own tailored algorithms.

Matrix multiplication is clearly a non-termwise operation since it requires computing the scalar vector product of a row of the left matrix with a column of the right matrix to get the value of a single matrix element of the product matrix. Therefore, we will present two different recursive procedures to perform matrix multiplication on function graphs. The first method was proposed by McGeer [9]. This algorithm has the most direct link to the common conventional method for matrix multiplication. In every recursion step the problem is divided into four subproblems until a terminal case has been reached. In these steps operands are expanded with regard to a pair of row and column designators. This expansion even takes place if the function graphs are actually not dependent on the current pair of internal variables. By doing so there is no need for a scaling step as is necessary in the second method. Let matrix multiplication be denoted by $\star$ and matrix addition by $+$. This method can be formally stated as:

$$h(\{x_0, y_0, \ldots, x_{m-1}, y_{n-1}\}) = f(\{x_0, z_0, \ldots, x_{m-1}, z_{p-1}\}) \star g(\{z_0, y_0, \ldots, z_{p-1}, y_{n-1}\})$$

or written in terms of matrices:

$$\begin{bmatrix} h_{\overline{x}\overline{y}} & h_{\overline{x}y} \\ h_{x\overline{y}} & h_{xy} \end{bmatrix} = \begin{bmatrix} f_{\overline{x}\overline{z}} & f_{\overline{x}z} \\ f_{x\overline{z}} & f_{xz} \end{bmatrix} \star \begin{bmatrix} g_{\overline{z}\overline{y}} & g_{\overline{z}y} \\ g_{z\overline{y}} & g_{zy} \end{bmatrix}$$

The computations performed in every recursion step are:

$$\begin{aligned} h_{\overline{x}\overline{y}} &= f_{\overline{x}\overline{z}} \star g_{\overline{z}\overline{y}} + f_{\overline{x}z} \star g_{z\overline{y}} \\ h_{\overline{x}y} &= f_{\overline{x}\overline{z}} \star g_{\overline{z}y} + f_{\overline{x}z} \star g_{zy} \\ h_{x\overline{y}} &= f_{x\overline{z}} \star g_{\overline{z}\overline{y}} + f_{xz} \star g_{z\overline{y}} \\ h_{xy} &= f_{x\overline{z}} \star g_{\overline{z}y} + f_{xz} \star g_{zy} \end{aligned}$$

Obviously, this method requires eight calls to *matrix multiply* and four calls to *matrix add* in every recursion step, i.e. for every internal variable pair.

The second method was proposed by Bahar [2]. Unlike the previous method it only expands the top variable of the two operands $f\{x_0, z_0, \ldots, x_{n-1}, z_{n-1}\}$ and $g\{z_0, y_0, \ldots, z_{n-1}, y_{n-1}\}$. In the process of matrix multiplication, the following variable order $\{x_0, z_0, y_0, \ldots, x_{n-1}, z_{n-1}, y_{n-1}\}$ is imposed to decide whether the top variable of f or g has to be selected as the top variable for expansion. Depending on the character of the expansion variable var one of the following computations is being made in every recursion step.

- if($var = z_i$) then

$$f(x,z) \star g(z,y) = (f_v(x,z) \star g_v(z,y)) + (f_{\overline{v}}(x,z) \star g_{\overline{v}}(z,y))$$

- if($var = x_i$ or $var = y_i$) then

$$f(x,z) \star g(z,y) = v \wedge (f_v(x,z) \star g_v(z,y)) \vee \overline{v} \wedge (f_{\overline{v}}(x,z) \star g_{\overline{v}}(z,y))$$

This approach only expands internal variables that are actually encountered in the function graphs f and g. It requires to keep track of missing z variables in f and g since every z expansion step corresponds to performing matrix addition. If p gives the number of omitted z expansions between two recursion steps we have to scale the result by 2^p before returning it. When using a cache we always store the unscaled results and scale the entry accordingly when reading the cache.

Another method was proposed by Clarke [9]. Its basic idea is to take all the products first and then compute all the sums.

For our matrix package we have implemented the second method which appears to be superior to the other two [2]. We implemented two different versions of this method. Version 1 passes the value of the edge weights down with every recursion step of *matrix multiply* and is of $O(\|f\| \cdot \|g\|)$ complexity. As we have done for multiplication of two FEVBDDs we suggest a second version for function graphs with a high ratio of reconverging branches (e.g. for recursively-affine matrices) as follows.

$$
\begin{aligned}
\langle c_h, w_h, \mathbf{h}, rule_h\rangle &= \langle c_f, w_f, \mathbf{f}, rule_f\rangle \star \langle c_g, w_g, \mathbf{g}, rule_g\rangle \\
&= ([c_f]_{n\times n} + w_f \cdot [f]_{n\times n}) \star ([c_h]_{n\times n} + w_h \cdot [h]_{n\times n}) \\
&= [2^n \cdot c_f \cdot c_g]_{n\times n} + [c_f \cdot w_g]_{n\times n} \star [h]_{n\times n} + \\
&\quad [f]_{n\times n} \star [c_g \cdot w_f]_{n\times n} + [f]_{n\times n} \star [g]_{n\times n} \\
&= [2^n \cdot c_f \cdot c_g]_{n\times n} + c_f \cdot w_g \cdot coladd([h]_{n\times n}) + \\
&\quad c_g \cdot w_f \cdot rowadd([f]_{n\times n}) + [f]_{n\times n} \star [g]_{n\times n}
\end{aligned}
$$

The operations $rowadd$ and $coladd$ which generate matrices such that

$$
rowadd(\begin{bmatrix} a_{00} & \dots & a_{0n} \\ \vdots & & \vdots \\ a_{n0} & \dots & a_{nn} \end{bmatrix}) = \begin{bmatrix} \sum_i a_{0i} \\ \vdots \\ \sum_i a_{ni} \end{bmatrix}
$$

$$
coladd(\begin{bmatrix} a_{00} & \dots & a_{0n} \\ \vdots & & \vdots \\ a_{n0} & \dots & a_{nn} \end{bmatrix}) = \begin{bmatrix} \sum_i a_{i0} & \dots & \sum_i a_{in} \end{bmatrix}
$$

only have complexity $O(|f|)$. This second version requires only $O(|f| \cdot |g|)$ calls to *matrix multiply* but every recursive call to *matrix multiply* also requires three calls to *matrix add*. It improves the cache efficiency of matrix multiplication considerably, if both operands are represented by FEVBDDs with a high ratio of reconverging branches. This outweighs the added overhead of three calls to *matrix add*. If this is not the case, it is better to use the original approach since it does not require the additional overhead.

Matrix transposition is performed by exchanging the roles of column and row designators belonging to the same expansion level. To maintain the imposed variable ordering the nodes in the function graph have to be exchanged and it is not sufficient to just interpret row as column designators and vice versa. Transposition can be done in $O(|f|)$ time.

Matrix inversion is done by performing Gaussian elimination on the original matrix and the identity matrix at the same time. In other words we solve the system of linear equations $A \star X = 1$ with the use of pivoting and row transformations. The steps required by Gaussian elimination consist of [19]:

- selecting a partial pivot in every step j such that $|a_{pj}| = max_{i \geq j} |a_{ij}|$
- normalizing the selected row by multiplying the row by the inverse of the pivot $\frac{1}{|a_{pj}|}$
- swapping rows j and p according to above pivot selection
- subtracting multiples of the pivot row j from all rows $i > j$ such that $a_{ji} = 0, \forall i > j$

All of the above operations except for row swapping can be implemented efficiently in time $O(|A|)$ or $O(|A| \cdot |R|)$ where R denotes the FEVBDD representing the pivot row. Row swapping is performed by matrix multiplication of matrix A with a permutation matrix P and therefore is of complexity $O(\|A\| \cdot \|P\|)$. Permutation matrices can be obtained by $P_{ij} = I \oplus M_{ij}$ where P_{ij} denotes a permutation matrix swapping rows i and j, I represents the identity matrix and M_{ij} designates a matrix

$$M = \begin{cases} m_{rs} = 1 & \text{if } r = i, j \text{ or } s = i, j \\ m_{rs} = 0 & \text{otherwise} \end{cases}$$

In general, partial pivoting is done in order to improve the numerical accuracy of Gaussian elimination. Since our implementation relies on fractions of arbitrary precision we always use the exact values and numerical stability is not an issue. In order to avoid unnecessary row swapping we only perform the partial pivoting if it holds in step j that $a_{jj} = 0$.

In addition to the basic matrix operations, fast search operations for specific matrix elements have been implemented. Algorithms for searching both the value and position of the minimal, maximal or absolute maximal element in a given matrix were developed. This approach makes use of the *min* and *max* fields that can be associated with every node. The computational complexity for finding both the value and position of the minimal or maximal element in a $n \times n$ matrix is $O(\log_2(\lceil n \rceil))$. We will now explain the basic idea behind the algorithms in the case of searching for the maximal element. Given a FEVBDD node **f** and its two successors $\mathbf{f_t}$ and $\mathbf{f_e}$ we can easily determine which edge leads to the maximal element. Based on the values of the max and min fields of $\mathbf{f_t}$ and $\mathbf{f_e}$ we simply recompute the max field of **f** and select the successor that originally generated the max field of **f**. If only the value of the maximal or minimal element is of interest, it can be computed directly from the min and max field of the top node **f** without any further computation.

value range	EVBDD	FEVBDD	
		GCD	RATIONAL
integer	12 bytes	20 bytes	24 bytes
fractions	16 bytes	NA	24 bytes

Table 5: Memory requirement per node

4.3 Experimental Results

We have applied our FEVBDD based matrix package to the problem of solving the Chapman-Kolmogorov equations [18] that arise when computing the global state probabilities of FSMs. Though the memory consumption of our inversion routine is relatively low (8M for inverting a 64x64 matrix), the run time is very high. This is due to several factors. First, the algorithm for Gaussian elimination is purely sequential whereas FEVBDDs are recursively defined. Consequently, computation caching for matrix inversion does not exist. A recursive algorithm for matrix inversion will perform much better on FEVBDDs. Secondly, when using fractions of arbitrary length all arithmetic operations need substantially more time than is necessary for ordinary integers. We therefore use the obtained inverses primarily as examples of real life non-sparse matrices that can be represented compactly using FEVBDDs and compare them with their EVBDD representations. As can be seen from the table below using FEVBDDs gives savings of up to 50% compared to EVBDDs in the number of nodes required to represent the non-sparse inverse. Of course, one has to consider that the storage requirement per node is higher for FEVBDDs than for EVBDDs. An overview of the memory usage per node in the various modes available for EVBDDs and FEVBDDs is given in table 5. We assume that every EVBDD node consists of an integer or fractional edge value and two pointers to the children. Every FEVBDD node consists of two fractional edge weights and two pointers in the RATIONAL mode or three integer edge weights and two pointers in the GCD mode.

The total memory consumption for storing the matrices using EVBDDs and FEVBDDs is shown in tables 6 and 7, respectively. The given memory usage is based on EVBDDs and FEVBDDs using fractions. The FEVBDDs have been generated using the RATIONAL rule. In the case of CK-Equations we have to use fractions for the edge weights since the matrix elements are fractions. As can be seen from the tables, FEVBDDs do better for the inverses but lose for the original matrices in terms of total memory consumption. This is due to the fact that the original matrices are sparse whereas the inverses are non-sparse. In the case of sparse matrices the additional properties of

FEVBDDs are not exploited so that EVBDDs and FEVBDDs perform similarly in the number of nodes. FEVBDDs, however, lose in terms of memory requirement because of the higher cost per FEVBDD node. Since EVBDDs do at least as good as MTBDDs this also gives an idea of the performance of FEVBDDs compared to MTBDDs.

5 Conclusion

We showed that by associating both an additive and a multiplicative weight with the edges of an Edge-Valued Binary Decision Diagram, EVBDDs could successfully be extended to Factored Edge-Valued Binary Decision Diagrams. The new data structure preserves the canonical property of the EVBDD and allows efficient caching of operational results. All properties that have been defined for EVBDDs could be adapted to FEVBDDs. The *additive property* was extended to the *affine property*. It was shown that FEVBDDs provide a more compact representation of arithmetic functions than EVBDDs. Additionally, the complexity of certain operations could be reduced significantly. We showed that FEVBDDs representing boolean functions allow us to incorporate the concept of complement edges that has originally been proposed for OBDDs. Furthermore, we showed that the EVBDD based Integer Linear Programming solver FGILP benefits from using FEVBDDs instead of EVBDDs.

In combination with the FEVBDD package we also implemented an arithmetic package which supplies arithmetic operations on both integers and fractions of arbitrary precision. A complete matrix package based on FEVBDDs was introduced. We applied the package to solving the Chapman-Kolmogorov equations. The experimental results show that in the majority of cases FEVBDDs win over the corresponding EVBDD representation of the matrices in terms of number of nodes and memory consumption.

Acknowledgement

The authors like to thank Y.-T. Lai for supplying them with the EVBDD package and many helpful discussions.

References

[1] S. B. Akers, “Binary decision diagrams,” *IEEE Transactions on Computers*, vol. C–27, pp. 509–516, June 1978.

CK-Equations	Size	EVBDD				FEVBDD			
		# of Nodes (orig.)	memory (byte)	# of Nodes (inv.)	memory (byte)	# of Nodes (orig.)	memory (byte)	# of Nodes (inv.)	memory (byte)
bbara	10	61	976	92	1472	49	1176	57	1368
bbtas	6	29	464	28	448	20	480	18	432
beecount	7	33	528	43	688	22	528	25	600
cse	16	86	1376	153	2448	63	1512	85	2040
dk14	7	31	496	43	688	24	576	27	648
dk15	4	12	192	12	192	8	192	7	168
dk16	27	150	2400	494	7904	130	3120	314	7536
dk17	8	33	528	41	656	26	624	26	624
dk27	7	24	384	33	528	19	456	24	576
dk512	15	56	896	124	1984	50	1200	85	2040
donfile	24	93	1488	139	2224	81	1944	120	2880
ex1	20	112	1792	279	4464	83	1992	172	4128
ex2	19	113	1808	251	4016	97	2328	156	3744
ex3	10	61	976	119	1904	49	1176	68	1632
ex4	14	48	768	56	896	36	864	42	1008
ex5	9	54	864	93	1488	44	1056	57	1368
ex6	8	40	640	58	928	28	672	31	744
ex7	10	65	1040	105	1680	57	1368	62	1488
keyb	19	133	2128	369	5904	93	2232	212	5088
kirkman	16	25	400	25	400	24	576	22	528
lion	4	12	192	10	160	8	192	7	168
lion9	9	38	608	55	880	31	744	45	1080
mark1	15	57	912	8	128	48	1152	8	192

Table 6: Experimental results

CK-Equations	Size	EVBDD # of Nodes (orig.)	EVBDD memory (byte)	EVBDD # of Nodes (inv.)	EVBDD memory (byte)	FEVBDD # of Nodes (orig.)	FEVBDD memory (byte)	FEVBDD # of Nodes (inv.)	FEVBDD memory (byte)
mc	4	10	160	9	144	8	192	6	144
modulo12	12	31	496	32	512	29	696	27	648
opus	10	56	896	70	1120	47	1128	54	1296
planet	48	209	3344	488	7808	157	3768	366	8784
planet1	48	209	3344	488	7808	157	3768	366	8784
pma	24	132	2112	296	4736	102	2448	182	4368
s1	20	149	2382	352	5632	117	2808	201	4824
s1a	20	149	2382	352	5632	117	2808	201	4824
s1494	48	242	3872	1525	24400	189	4536	841	20184
s208	18	62	992	217	3472	55	1320	98	2354
s27	6	29	464	35	560	22	528	21	504
s386	13	68	1088	146	2336	52	1248	83	1992
s8	5	28	448	28	448	20	480	20	480
sand	32	119	1904	193	3088	86	2064	135	3240
shiftreg	8	26	416	30	480	21	504	22	528
styr	30	170	2720	370	5920	116	2784	226	5424
tav	4	9	144	9	144	8	192	6	144
tbk	32	73	1168	81	1296	62	1488	61	1464
tma	20	114	1824	249	3984	88	2112	155	3720
train11	11	52	832	71	1136	43	1032	57	1368
train4	4	9	144	9	144	8	192	6	144

Table 7: Experimental results (cont.)

[2] R. I. Bahar, E. A. Fromm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic Decision Diagrams and their Applications" *International Conference on Computer-Aided Design*, pp. 188–191, November 1993.

[3] B. Becker, R. Drechsler, and R. Werchner, "On the relation between BDDs and FDDs", Technical Report 12/93, University of Frankfurt, 1993.

[4] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," *Proceedings. of the 27th Design Automation Conference*, pp. 40–45, 1990.

[5] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C–35(8), pp. 677–691, August 1986.

[6] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Surveys*, vol. 24, No. 3, pp. 293–318, September 1992.

[7] R. E. Bryant, and Yirng-An Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams," *CMU-CS-94-160*, May 1994.

[8] E. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. C.-Y. Yang, "Spectral transforms for large Boolean functions with application to technology mapping", *30th ACM/IEEE Design Automation Conference*, Dallas, TX, pp. 54–60 June 1993.

[9] E. Clarke, M. Fujita, P. C. McGeer, K. L. McMillan, and J. C.-Y. Yang, "Multi-terminal binary decision diagrams: an efficient data structure for matrix representation," unpublished, 1993.

[10] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski, "Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decisiond Diagrams", .

[11] D. E. Knuth, "The Art of Computer Programming Volume 2: Seminumerical Algorithms", 2nd edition, Addison Wesley, 1981.

[12] Y.-T. Lai, and S. Sastry, "Edge-valued binary decision diagrams for multi-level hierarchical verification", *29th Design Automation Conference,* pp. 608–613, June 1992.

[13] Y-T. Lai, M. Pedram and S. B. K. Vrudhula, "EVBDD-based algorithms for integer linear programming, spectral transformation and function decomposition", *IEEE Trans. on Computer-Aided Design,* Vol. 13, No. 8, pages 959-975, 1994.

[14] Y-T. Lai, M. Pedram and S. B. K. Vrudhula "Edge-valued binary decision diagrams", Submitted to *IEEE Trans. on Computers*, 1993.

[15] Y.-T. Lai, M. Pedram and S. B. K. Vrudhula, "FGILP: An integer linear program solver based on function graphs", *International Conference on Computer-Aided Design*, pp. 685–689, November 1993.

[16] C. Y. Lee, "Representation of switching circuits by binary-decision-programs", *Bell. Syst. Tech. J.*, vol. 38, pp. 985–999, July 1959.

[17] D. F. Elliott, K. R. Rao, "Fast Transforms. Algorithms, Analyses, Applications" Academic Press, 1982

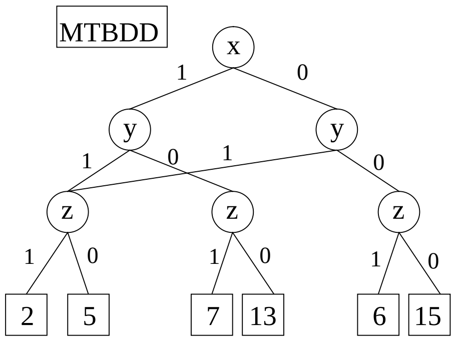
[18] S. Ross, "A First Course in Probability", Macmillan, 1988.

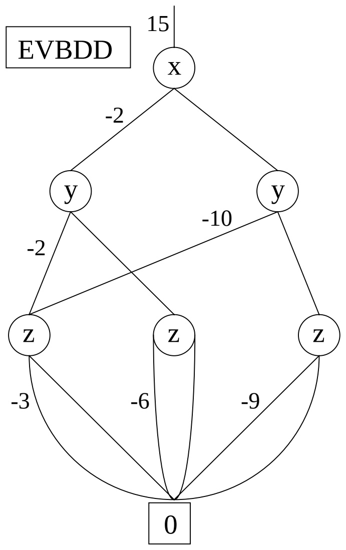
[19] J. Stoer, and R. Bulirsch, "Introduction to Numerical Analysis", Springer-Verlag, 1980.

[20] P. Tafertshofer, and M. Pedram, "Factored Edge-Valued Binary Decision Diagrams and their Application to Matrix Representation and Manipulation", Technical Report CENG 94-27, Department of Electrical Engineering-Systems, University of Southern California, October 1994.

Tabular Form

x	y	z	f
0	0	0	15
0	0	1	6
0	1	0	5
0	1	1	2
1	0	0	13
1	0	1	7
1	1	0	5
1	1	1	2





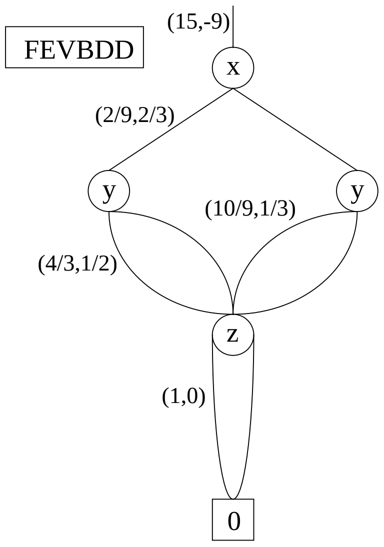


Figure 1: Example for function decompositions and function graphs

Figure 2: Representations of signed integers using FEVBDDs

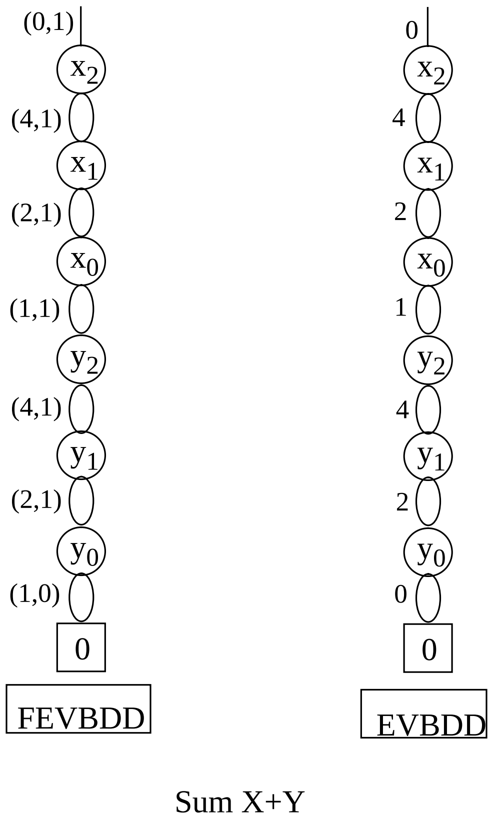


Figure 3: Representations of word-level sum

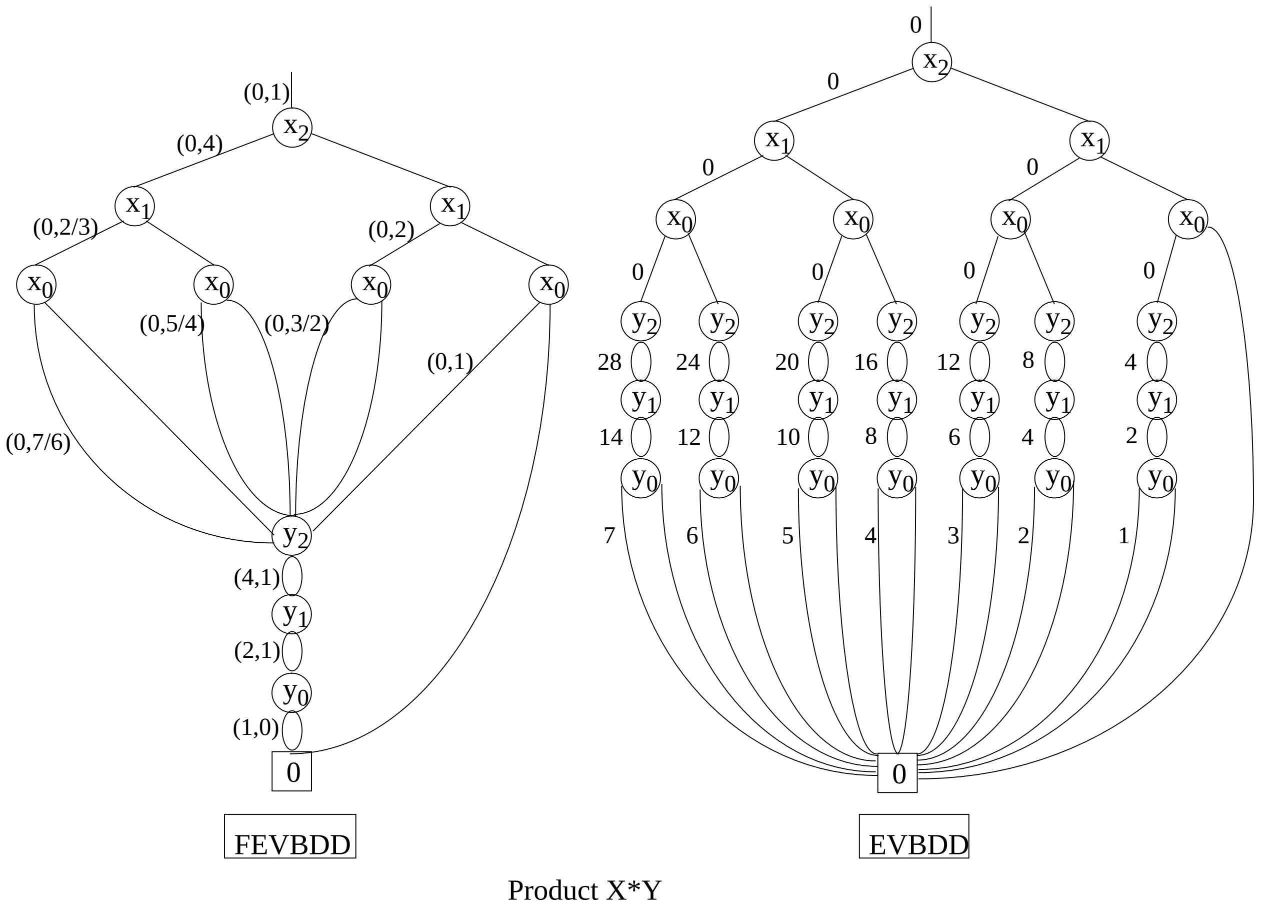


Figure 4: Representations of word-level product

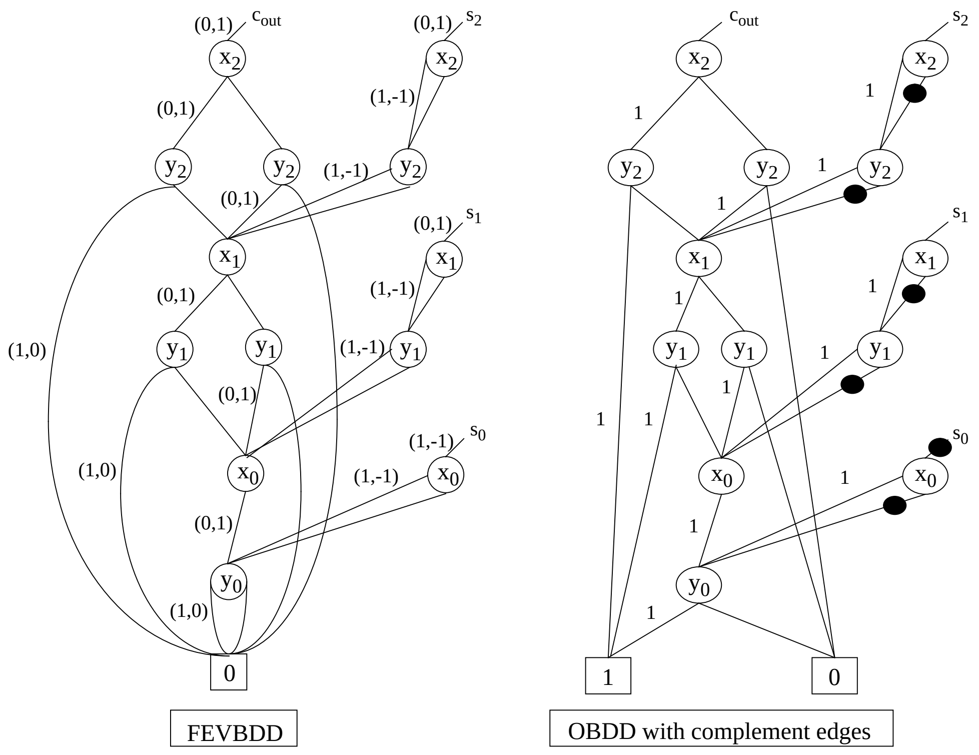


Figure 5: FEVBDD representation of the four output functions of a 3-bit adder

Figure 6: Correspondence of submatrices and subtrees

Figure 7: Recursively-affine matrices

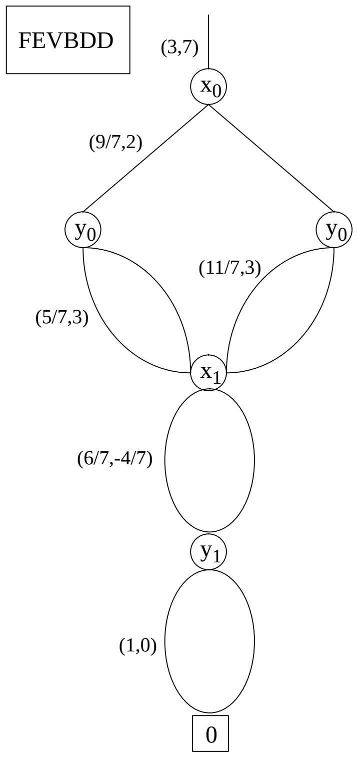


Figure 8: The FEVBDD for a recursively-affine matrix

Figure 9: FEVBDD and EVBDD representations of the Walsh matrix H_3^h