Edge-Shifted Decision Diagrams for Multiple-Valued Logic

Benet Devereux   Marsha Chechik
Department of Computer Science
University of Toronto

October 29, 2001

Abstract

Symbolic data structures for multi-valued logics are useful in a number of applications, from model-checking to circuit design and switch-level circuit verification. Such data structures are referred to as *decision diagrams*, and are typically considered effective if they are small, i.e., common co-factors of a function are shared, and canonical, i.e., given a variable ordering, there is a unique decision diagram representing this function.

In this paper we propose and evaluate a decision diagram variety, called *Edge-Shifted Decision Diagrams* (ESDDs). These diagrams are multi-valued, support structure sharing, and enjoy canonicity. We further list sufficient conditions to ensure canonicity of decision diagrams with unary edge transformers.

1 Introduction

Symbolic data structures for multiple-valued logics are useful in a number of applications, from model-checking to circuit design and switch-level circuit verification.

In particular, multiple-valued logics can explicitly represent uncertainty and disagreement. Thus they can be applied to the modeling of software behavior, especially the exploratory modeling used in the early stage of requirements engineering and architectural design, to verify properties of models that contain uncertainty or disagreement. This can be done fully automatically, using *model-checking* [7]. Symbolic model-checking over a class of *quasi-boolean* multiple-valued logics has been studied in [6, 5]. Multiple-valued logic also has hardware design applications, pioneered by K.C.Smith [15] and recently surveyed in [8]. When modeling circuits with many inputs and outputs, the inputs and outputs can be grouped to yield multiple-valued, "word-level" functions. This grouping is used to simplify representation, simulation, and verification of circuits, even though the actual implementation is still two-valued. The tool VIS [19] is an example of this application. Circuits can also be implemented using multiple-valued digital logic; such an implementation can help reduce the number of connections of an integrated circuit. Finally, multiple-valued logic is useful for switch-level verification [9]. Switch-level modeling is one level of abstraction below gate-level modeling: interconnections between transistors are explicitly modeled, and signal levels other than on or off are possible. We formally define multiple-valued logics in Section 2.

Both circuits and state-based system specifications are usually modeled as functions from inputs to outputs and represented by decision diagrams [3]. A decision diagram is a rooted, directed acyclic graph, whose non-terminal nodes are labelled with input variables, and whose terminal nodes are labelled with possible output values. The function is evaluated by an appropriate traversal of the graph. The two highly desirable properties of decision diagrams are *sharing* – common co-factors of a function are shared, leading to a smaller size of the resulting diagram; and *canonicity* – representation of identical functions by the same structure. The later is essential for efficient check of equivalence between circuits and quick computation of fixpoints in symbolic model-checking. There is an extensive literature on the subject of devising efficient symbolic data structures for various applications, typically assuming that variables are boolean [4].

One can view decision diagrams over boolean variables as follows: each decision node is labelled with a variable x, and has two outgoing edges, leading to functions f_0 and f_1 that do not depend on x. The node itself combines these two functions depending on the value of x: if $x = 0$ then f_0 else f_1. In this paper we are interested in efficient multiple-valued decision diagrams, MDDs. In MDDs, a node in a decision diagram is also labelled with a variable but has several outgoing edges, one for each value of the logic. Each edge leads to a function that does not depend on the current variable, and the functions are combined to yield a function that depends on the current variable. We define multiple-valued decision diagrams formally in Section 3.

In order to achieve greater structure sharing and thus reduce the size of the symbolic data structure, some optimizations of decision diagrams enable a single subgraph to represent more than one function, based on the context it appears in. In ordinary BDDs, an edge may only terminate in a single subgraph, representing a function. It is also possible to "label" an edge with a unary function, e.g., an indicator that the destination function is complemented. Thus, if the subgraph reached by the complemented edge represents f, then the entire function is $\neg f$. Even further, the function represented by an outgoing edge may be binary, i.e, a *combination* of the functions represented by two or more subgraphs: the edge may split and lead to subgraphs representing f and g, and yield the function $f \wedge g$. We survey several approaches to edge annotations in Section 4: complement edges for BDDs [16], edge-valued decision diagrams for boolean-input numeric-output functions [12], and mod-p decision diagrams [13] for multiple-valued logic functions. The latter enjoy a compact representation of multiple-valued functions at the expense of canonicity.

The principal contribution of this paper is the definition of a new multiple-valued decision-diagram variety that is space-efficient while remaining canonical. We refer to these as *Edge-Shifted Decision Diagrams* (ESDDs). These diagrams are discussed in Section 5. We also provide a formal proof of canonicity and experimental comparisons between our implementation and that of standard MDD representations.

In Section 6 we give general conditions guaranteeing canonicity for decision diagrams for finite multiple-valued output functions. Given a proposed set of unary edge transformations for decision diagrams, we give sufficient conditions to ensure that this set preserves canonicity.

Finally, we conclude the paper in Section 7 with a summary of contributions and an outline of future work.

2 Multiple-Valued Logics

In this section we define a large family of multiple-valued logics. Our definition is general enough to include logics used for symbolic model-checking and for circuit design and verification.

Definition 1 *A multiple-valued logic is a tuple* $L = (R, \wedge, \mathbf{1}, \vee, \mathbf{0}, \{\mathbf{eq}_r\}_{r \in R})$ *where:*

- R *is a finite set of truth values* $\{r_1, \ldots, r_n\}$*, where* $r_1 = \mathbf{0}$ *and* $r_n = \mathbf{1}$*;*
- $\wedge$ *is an associative, commutative binary operation on* R*, with identity* $\mathbf{1}$*;*
- $\vee$ *is an associative, commutative binary operation on* R*, with identity* $\mathbf{0}$*;*
- $x \wedge \mathbf{0} = \mathbf{0}$ *and* $x \vee \mathbf{1} = \mathbf{1}$ *(base laws);*
- $\{\mathbf{eq}_r\}_{r \in R}$ *is a family of unary operations on* R*, one for each value:*

$$\mathbf{eq}_r(x) \triangleq \begin{cases} \mathbf{1} & \textit{if } x = r \\ \mathbf{0} & \textit{otherwise} \end{cases} \qquad \text{(definition of } \mathbf{eq}\text{)}$$

Precedence of the operators is such that $\mathbf{eq}_r$ binds more strongly than $\wedge$, which in turn binds more strongly than $\vee$.

For example, ordinary boolean logic is defined by the tuple $(\{0, 1\}, \wedge, 1, \vee, 0, \{\mathbf{eq}_0, \mathbf{eq}_1\})$. In general, for some propositional variable x, we write $\mathbf{eq}_1(x)$ simply as x, and $\mathbf{eq}_0(x)$ as $\neg x$. For example, the exclusive-or function can be written in full as $x \oplus y = (\mathbf{eq}_1(x)) \wedge (\mathbf{eq}_0(y)) \vee (\mathbf{eq}_0(x) \wedge \mathbf{eq}_1(y))$ or, in the more familiar notation, as $x \wedge \neg y \vee \neg x \wedge y$.

We are frequently interested in functions whose *output* values come from a multiple-valued logic L, but whose inputs are from some other finite domain D. In this case, we also define the set $\{\mathbf{eq}_d\}_{d \in D}$ where $\mathbf{eq}_d : D \to R$ is defined as above. We let $m = |D|$.

Definition 2 *For any multiple-valued logic* L *and input domain* D*,* **case** $: D \times R^m \to R$ *is defined as follows:*

$$\textbf{case}(x, y_1, \ldots, y_m) \triangleq \mathbf{eq}_{d_1}(x) \wedge y_1 \vee \cdots \vee \mathbf{eq}_{d_m}(x) \wedge y_m \qquad \text{(definition of } \textbf{case}\text{)}$$

For all $d_i \in D$*, this expression evaluates to* y_i *if and only if* $x = d_i$*.*

The theorem below states two properties of **case** without proof. The property of **case** idempotence states that if all of the cases in a **case** are the same, then it does not depend on the input and can be eliminated; **case** equality states that if two **case** statements with the same input are identical, then all of the cases are identical.

Theorem 1 **case** *has the following properties:*

$$\mathbf{case}(x, y, \ldots, y) = y \qquad (\mathbf{case}\text{ idempotence})$$
$$(\mathbf{case}(x, y_1, \ldots, y_m) = \mathbf{case}(x, y'_1, \ldots, y'_m)) \Rightarrow (\forall i \cdot y_i = y'_i) \qquad (\mathbf{case}\text{ equality})$$

In the classical case, standard boolean connectives $\wedge$, $\vee$, and $\neg$ form an adequate set: all functions can be expressed using them. However, this set is insufficient for multiple-valued logics, and the addition of the constant equality operator $\mathbf{eq}$ is necessary. Consider, for example, a logic defined over the truth values T, M, F, representing *True*, *Maybe* and *False*, respectively [11]. Conjunction and disjunction are defined via the following truth tables:

$\wedge$	F	M	T
F	F	F	F
M	F	M	M
T	F	M	T

$\vee$	F	M	T
F	F	M	T
M	M	M	T
T	T	T	T

Note that T is the identity for $\wedge$, and F the identity for $\vee$. Negation is defined as follows: ¬T = F, ¬M = M, ¬F = T. We call this function $\mathcal{L}_3$. Note that a function f s.t. $f(\mathrm{T}) = \mathrm{T}$, $f(\mathrm{M}) = \mathrm{T}$ and $f(\mathrm{F}) = \mathrm{F}$ cannot be expressed using only $\wedge, \vee$, and $\neg$; however, it can be expressed as $f(x) = \mathbf{eq}_{\mathrm{M}}(x) \vee \mathbf{eq}_{\mathrm{T}}(x)$.

We conclude this section by listing some notational conventions used in the remainder of this paper. Throughout the paper we shall use $(\circ)$ to denote composition of functions. In addition, we present proofs in the calculational style [10, 2]. In the diagrams that follow, either edges are annotated, or low edges are indicated by dashed lines.

3 Multiple-Valued Decision Diagrams

In this section we present multiple-valued decision diagrams or MDDs [17], a graph-based representation for functions on a finite domain. When restricted to boolean variables and boolean output, these are simply binary decision diagrams [16].

3.1 Function Decomposition

We consider a variable domain $D = \{d_1, \ldots, d_m\}$ of finite size m, and a range $R = \{r_1, \ldots, r_{n-1}\}$ of size n, where $L = (R, \wedge, \mathbf{1}, \vee, \mathbf{0}, \{\mathbf{eq}_r\}_{r \in R})$ is a multiple-valued logic.

Functions of type $D^k \to R$ (for any k) can be represented by *reduced* and *ordered* decision diagrams. The principal benefits of this representation are *compactness*, since redundancy in

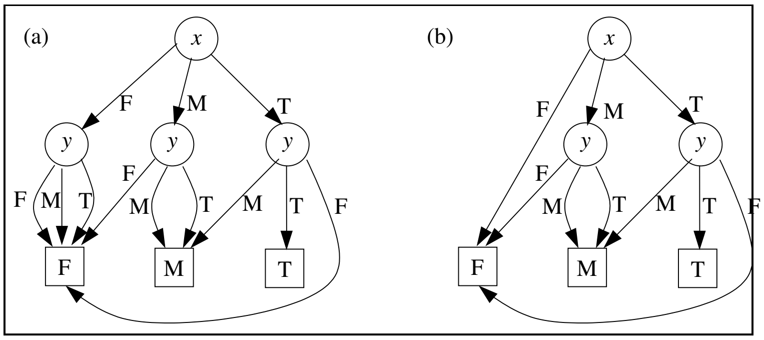

Figure 1: Representing a function $f = y \wedge x$: (a) using a decision tree; (b) using a reduced MDD.

the function is leveraged to reduce the size of the data structure, and *canonicity*: each function is represented by a unique decision diagram, and thus checking for equality can be done in constant time. We write such functions in the form $f(x_1, \ldots, x_k)$, where each x_i is an input variable of type D; this yields a total ordering on the input variables.

The finiteness of D allows the hierarchical decomposition of a function in k variables into m functions of $(k-1)$ variables, called *cofactors*; these can, in turn, themselves be recursively decomposed, until 0-variable functions – constants – are reached. This process yields a *decision tree* representation of the function, and by the removal of redundant comparisons and merging of identical subgraphs, the decision tree can be transformed into a reduced decision diagram containing fewer nodes than the full decision tree.

For example, consider representing a function $y \wedge x$ in the three-valued logic $\mathcal{L}_3$, defined in Section 2. The ordered decision tree representing this function is given in Figure 1(a). For example, if x is M and y is T, the function evaluates to M. We note that the left-most y node is redundant: the function evaluates to F regardless of the value of y. Thus, this node is not part of the reduced MDD, shown in Figure 1(b). No further reductions were possible.

We now give a formal definition of cofactors and their composition.

Definition 3 *Let $f : D^k \to R = f(x_1, \ldots, x_k)$ be a function, and $D = \{d_1, \ldots, d_m\}$. For any i, j with $0 \leq i < k, 0 \leq j < |D|$, the j-th* cofactor of f with respect to x_i *is:*

$$f|_{x_i = d_j} \triangleq f(x_1, \ldots, x_{i-1}, d_j, x_{i+1}, \ldots, x_k) \quad \text{(definition of cofactor)}$$

So any function f can be represented as a **case** over its cofactors:

$$f = \textbf{case}(x_i, f|_{x_i=d_1}, \ldots, f|_{x_i=d_m}) \quad \text{(generalized Shannon expansion)}$$

Frequently, particularly in hardware applications, decision diagrams are used to represent functions with multiple outputs. A multiple-output function $f : D^k \to R^j$ is equivalent to a vector of functions $(f_1 : D^k \to R, \ldots, f_j : D^k \to R)$: each of these functions can be represented as a decision diagram, and these diagrams share subgraphs. That is, if any cofactor appears in the decomposition of two distinct functions, it will be represented by the *same* subgraph, not just an isomorphic one. In implementation terms, this amounts to using the same unique table of diagram nodes for each function, rather than creating several. Since sharing actually simplifies the implementation, its use in decision-diagram packages is standard, and thus is rarely discussed explicitly. We use shared diagrams in our experiments in Section 5.5, where our benchmarks are multiple-output adders and multipliers.

3.2 Formal Definition

A decision diagram is a directed acyclic graph augmented with some additional information, defined below.

Definition 4 *A D-input R-output decision diagram, on an ordered set of variables $A = \{a_1, \ldots, a_k\}$ is a rooted directed acyclic graph $(V \cup T, E)$ where:*

- *Each $u \in V$ is labeled with a variable from A; we denote this variable by* $\mathbf{label}(u)$.
- *Each $t \in T$ is labeled with a value from R; we denote this value by* $\mathbf{value}(t)$.
- *For each node $u \in V$, there is a function* $\mathbf{child}_u : D \to (V \cup T)$ *which assigns, for every possible value of the variable u, a child node corresponding to that input value. An edge (u, u') is in E if and only if there is some $d \in D$ for which* $\mathbf{child}_u(d) = u'$.

Definition 5 *A D-input R-output decision diagram $(V \cup T, E)$ is* reduced *and* ordered *if:*

$$
\begin{array}{ll}
\forall d \in D \cdot \forall u \in V \cdot (\mathbf{child}_u(d) \in T) \vee (\mathbf{label}(v) < \mathbf{label}(\mathbf{child}_u(d))) & \text{(orderedness)} \\
\forall u \in V \cdot \exists d, d' \in D \cdot \mathbf{child}_u(d) \neq \mathbf{child}_u(d') & \text{(reducedness 1)} \\
(\mathbf{label}(u') = \mathbf{label}(u) \wedge (\forall d \in D \cdot \mathbf{child}_u(d) = \mathbf{child}_{u'}(d))) \Rightarrow (u = u') & \text{(reducedness 2)}
\end{array}
$$

For each node $u \in V$, the outgoing edge $(u, \mathbf{child}_u(\mathbf{0}))$ is called the *low edge*, and $(u, \mathbf{child}_u(\mathbf{1}))$ the *high edge*.

For any node u and a function f, let function f_u represented by the node be defined as follows:

$$
f_u \triangleq \begin{cases} \mathbf{case}(\mathbf{label}(u), f_{\mathbf{child}_u(d_1)}, \ldots, f_{\mathbf{child}_u(d_m)}) & \text{if } u \in V \\ \mathbf{value}(u) & \text{if } u \in T \end{cases}
$$

The absence of cycles in the decision diagram guarantees that the function can be evaluated in at most k steps.

Theorem 2 *[17] For any finite D and R, reduced and ordered DDs are* canonical*: that is, if there are two nodes $u, u' \in (V \cup T)$ such that $f_u = f_{u'}$, then $u = u'$.*

That is, reducedness and orderedness of decision diagrams is a sufficient condition for their canonicity.

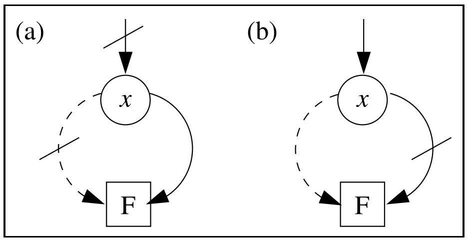

Figure 2: Two possible representations for $f(x) = x$ as a complement-edged BDD: (a) a non-canonical representation; (b) a canonical representation.

4 Decision Diagrams With Edge-Valuations

In this section, we discuss the annotation of decision diagram edges. Edges of the diagram can be annotated with *edge transformers*, which alter the function computed by their destination nodes. Thus, a single diagram with an incoming edge can actually represent more than one function, depending on the transformer attached to the incoming edge. This allows greater sharing of common subexpressions and consequently smaller diagrams. On the other hand, size reduction is achieved at the cost of storing the edge transformers and complexity of manipulation.

In this section we describe three different kinds of edge transformation, of increasing complexity. The first two cases, *complement edges* for BDDs [16] and *addition edges* for edge-valued decision diagrams (EVDDs) [12], rely entirely on unary functions, and are equipped with conditions that make them canonical. The *mod-p decision diagrams* of Sack et al. [13] use edge transformations which are of a higher arity but lose canonicity.

4.1 Complement Edges

Ordinary BDDs are commonly optimized with *complement edges*. These edges indicate that the node they terminate in is to be complemented; if it represents a function f by itself, then with the complement edge it represents $\neg f$. This allows for the sharing of common subexpressions and their complements: so a function $y \wedge x$ is represented by the same subgraph as $\neg(y \wedge x) = \neg y \vee \neg x$.

In order to achieve canonicity, we start by ensuring reducedness of these diagrams. This is done by keeping only one terminal node: the other constant value is obtained by attaching a complement edge to it. However, this change is still not sufficient to guarantee canonicity. For example, consider a one-variable identity function $f(x) = x$. Two possible representations of this function are shown in Figure 2. Thus, reducedness and orderedness are no longer sufficient to guarantee canonicity. Instead, we impose the restriction that the low edge must never be complemented. A complement on a low edge should always be "factored out". Since for boolean logic, for every x, $\neg\neg x = x$, negation on the low edge can be removed by toggling the annotation on the incoming edge and on the remaining, high edge. For example, this operation performed on the diagram in Figure 2(a) yields the diagram in Figure 2(b), which is considered canonical. It is easy to see that this operation yields the same function.

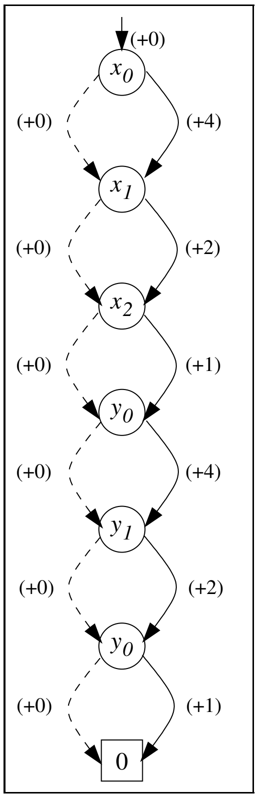

Figure 3: EVDD representing a 3-bit adder.

4.2 Addition Edges

Where D is binary and R is the set of real numbers, edges can be annotated with addition by a constant [12]; this yields *edge-valued decision diagrams* (EVDDs). A further extension of this idea [18] allows edges to be annotated with multiplication by a constant, followed by addition. The diagrams that support these transformations are called *factored edge-valued* DDs (FEVDDs). As in the case of BDDs with complement edges, described above, EVDDs and FEVDDs have exactly one terminal node. This node is usually given the value **0**. In addition, the low edge of each decision node is required to be the identity function (addition of zero in the case of EVDDs).

The example in Figure 3 shows a 3-bit adder represented as an EVDD. In the Figure, the edge annotation $(+n)$ is the function $f(x) = x + n$. Consider the inputs $x = 3$ and $y = 5$, represented by the bit-strings 011 and 101, respectively. For example, we get the edge transformer for x_0 from the low edge, since when $x = 3$, $x_0 = 0$. Composing these edge transformers along the path from the root to the terminal node, including the incoming edge transformer, we get $(+0) \circ (+0) \circ (+2) \circ (+1)$ (for x), followed by $(+4) \circ (+0) \circ (+1)$ (for y). The composition yields $(+8)$, correctly computing the sum of x and y.

$z \setminus xy$	FF	FM	FT	MF	MM	MT	TF	TM	TT
F	M	F	T	F	F	F	F	F	F
M	T	F	M	M	M	M	F	F	F
T	F	M	T	T	T	T	F	F	F

Table 1: Function Q.

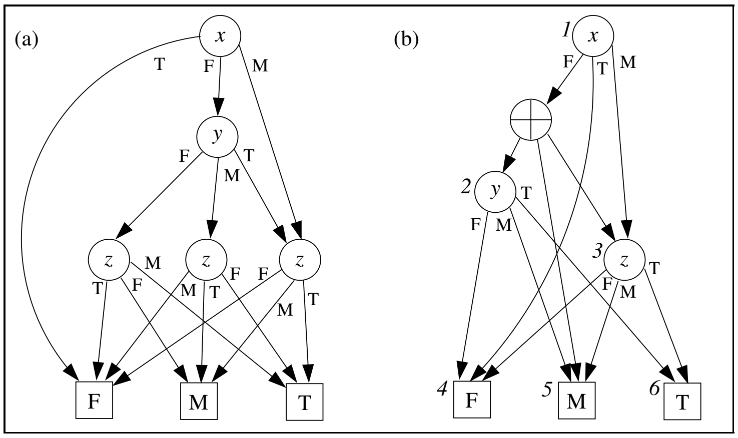

Figure 4: Representation of function Q in Table 1: (a) as an MDD; (b) as a mod-p DD.

4.3 Mod-p Addition Edges

In the above examples, a node u represented not only a function f, but also potentially a function $\varphi \circ f$, where φ is one of the available unary operators. Edges in the diagram can also be annotated with *binary* functions on R, or functions of even higher arity. They must, then, have two or more destination nodes rather than one. In practice this is implemented by adding "operator nodes" which combine its children in the appropriate way.

Andersen et al. [1] described *binary expression diagrams*, with all 16 binary boolean functions as edge transformers. For example, to express the function $f_u \wedge f_{u'}$, binary expression diagrams use an operator node labeled with the function $\wedge$. This node has two children, u and u'. These diagrams lack the canonicity property of standard BDDs, but still find applications in verification.

Decision diagrams with non-unary edge transformers have also been proposed for multiple-valued logic applications. Like binary expression diagrams, *mod-p* decision diagrams of Sack et al. [13] contain decision-free operator nodes which add the functions represented by their children modulo $p = |R|$, the size of the output range.

For example[1], consider representing a function Q defined in Table 1. The function takes three variables, x, y, and z from the three-valued logic $\mathcal{L}_3$, and yields a three-valued result. Figure 4 shows this function represented by an MDD and by a mod-p DD. The mod-p node, indicated by a $\oplus$, represents the combination of its children by addition modulo 3. In this diagram, the $f|_{x=\mathrm{F}}$

[1]This example is taken directly from [13].

cofactor does not need to be represented explicitly. Instead, it can be obtained by combining three already-existing subgraphs. More formally, a mod-p node sums over an indexed set of R-output functions. Thus, edges in mod-p diagrams are either ordinary directed edges (u, u') with no annotation, or directed hyperedges $(u, \{u_i\})$, where the functions represented by each u_i are summed modulo p. Let **label**$(u) = a$, $f = f_u$ and **child**$_u(d) = \{u_1, u_2, u_3, , ..., u_j\}$ for some node u, label a, logic value d and natural number j. Note that we are overloading the function **child**$_u(d)$ to return a *set* of successors, if appropriate. Then,

$$f|_{a=d} = (f_{u_1} + f_{u_2} + f_{u_3} + ... + f_{u_j}) \bmod p$$

For example, using numbers of nodes shown in Figure 4(b), we have: **label**$(\mathit{1}) = x$, $f = f_{\mathit{1}}$, **child**$_{\mathit{1}}$(F) $= \{\mathit{2}, \mathit{3}, \mathit{5}\}$. Therefore,

$$f|_{x=\mathrm{F}} = (f_{\mathit{2}} + f_{\mathit{3}} + f_{\mathit{5}}) \bmod 3$$

In this Figure, the use of mod-p addition reduces the number of decision nodes to 3 (plus one operator node), down from 5. This reduction is obtained at the cost of canonicity. The principal drawback of non-canonical diagrams is that there may be two different subgraphs representing the same function, and checking for equality can no longer be done in constant time. Model-checking involves the computation of fixpoints, which requires frequent checking for equality, so a non-canonical decision diagram, even if it is small, has severe drawbacks. In the following section, we will propose a variety of canonical decision diagram for multiple-valued logic, which also uses modular addition.

5 Canonical Edge-Shifted Decision Diagrams

In this section we present and analyze efficient canonical decision diagrams for multi-valued logic.

5.1 Cyclic Shifts

Here, we propose a set of edge-transformers to allow us to create more compact MDDs. In this construction we borrow the idea of addition from EVDDs, and the modular addition from mod-p diagrams. We also suggest that modular addition of a constant can be used as a unary transformer that guarantees canonicity.

Definition 6 *For a multiple-valued logic* $(\{r_1, \ldots, r_n\}, \wedge, \mathbf{1}, \vee, \mathbf{0}, \{\mathbf{eq}_{r_i}\}_{1 \leq i \leq n})$*, the family of unary operators* Sh_i*,* $1 \leq i \leq n$*, is defined as follows:*

$$Sh_i(r_j) \triangleq r_{1+((i+j-1) \bmod n)} \qquad \text{(definition of } Sh\text{)}$$

For example, consider the three-valued logic $\mathcal{L}_3$, where the logic elements are ordered as F < M < T. Then, Sh_0(F) = F, Sh_1(F) = M, Sh_2(F) = T. Overloading Sh_i to take a *set* of elements rather than a singleton element, we get:

$$\begin{array}{rcl} Sh_0(\{\text{F,M,T}\}) & = & \{\text{F,M,T}\} \\ Sh_1(\{\text{F,M,T}\}) & = & \{\text{M,T,F}\} \\ Sh_2(\{\text{F,M,T}\}) & = & \{\text{T,F,M}\} \end{array}$$

We note that if R is boolean, then $Sh_0 = id$ and $Sh_1 = \neg$; so this definition reduces to the standard one of complement-edge BDDs.

The set of cyclic shifts $\{Sh_i\}_{0 \le i < |R|}$ has a number of properties. For example, exactly one shift takes **0** into any given logic value. Clearly, shifts are invertible, and the composition of two shifts is also a shift. Finally, shifting distributes over **case**. We formalize and prove these properties in Lemma 1.

Lemma 1 *For any range R, the set $SH = \{Sh_i\}_{0 \le i < |R|}$ of cyclic shifts obeys the conditions listed below. Implicitly, i, j, k range over $\{0, \ldots, |R| - 1\}$.*

$\forall r \in R \cdot \exists! i \cdot Sh_i(\mathbf{0}) = r$	(unique image)
$\forall i \cdot Sh_i(\textbf{case}(x, y_1, \ldots, y_n)) = \textbf{case}(x, Sh_i(y_1), \ldots, Sh_i(y_n))$	(distribution over **case**)
$\forall i \cdot \exists! j \cdot Sh_i \circ Sh_j = Sh_0$	(inverse)
$\forall i, j \cdot \exists l \cdot Sh_i \circ Sh_j = Sh_l$	(composition)

Proof:
We prove several of the above properties. The rest are similar.

$$\begin{array}{rl} \text{(composition)} & Sh_i(Sh_j(r_k)) \\ = & \text{(definition of } Sh\text{)} \\ & Sh_i(r_{1+((j+k-1) \bmod n)}) \\ = & \text{(definition of } Sh\text{)} \\ & r_{1+(i+(1+((j+k-1) \bmod n)-1) \bmod n)} \\ = & \text{(ordinary arithmetic)} \\ & r_{1+(i+((j+k-1) \bmod n) \bmod n)} \\ = & \text{(modular arithmetic)} \\ & r_{1+((((i+j) \bmod n)+k-1) \bmod n)} \\ = & \text{(definition of } Sh\text{)} \\ & Sh_{(i+j) \bmod n}(r_k) \\ & \text{Thus, we pick } l = (i+j) \bmod n. \end{array}$$

Since $D = \{d_1, \ldots, d_m\}$, any $x \in D$ is equal to d_j for some j between 1 and m. Without loss of generality, we can replace a free variable x in any expression by d_j, where j is free. Then the proof of distribution over **case** proceeds as follows:

$$
\begin{array}{rl}
\text{(distribution over \textbf{case})} & Sh_i(\mathbf{case}(d_j, y_1, \ldots, y_m)) \\
= & \text{(definition of \textbf{case})} \\
& Sh_i(\mathbf{eq}_1(d_j) \wedge y_1 \vee \ldots \vee \mathbf{eq}_j(d_j) \wedge y_j \vee \ldots \vee \mathbf{eq}_m(d_j) \wedge y_m) \\
= & \text{(definition of \textbf{eq})} \\
& Sh_i(\mathbf{0} \wedge y_1 \vee \ldots \vee \mathbf{1} \wedge y_j \vee \ldots \vee \mathbf{0} \wedge y_m) \\
= & \text{(identity and base laws)} \\
& Sh_i(y_j) \\
= & \text{(identity laws)} \\
& \mathbf{0} \vee \ldots \vee \mathbf{1} \wedge Sh_i(y_j) \vee \ldots \vee \mathbf{0} \\
= & \text{(base laws)} \\
& \mathbf{0} \wedge Sh_i(y_1) \vee \ldots \vee \mathbf{1} \wedge Sh_i(y_j) \vee \ldots \mathbf{0} \wedge Sh_i(y_m) \\
= & \text{(definition of \textbf{eq})} \\
& \mathbf{eq}_1(d_j) \wedge Sh_i(y_1) \vee \ldots \vee \mathbf{eq}_j(d_j) \wedge Sh_i(y_j) \vee \ldots \vee \mathbf{eq}_m(d_j) \wedge Sh_i(y_m) \\
= & \text{(definition of \textbf{case})} \\
& \mathbf{case}(d_j, Sh_1(y_1), \ldots, Sh_i(y_m))
\end{array}
$$

□

The unique image property implies the existence of an injection $\nu : R \rightarrow SH$:

$$\forall r \in R \cdot (\nu(r))(\mathbf{0}) = r \qquad \text{(definition of } \nu)$$

In particular, $\forall r_j \in R \cdot \nu(r_j) = Sh_j$. Finally, we denote the inverse of a shift Sh_i as $(Sh_i)^{-1}$; it is easy to see that $(Sh_i)^{-1} = Sh_{(-i) \bmod n}$.

5.2 Diagrams with Cyclic Shifts

Definition 7 *An* edge-shifted *D-input R-out reduced ordered decision diagram (ESDD) on an ordered set of variables $A = \{a_1, ..., a_k\}$ is an ordered pair (u, φ) where $\varphi \in SH$ is a cyclic shift representing a transformer on the incoming edge, and u is the root of a rooted directed acyclic graph $(V \cup T, E)$ where:*

- *there is exactly one terminal node $\mathbf{t} \in T$, and* $\mathbf{value}(\mathbf{t}) = \mathbf{0}$ (unique terminal)*;*
- *$SH = \{Sh_i\}_{0 \leq i < |R|}$ is the set of cyclic shifts on R satisfying properties in Lemma 1;*
- *$\forall u \in V$, $\mathbf{edge}_u(d)$ is the transformer $Sh_i \in SH$ attached to the edge $(u, \mathbf{child}_u(d))$;*
- *$\forall u \in V$, $\mathbf{edge}_u(d_1)$ is Sh_0* (normality)*;*
- *the diagram is* ordered*: the definition of orderedness is identical to that of MDDs;*
- *the diagram is* reduced*: that is, the following two properties hold:*

$$\forall u \in V \cdot \exists d, d' \in D \cdot (\mathbf{child}_u(d), \mathbf{edge}_u(d)) \neq (\mathbf{child}_u(d'), \mathbf{edge}_u(d')) \qquad \text{(reducedness 1)}$$

$$((\mathbf{label}(u') = \mathbf{label}(u)) \wedge (\forall d \in D \cdot (\mathbf{child}_u(d) = \mathbf{child}_{u'}(d)) \ \wedge \ (\mathbf{edge}_u(d) = \mathbf{edge}_{u'}(d)))) \Rightarrow (u = u') \qquad \text{(reducedness 2)}$$

The function $f_{(u,\varphi)}$ represented by an ESDD (u, φ) is defined recursively, in a similar manner to the standard MDD definition:

$$f_{(u,\varphi)} \triangleq \begin{cases} \varphi(\textbf{case}(\textbf{label}(u), f_{(\textbf{child}_u(d_1), \textbf{edge}_u(d_1))}, \dots, \\ \qquad f_{(\textbf{child}_u(d_m), \textbf{edge}_u(d_m))})) & \text{if } u \in V \\ \varphi(\mathbf{0}) & \text{if } u = \mathbf{t} \end{cases} \qquad \text{(ESDD semantics)}$$

In effect, for an ESDD (u, φ), where u is labeled with some variable with a value d_i, the function evaluation consists of two steps: first, recursively evaluate the ESDD $(\textbf{child}_u(d_i), \textbf{edge}_u(d_i))$ and then apply the shift φ to the result. Recall that the output of an MDD can be computed by tracing a path from the root to a leaf dictated by the variable assignment, and that the value of the terminal node which is reached gives the output. For an ESDD, the same path is traced, all of the edge shifts are composed together, along with the incoming shift φ, and this composed shift is applied to the terminal **0** to give the output. This algorithm is based on the observation of the following lemma.

Lemma 2 *Applying an edge transformer $\psi \in SH$ to a function $f_{(u,\varphi)}$ implies composing the two transformers:*

$$\psi(f_{(u,\varphi)}) \quad = \quad f_{(u,\psi\circ\varphi)} \qquad \text{(factoring lemma)}$$

Proof:

$$\begin{aligned} & \psi(f_{(u,\varphi)}) \\ & \text{(ESDD semantics)} \\ = \;& \psi(\varphi(\textbf{case}(\textbf{label}(u), f_{(\textbf{child}_u(d_1), \textbf{edge}_u(d_1))}, \dots, f_{(\textbf{child}_u(d_m), \textbf{edge}_u(d_m))}))) \\ & \text{(function composition)} \\ = \;& (\psi \circ \varphi)(\textbf{case}(\textbf{label}(u), f_{(\textbf{child}_u(d_1), \textbf{edge}_u(d_1))}, \dots, f_{(\textbf{child}_u(d_m), \textbf{edge}_u(d_m))})) \\ & \text{(ESDD semantics)} \\ = \;& f_{(u,\psi\circ\varphi)} \end{aligned}$$

□

The factoring lemma and the existence of inverses for every edge transformer allow the normality property to be maintained; the edge-shift for the d_1-cofactor can always be factored up to become an incoming edge, and the other edge-shifts transformed appropriately to keep the semantics the same.

Figure 5 shows the function Q from Figure 4 as an edge-shifted decision diagram. Suppose we are interested in evaluating Q when $x = \mathrm{M}$, $y = \mathrm{M}$ and $z = \mathrm{T}$. Let u be the root node. First, apply the incoming edge Sh_0 to the result of computing $(Sh_1, \textbf{child}_u(\mathrm{M}))$; next, we apply $Sh_0 \circ Sh_1$ to $(Sh_2, \textbf{child}_{\textbf{child}_u(\mathrm{M})}(\mathrm{M}))$; finally, we apply $Sh_0 \circ Sh_1 \circ Sh_2$ to (Sh_2, F). This yields $Sh_0(Sh_1(Sh_2(Sh_2(\mathrm{F})))) = Sh_2(\mathrm{F}) = \mathrm{T}$. Table 1 confirms that this is the correct output.

We are now ready to show that ESDDs are canonical: each function can be represented by exactly one ESDD. The intuition is that two functions f and g share a node, that is, $f = f_{(u,\varphi)}$ and $g = f_{(u,\psi)}$, if and only if f can be obtained from g by a cyclic shift of its outputs. This notion is formalized and proven below.

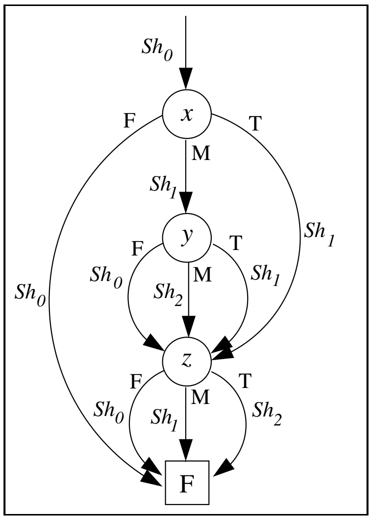

Figure 5: Representation of function Q in Table 1 as an edge-shifted DD.

Theorem 3 *Edge-Shifted Decision Diagrams are canonical, that is, each function can be represented by exactly one ESDD:*

$$\forall f \cdot \exists (u, \varphi) \cdot f = f_{(u,\varphi)} \qquad \text{(representation)}$$
$$\exists (u, \varphi), (u', \psi) \cdot (f_{(u,\varphi)} = f_{(u',\psi)}) \Rightarrow ((u = u') \ \wedge \ (\varphi = \psi)) \qquad \text{(uniqueness)}$$

Proof:
The proof proceeds by induction on the number of variables k.

Base case (k = 0). On 0 variables, f is a constant function. Suppose $f = r_j$ for some $r_j \in R$. We show that $(\nu(r_j), \mathbf{t})$ satisfies conditions of Theorem 3. First, $(\nu(r_j), \mathbf{t})$ is an ESDD that trivially satisfies normality and unique terminal conditions. The remaining properties are shown below.

$$\begin{array}{rl} \text{(representation)} & f_{(\nu(r_j),\mathbf{t})} \\ & \text{(ESDD semantics)} \\ = & (\nu(r_j))(\mathbf{0}) \\ & \text{(definition of } \nu\text{)} \\ = & r_j \end{array}$$

Since f is constant, suppose it can be represented by some other ESDD (φ, t'). Then,

$$\begin{array}{rl} \text{(uniqueness)} & f_{(\varphi,t')} = r_j \\ & \text{(unique terminal)} \\ = & (t' = \mathbf{t}) \wedge (f_{(\varphi,t')} = r_j) \\ & \text{(one-point law)} \\ = & (t' = \mathbf{t}) \wedge (f_{(\varphi,\mathbf{t})} = r_j) \\ & \text{(ESDD semantics)} \\ = & (t' = \mathbf{t}) \wedge (\varphi(\mathbf{0}) = r_j) \\ & \text{(unique image)} \\ \Rightarrow & (t' = \mathbf{t}) \wedge (\varphi = \nu(r_j)) \end{array}$$

Inductive hypothesis (p = k). Assume $\forall 0 \leq p \leq k \cdot \forall f : D^p \rightarrow R$ the statement of Theorem 3 holds.

Inductive step (p = k+1). Let $f : D^{k+1} \rightarrow R$ be a function in $k+1$ variables. By generalized Shannon expansion, $f = \mathbf{case}(x_{k+1}, f|_{x_{k+1}=d_1}, \ldots, f|_{x_{k+1}=d_m})$. Since each cofactor $f|_{x_{k+1}=d_i}$ is over k variables, by inductive hypothesis it can be represented by a canonical ESDD (u_j, φ_j). We use these cofactors to construct the ESDD on $k+1$ vectors as follows:

Case 1: $\forall 1 \leq i, j \leq m \cdot (u_i, \varphi_i) = (u_j, \varphi_j)$. Then,

$$\begin{array}{rl} \text{(representation)} & f \\ = & \text{(generalized Shannon expansion)} \\ & \mathbf{case}(x_{k+1}, f_{(u_i,\varphi_i)}, \ldots, f_{(u_i,\varphi_i)}) \\ = & (\mathbf{case}\text{ idempotence}) \\ & f_{(u_i,\varphi_i)} \end{array}$$

By inductive hypothesis, (u_i, φ_i) is also a unique ESDD.

Case 2: $\exists 1 \leq i, j \leq m \cdot (u_i, \varphi_i) \neq (u_j, \varphi_j)$. Construct the following ESDD (u, φ):

1. $\mathbf{label}_u$:= x_{k+1}
2. $\mathbf{child}_u(d_j)$:= u_j
3. $\mathbf{edge}_u(d_j)$:= $(\varphi_1^{-1} \circ \varphi_j)$, for all $1 \leq j \leq m$. Conversely, $\varphi_j = \varphi_1 \circ \mathbf{edge}_u(d_j)$.
4. φ := φ_1

The resulting construction is an ESDD because unique terminal property holds by induction and normality is preserved by step (3) of the above construction. It is done by ensuring that φ_1, the edge transformer of the d_1 cofactor of f, is factored out. The remaining proofs are given below.

$$\begin{array}{rl} \text{(representation)} & f_{(u,\varphi)} \\ = & \text{(ESDD semantics)} \\ & \varphi(\mathbf{case}(x_{k+1}, f_{(\mathbf{child}_u(d_1), \mathbf{edge}_u(d_1))}, \ldots, f_{(\mathbf{child}_u(d_m), \mathbf{edge}_u(d_m))})) \\ = & \text{(construction of } u\text{), } (\varphi = \varphi_1) \\ & \varphi_1(\mathbf{case}(x_{k+1}, f_{(u_1, \varphi_1^{-1} \circ \varphi_1)}, \ldots, f_{(u_m, \varphi_1^{-1} \circ \varphi_m)})) \\ = & \text{(inverse)} \\ & \varphi_1(\mathbf{case}(x_{k+1}, f_{(u_1, Sh_0)}, \ldots, f_{(u_m, \varphi_1^{-1} \circ \varphi_m)})) \\ = & \text{(distribution over } \mathbf{case}\text{)} \\ & \mathbf{case}(x_{k+1}, \varphi_1(f_{(u_1, Sh_0)}), \ldots, \varphi_1(f_{(u_m, \varphi_1^{-1} \circ \varphi_m)})) \\ = & \text{(factoring lemma)} \\ & \mathbf{case}(x_{k+1}, f_{(u_1, \varphi_1 \circ Sh_0)}, \ldots, f_{(u_m, \varphi_1 \circ \varphi_1^{-1} \circ \varphi_m)}) \\ = & \text{(function composition), (identity), (inverse)} \\ & \mathbf{case}(x_{k+1}, f_{(u_1, \varphi_1)}, \ldots, f_{(u_m, \varphi_m)}) \\ = & \text{(inductive hypothesis)} \\ & \mathbf{case}(x_{k+1}, f|_{(x_{k+1}=d_1)}, \ldots, f|_{(x_{k+1}=d_m)}) \\ = & \text{(generalized Shannon expansion)} \\ & f \end{array}$$

Our last proof shows uniqueness of the construction. We show that if (u', ψ) is another reduced, ordered, and normal ESDD representing f, then $u = u'$ and $\varphi = \psi$.

$$
\begin{array}{rl}
\text{(uniqueness)} & f = f_{(u',\psi)} \\
\Leftrightarrow & \text{(ESDD semantics)} \\
& f = \psi(\mathbf{case}(x_{k+1}, f_{(\mathbf{child}_{u'}(d_1),\mathbf{edge}_{u'}(d_1))}, \ldots, f_{(\mathbf{child}_{u'}(d_m),\mathbf{edge}_{u'}(d_m))})) \\
\Leftrightarrow & \text{(distribution over } \mathbf{case}\text{)} \\
& f = \mathbf{case}(x_{k+1}, \psi(f_{(\mathbf{child}_{u'}(d_1),\mathbf{edge}_{u'}(d_1))}), \ldots, \psi(f_{(\mathbf{child}_{u'}(d_m),\mathbf{edge}_{u'}(d_m))})) \\
\Leftrightarrow & \text{(factoring lemma), (generalized Shannon expansion)} \\
& \mathbf{case}(x_{k+1}, f|_{x_{k+1}=d_1}, \ldots, f|_{x_{k+1}=d_m}) = \\
& \ \mathbf{case}(x_{k+1}, f_{(\mathbf{child}_{u'}(d_1),\psi\circ\mathbf{edge}_{u'}(d_1))}, \ldots, f_{(\mathbf{child}'_{u'}(d_m),\psi\circ\mathbf{edge}_{u'}(d_m))}) \\
\Leftrightarrow & (\mathbf{case}\text{ equality}) \\
& \forall d_i \in D \cdot f|_{x_{k+1}=d_i} = f_{(\mathbf{child}_{u'}(d_i),\psi\circ\mathbf{edge}_{u'}(d_i))} \\
\Leftrightarrow & \text{(inductive hypothesis)} \\
& \forall d_i \in D \cdot (\mathbf{child}_{u'}(d_i) = u_i) \ \wedge \ (\psi \circ \mathbf{edge}_{u'}(d_i) = \varphi_i) \\
\Leftrightarrow & \text{(construction of } u\text{)} \\
& \forall d_i \in D \cdot (\mathbf{child}_{u'}(d_i) = \mathbf{child}_u(d_i)) \ \wedge \ (\psi \circ \mathbf{edge}_{u'}(d_i) = \varphi \circ \mathbf{edge}_u(d_i)) \\
\Leftrightarrow & \text{(specialization)} \\
& (\forall d_i \in D \cdot (\mathbf{child}_{u'}(d_i) = \mathbf{child}_u(d_i)) \ \wedge \ (\psi \circ \mathbf{edge}_{u'}(d_i) = \varphi \circ \mathbf{edge}_u(d_i))) \ \wedge \\
& \ (\psi \circ \mathbf{edge}_{u'}(d_1) = \varphi \circ \mathbf{edge}_u(d_1)) \\
\Leftrightarrow & \text{(normality of } (u',\psi)\text{), (construction of } (u,\varphi)\text{)} \\
& (\forall d_i \in D \cdot (\mathbf{child}_{u'}(d_i) = \mathbf{child}_u(d_i)) \ \wedge \ (\psi \circ \mathbf{edge}_{u'}(d_i) = \varphi \circ \mathbf{edge}_u(d_i))) \ \wedge \\
& \ (\psi \circ Sh_0 = \varphi \circ Sh_0) \\
\Leftrightarrow & \text{(inverse), (identity)} \\
& (\forall d_i \in D \cdot (\mathbf{child}_{u'}(d_i) = \mathbf{child}_u(d_i)) \ \wedge \ (\psi \circ \mathbf{edge}_{u'}(d_i) = \varphi \circ \mathbf{edge}_u(d_i))) \ \wedge \\
& \ (\psi = \varphi) \\
\Leftrightarrow & \text{(one-point law), (inverse)} \\
& (\forall d_i \in D \cdot (\mathbf{child}_{u'}(d_i) = \mathbf{child}_u(d_i)) \ \wedge \ (\mathbf{edge}_{u'}(d_i) = \mathbf{edge}_u(d_i))) \ \wedge \ (\psi = \varphi) \\
\Rightarrow & \text{(reducedness 2)} \\
& u = u' \wedge \psi = \varphi
\end{array}
$$

Thus, Theorem 3 holds for any number of variables. □

5.3 Operations on ESDDs

We describe two of the basic operations on edge-shifted decision diagrams, given in Figure 6. For simplicity of presentation, we assume that values of terminal nodes are represented as integers: if the value of a terminal node is r_i, then it is labeled with i. For example, the value of F in logic $\mathcal{L}_3$ is assumed to be 1, the value of M is assumed to be 2, and the value of T is assumed to be 3. Likewise, edge-shifts are represented by integers: if the ith outgoing edge of node u is labeled with the transformer Sh_j, then $\mathbf{edge}_u(i)$ is an integer j. Thus, the composition of edge transformers, and the application of transformers to constants are computed using simple modular addition. For instance, as we demonstrated previously in slightly different terms, the

```
// input: two ESDDs
// returns: an ESDD
function Apply( (u,φ) : ESDD, (v,ψ) : ESDD, op : operator)
    children : array[d_1..d_m] of ESDD
    if (u = t ∧ v = t) // u = v is the terminal node
        return (t, φ op ψ)
    else if (label(u) = label(v))
        foreach d ∈ D
            children[d] := Apply(child_u(d), (φ + edge_u(d)) mod n),
                                 (child_v(d), (ψ + edge_v(d)) mod n), op)
    else if (label(u) < label(v))
        foreach d ∈ D
            children[d] := Apply((child_u(d), (φ + edge_u(d)) mod n), (v,ψ), op)
    else // label(u) > label(v)
        foreach d ∈ D
            children[d] := Apply((u,φ), (child_v(i), (ψ + edge_v(i)) mod n), op)
        return Build(label(u), children)

// returns: an ESDD
function Restrict ((u,φ) : ESDD, a : A, d : D)
    children : array[d_1..d_m] of ESDD
    if (label(u) > a)
        return (u,φ)
    else if (label(u) < a)
        foreach d ∈ D
            children[d] := Restrict((child_u(d), edge_u(d)), a, d)
        return Build(label(u), children)
    else // label(u) = a
        return (child_u(d), (φ + edge_u(d)) mod n)
```

Figure 6: Operations `Apply` and `Restrict` of ESDDs.

path in Figure 5 dictated by the assignment $x = \mathrm{M}$, $y = \mathrm{M}$ and $z = \mathrm{T}$ is labeled with the edge values 1, 2, 2 and leads to 1, for the output $(1+2+2+1) \bmod 3 = 3$. This corresponds to the value T in the logic.

The function `Apply` is used to apply a binary operation $\diamond \in \{\wedge, \vee\}$ to two ESDDs (u, φ) and (v, ψ). There are two major cases: (1) either the root of both diagrams is the constant node, or (2) one or both is non-constant. In the first case, $u = v = \mathtt{t}$, and the result is the ESDD $(\mathtt{t}, \nu(\varphi(\mathbf{0}) \diamond \psi(\mathbf{0})))$. In the second case, suppose the roots of both diagrams have the same label. The other cases are similar. `Apply` is called recursively on the children of u and v, with the incoming edges factored through the nodes, that is, `Apply` is called with $(\mathbf{child}_u(d_1), \varphi \circ$

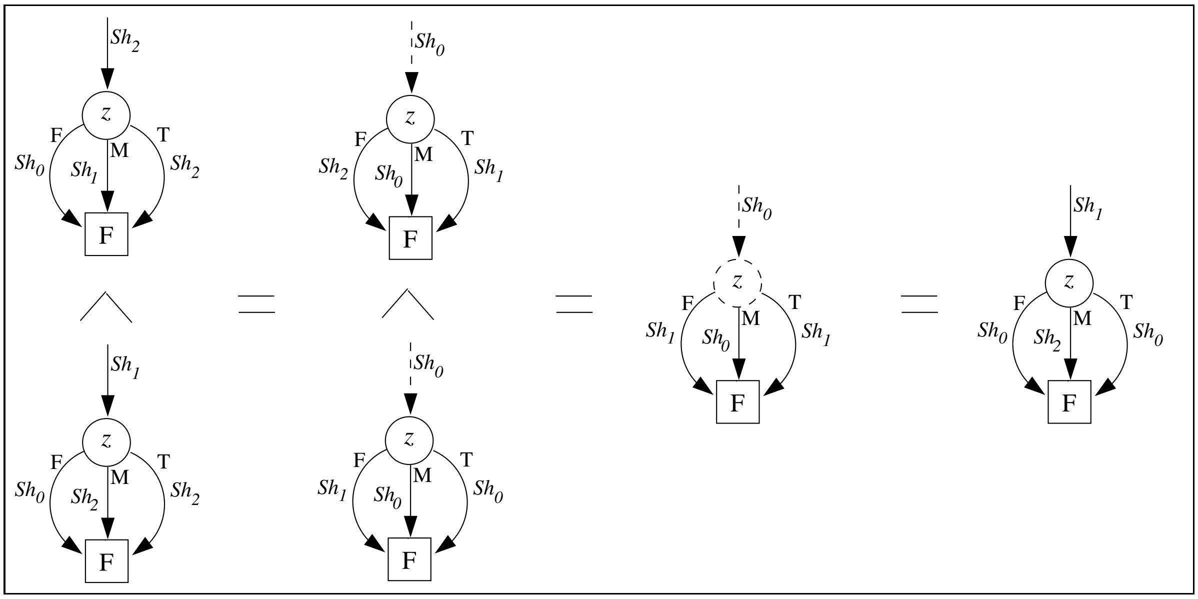

Figure 7: Illustration of the `Apply` algorithm.

$\mathbf{edge}_u(d_1))$ and $(\mathbf{child}_v(d_1), \psi \circ \mathbf{edge}_v(d_1))$ as operands, yielding an ESDD for the d_1-cofactor of $f_{(u,\varphi)} \diamond f_{(v,\psi)}$. The resulting construction is passed into the function `Build` which not only maintains the uniqueness of nodes and thus guarantees reducedness, but also ensures that the output ESDD preserves normality.

In Figure 7, we illustrate the use of `Apply` to compute the conjunction of two one-variable functions. The ESDDs for these functions appear on the far left of the Figure, one above the other. In the next pair of diagrams to the right, we see how the incoming edges have been pushed over the decision node: the incoming edge is now dashed, to indicate that we no longer need to consider it. Next, the cofactors of the conjunction are computed by pairwise conjunction of cofactors of the operands. For example, (Sh_2, F) is conjoined with (Sh_1, F) to yield (Sh_1, F) in the result. This represents the computation T $\wedge$ M = M. We see the three cofactors of the conjunction in the third diagram, where the z-node is dashed to indicate that it has yet to be properly constructed. The normalization is restored by factoring out Sh_1 and placing it on the incoming edge of the new node. The other two edge-shifts of the node's children are transformed similarly. The resulting ESDD is shown on the far right in Figure 7.

The function `Restrict` is used to substitute a value for one variable in the decision diagram; say the variable is x, to be restricted to value d. This is performed, as in the case of `Apply`, via recursive descent through the diagram, which proceeds until a node labeled with x is found. Let this node be called u; suppose the value to be substituted is d. Then, for every node v where $\mathbf{child}_v(d') = u$ for some d', two things must be done: first, $\mathbf{child}_v(d')$ is made equal to $\mathbf{child}_u(d)$; next, $\mathbf{edge}_v(d')$ is changed to $\mathbf{edge}_v(d') \circ \mathbf{edge}_u(d)$. If u is the terminus of an root edge, this must be likewise composed with $\mathbf{edge}_u(d)$. After this is done, u is no longer part of the diagram, and can be removed. The recursive descent guarantees that this will be carried out for *every*

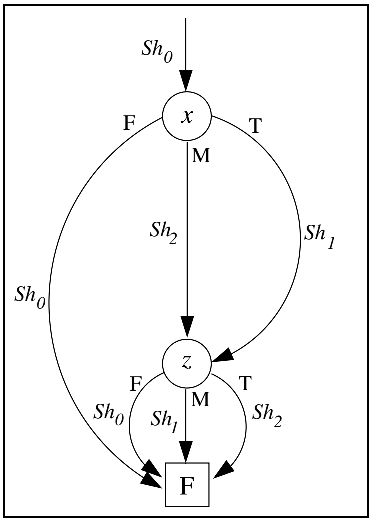

Figure 8: Result of restricting y to T in ESDD in Figure 5.

node labeled with x, and thus it can be shown that `Restrict` produces an ESDD for the function $f|_{x=d}$, which does not depend on x.

Consider restricting y to T in Figure 5. There is one incoming edge to the y-node – the M-edge of the x-node, labeled with Sh_1. The T-edge of the y-node is also labeled with Sh_1, and leads to the z-node. The elimination of the y-node by `Restrict` causes these two edges to be joined together, and their shifts must likewise be composed; thus, the end result is that the M-edge of x leads directly to the z-node, labeled with the shift $Sh_1 \circ Sh_1 = Sh_2$. The resulting ESDD is shown in Figure 8.

To assess the space efficiency of edge-shifted diagrams, refer to the representation of function Q via the different *canonical* decision diagrams (Figures 5 and 4(a)). Given the same variable ordering, the MDD representation requires 5 decision nodes. One the other hand, the ESDD representation requires only 3 decision nodes. Each node, of course, is 6 bits larger, because of the space needed to store edge annotations (2 bits per edge). Making a moderate assumption that it takes 16 bits to store a pointer to a child node, an ordinary ternary DD node requires 48 bits, an edge-shifted ternary DD node requires 54, and thus is larger by a factor of 1.125. The three decision nodes required in this diagram take the space equal to 3.375 ordinary decision nodes, which is still a substantial improvement. We confirm this improvement on experimental results below.

5.4 Experimental Methodology

In our experiments, we compare two representations of multiple-valued logic functions: ordinary MDDs, and edge-shifted DDs. In this comparison, we take number of nodes in the diagram (for a chosen variable ordering) as the criterion.

The benchmarks we use are simple circuits: adders and multipliers of 3, 5, and 7 bits. Both

	Node Count		ESDD
	MDD	ESDD	(Adjusted Size)
adr3	13	6	7
adr5	26	15	17
adr7	43	28	31
mul3	28	24	26
mul5	175	121	133
mul7	912	562	618

Table 2: Experimental results.

BDDs and EVDDs with binary input and integer output are known to be poor representations for multipliers [4]. It has been shown [14] that grouping binary inputs and outputs into 4-valued inputs and outputs yields a considerable improvement in size for adders. We hope to see, first, whether the use of this grouping, together with edge-shifts, can do well for multipliers as well as for adders; and also whether the use of edge-shifts can further reduce the storage space needed for adders, all other factors remaining the same.

The 4-valued inputs to the functions are obtained by pairing bits from each operand. For instance, a 3-bit adder that sums two numbers is represented in binary as $x_0x_1x_2$ and $y_0y_1y_2$. These 6 input bits are grouped in pairs to yield the three four-valued inputs $X_0 = (x_0, y_0)$, $X_1 = (x_1, y_1)$, and $X_2 = (x_2, y_2)$. The outputs – in this case the bits z_0, z_1, z_2, z_3 – are simply paired as $Z_0 = (z_0, z_1)$ and $Z_1 = (z_2, z_3)$. This grouping of inputs and outputs is based on previous experimental work on MDDs [14]; the results we obtain for adders can be compared directly with the results of [14], which did not deal with edge-transformers.

5.5 Experimental Results

In order to attempt to isolate the effects of edge-shifts, we directly compare the sizes of MDDs and edge-shifted MDDs for each benchmark function. These results appear in the leftmost two columns of Table 2. Note that we *do not* measure potential trade-offs in the execution time of the operations of the two DDs. The results clearly show that the use of edge-shifts decreases the size considerably for both adders and multipliers. For adders, the improvement is independent of the number of bits, cutting the number of nodes roughly in half. In contrast, for multipliers, the improvement becomes more marked as the number of bits increases. For instance, a 3-bit multiplier requires 28 MDD nodes and 24 ESDD nodes, only a slight improvement; but a 7-bit multiplier requires 912 MDD nodes and 562 ESDD nodes.

However, these results require some adjustment before we can credit them fully. The reason is that introducing edge annotation increases the amount of space required to store a node. In order to do the comparison, we adjust the node-count for ESDDs based on an estimate of the difference in size between ESDD nodes and unadorned MDD nodes. We compute this factor for the increase in node-size that edge annotations bring about, under the assumption that edge annotation adds 2×4=8 bits to a 4-way decision node. We further assume that pointers are 16 bits in size, so a 4-way decision node requires 4×16=64 bits to store pointers to its child nodes. When an extra pointer for the label is factored in, this increases the size to 64+16=80 bits. Since edge annotation adds 8 bits, the total for an edge-annotated node is 88 bits. So a node in an edge-shifted diagram is 88/80=1.1 times the size of a node in a standard MDD. Performing this calculation yields the adjusted size for ESDDs, which appears in the rightmost column of Table 2. We can see that this adjustment does not affect our conclusions: the improvement for adders remains at about a half, and for multipliers the ratio between the adjusted ESDD node size and the MDD size goes down from 26/28=0.93 to 618/912=0.68 as the number of bits increases, representing a substantial improvement.

6 Canonicity Conditions for Decision Diagrams with Unary Edge Transformers

We now turn to identifying conditions that allow to prove canonicity of an arbitrary set Φ of edge transformers. In ESDDs, $\Phi = SH$.

Let Φ have the following properties: given a distinguished logic element $\mathbf{0}$, and some other element r, there is precisely one transformation $\varphi \in \Phi$ with $\varphi(\mathbf{0}) = r$. Furthermore, the identity function $\mathbf{id}(r) = r$ must be in Φ; Φ must be closed under composition, and for any $\varphi \in \Phi$ there must be an inverse $\varphi^{-1} \in \Phi$. These conditions are equivalent to Φ forming a group. Formally,

$\forall d \in D \cdot \exists! \varphi \in \Phi \cdot d = \varphi(\mathbf{value}(\mathbf{t}))$	(unique image)
$\forall \psi \in \Phi \cdot \psi \circ \mathbf{case}(x, \varphi_1, \ldots, \varphi_n) = \mathbf{case}(x, \psi \circ \varphi_1, \ldots, \psi \circ \varphi_n)$	(distribution over **case**)
$\mathbf{id} \in \Phi$	(identity)
$\forall \varphi, \psi \in \Phi \cdot \varphi \circ \psi \in \Phi$	(composition)
$\forall \varphi \in \Phi \cdot \exists \varphi^{-1} \in \Phi \cdot \varphi \circ \varphi^{-1} = \mathbf{id}$	(inverse)

The set SH of edge transformers used in ESDDs satisfies the above properties by Lemma 1 when Sh_0 is the **id** function of Φ.

Now we define *generalized edge-shifted* decision diagrams over the set of edge transformers Φ:

Definition 8 *A* generalized edge-shifted *D-input R-out reduced ordered decision diagram (GESDD) on an ordered set of variables $A = \{a_1, ..., a_k\}$ is an ordered pair (u, φ) where $\varphi \in \Phi$ is a unary transformer on the incoming edge, and u is the root of a rooted directed acyclic graph $(V \cup T, E)$ where:*

- *there is exactly one terminal node $\mathbf{t} \in T$, and* $\mathbf{value}(\mathbf{t}) = \mathbf{0}$ (unique terminal)*;*
- $\forall u \in V$*,* $\mathbf{edge}_u(d)$ *is a transformer $\varphi \in \Phi$ attached to the edge $(u, \mathbf{child}_u(d))$;*
- $\forall u \in V \cdot \mathbf{edge}_u(d_1) = \mathbf{id}$ (normality)*;*

- *the diagram is* ordered *and* reduced.

Clearly, ESDDs are an instance of GESDDs.

We are now ready to show that GESDDs are canonical: each function can be represented by exactly one GESDD. The intuition, as in ESDDs, is that two functions f and g share a node if and only if f can be obtained from g by an application of some $\varphi \in \Phi$.

Theorem 4 *Generalized edge-shifted Decision Diagrams are canonical, i.e.*

$$\begin{array}{ll} \forall f \cdot \exists (u, \varphi) \cdot f = f_{(u,\varphi)} & \text{(representation)} \\ \exists (u, \varphi), (u', \psi) \cdot (f_{(u,\varphi)} = f_{(u',\psi)}) \Rightarrow ((u = u') \ \wedge \ (\varphi = \psi)) & \text{(uniqueness)} \end{array}$$

Proof:
The proof is virtually identical to that of Theorem 3 and is omitted here. □

7 Conclusion and Future Work

Compact canonical representations of multiple-valued functions are essential in many applications, in particular, model-checking and circuit design. Compactness can be achieved by annotating edges with transformers that get invoked when the edge is traversed. In this paper we proposed and evaluated edge-shifted decision diagrams (ESDDs). They enjoy canonicity and allow for smaller sizes and greater sharing than conventional diagrams for multiple-valued logic – MDDs. We then generalized these results to state sufficient conditions for guaranteeing canonicity when the transformers are unary, as in the case of ESDDs. We hope that the work presented in this paper will lead to creation of even better decision diagrams to improve symbolic reasoning in multiple-valued logic.

The work reported here can be carried forward in a number of ways. First of all, we can improve our experiments by reproducing them on a larger benchmark suite, with comparisons not only with MDDs but with other multi-valued decision diagrams.

Second, the sufficient conditions we stated in this paper are almost certainly too strong. We are now insterested in finding *necessary* conditions.

Third, we want to explore the space of possible edge transformers for multiple-valued (but finite) logics. We started with cyclic shifts, as the most obvious; however, for a given problem domain, there may be a group of more effective transformers that makes for more sharing and faster execution.

Finally, we have started looking for sufficient conditions for canonicity of decision diagrams with non-unary edge transformers: binary, as in binary expression diagrams [1] or of higher arity, as in mod-p decision diagrams [13]. Our preliminary results show that such sufficient conditions, if they exist, might be too limiting to produce canonical data structures of practical use.

References

[1] H.R. Andersen and H. Hulgaard. "Boolean Expression Diagrams". In *LICS: IEEE Symposium on Logic in Computer Science*, 1997.

[2] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Approach*. Springer-Verlag, 1998.

[3] R. E. Bryant. "Graph-based algorithms for boolean function manipulation.". *Transactions on Computers*, 8(C-35):677–691, 1986.

[4] R.E. Bryant. "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification". In *IEEE/ACM International Conference on Computer Aided Design, ICCAD, San Jose/CA*, pages 236–243. IEEE CS Press, Los Alamitos, 1995.

[5] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking". Submitted for publication, October 2001.

[6] M. Chechik, S. Easterbrook, and V. Petrovykh. "Model-Checking Over Multi-Valued Logics". In *Proceedings of FME'01*, volume 2021 of *LNCS*, pages 72–98. Springer, March 2001.

[7] E.M. Clarke, E.A. Emerson, and A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[8] E. Dubrova. "Multiple-Valued Logic in VLSI: Challenges and Opportunities". In *Proceedings of NORCHIP'99*, Oslo, Norway, 1999.

[9] R. Hähnle and W. Kernig. "Verification of Switch-Level Designs with Many-Valued Logic". In *Proceedings of International Conference on Logic Programming and Automated Reasoning*, volume 698 of *LNCS*, pages 158–169, 1993.

[10] E.C.R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1993.

[11] S. C. Kleene. *Introduction to Metamathematics*. New York: Van Nostrand, 1952.

[12] Y.-T. Lai and S. Sastry. "Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification". In *Design Automation Conference*, pages 608–613, 1992.

[13] H. Sack, E. Dubrova, and C. Meinel. "Representation of Multiple-Valued Functions with Mod-p Decision Diagrams". In *Proceedings of IEEE/ACM International Workshop on Logic Synthesis (IWLS2000)*, Dana Point, CA, USA, 2000.

[14] T. Sasao and J.T. Butler. "A Method to Represent Multiple-Output Switching Functions Using Multi-Valued Decision Diagrams". In *Proceedings of IEEE International Symposium on Multiple-Valued Logic*, pages 248–254, Santiago de Compostela, Spain, 1996.

[15] K.C. Smith. "The prospects for multivalued logic: A technology and applications view". *IEEE Transactions on Computers*, C-30(9):619–634, 1981.

[16] Fabio Somenzi. "Binary Decision Diagrams". In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*, pages 303–366. IOS Press, 1999.

[17] A. Srinivasan, T. Kam, S. Malik, and R.E. Brayton. "Algorithms for Discrete Function Manipulation". In *IEEE International Conference on Computer-Aided Design*, pages 92–95, 1990.

[18] P. Tafertshofer and M. Pedram. "Factored Edge-Valued Binary Decision Diagrams". *Formal Methods in System Design: An International Journal*, 10(2/3):243–270, April 1997.

[19] The VIS Group. "VIS: A System for Verification and Synthesis". In R. Alur and T. Henzinger, editors, *Proc. 8th International Conference on Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 428–432. Springer-Verlag, 1996.