12

Implementation of a Multiple-Domain Decision Diagram Package

Stefan Höreth
Department of Electrical Engineering
Darmstadt University of Technology
Merckstr. 25, 64283 Darmstadt, Germany
E-Mail: sth@rs.E-Technik.TH-Darmstadt.DE

Abstract

Graph-based symbolic techniques are part of many synthesis and verification tools. Problems occur if an application requires multiple graph types to model complex designs, since there are many exponential gaps (for time and space) between different types of decision diagrams.

This paper presents hybrid graph manipulation algorithms that integrate common decision diagrams into a single graph manipulation package. An important feature of the presented work is that graph operations are no longer restricted to a single graph type or to a single decomposition type list. Operations on multiple graph types are using an implicit type conversion scheme that prevents from many exponential gaps between different types of decision diagrams.

The package implementation provides the graph types BDD, FDD, OKFDD, MTBDD, EVBDD, BMD, *BMD, p*BMD, HDD, K*BMD and ZDD in order to represent boolean and integer functions as well as sets. Applications of the presented method exist virtually in any approach based on decision diagrams. The paper investigates manipulation of bit-level and word-level functions as well as bit selection from arithmetic expressions.

A convenient extension of DD packages is the ability to dynamically adapt variable ordering. This technique called 'sifting' has been first introduced for BDDs. This paper defines an extension to *BMDs, called positive *BMDs (p*BMDs), that allows for dynamic variable reordering. Reordering techniques in the package can be applied even if multiple graph types are used together.

Keywords

Hybrid graph algorithms, p*BMD, dynamic reordering, decision diagrams

1 INTRODUCTION

Graph-based representations of discrete functions have always been a major concern for VLSI CAD. State-of-the-art symbolic techniques for the synthesis and verification of digital circuits are often based on decision diagrams (DDs).

Naturally, different graph types have been proposed for solving specialized design tasks. Most notably, BDDs (Bryant 1986) are used for efficient representation and manipulation of boolean functions, while OKFDDs (Sarabi *et al.* 1994) have been proven to be the most compact boolean decision diagram (cf. (Drechsler *et al.* 1995)). OKFDDs can be used to resemble BDDs as well as functional decision diagrams (FDDs) (Kebschull *et al.* 1992).

Arithmetic decision diagrams are mappings from boolean values into the integer domain. They can be used to model bit-level (boolean) functionality as well as word-level (arithmetic) circuit specifications. Examples of arithmetic decision diagrams are *BMD (Bryant *et al.* 1995), MTBDD (McMillan *el al.* 1993), EVBDD (Sastry *et al.* 1992), HDD (Zhao *et al.* 1995) or K*BMD (Drechsler *et al.* 1996). Arithmetic decision diagrams have been successfully applied where (vectors of) boolean decision diagrams fail, e.g. for verification of the multiplier function (Chen *et al.* 1994) or in word-level model checking (Zhao *et al.* 1996).

On the contrary, there exist applications, where boolean decision diagrams are more compact than arithmetic graph types. For example, selecting the least significant bit (LSB) in a BMD by performing a *modulo-2* operation is an easy task if the result is represented as an FDD. In this case, the final graph is obtained simply by replacing the (integer) terminal nodes of the BMD with their LSB and reducing the graph. The FDD representation of the selected bit is guaranteed to be smaller (or equal) to the size of the original BMD, while the size of representations like *BMD or BDD can be much larger (i.e. worst-case exponential in the number of variables). An overview of exponential gaps between decision diagrams is given in (Enders *et al.* 1997). The paper proves, that there exists no single graph type that can represent arithmetic and boolean functions with the same efficiency.

Consequently, it is necessary to include multiple graph types in a single graph manipulation package. For the design of such a package, it is crucial to support a flexible graph manipulation scheme that avoids explicit type conversions as much as possible.

Package implementations exist, that provide a basic set of graph types (e.g. the CUDD package or the work of Zhao on HDDs). However, these implementations are restricted in the sense that arguments of operations must be of the same graph type and type conversions are always explicit.

This paper presents an integrating, algebraic approach to manipulation of decision diagrams. Functions are represented by common decision diagrams, but package operations are no longer restricted to a particular graph type rather than the algebraic domain of the represented functions. For example,

a boolean AND operation can take a *BMD (representing a boolean function) and a BDD as arguments and return any graph type that can represent boolean functions. Type conversion of arguments is done implicitly while constructing the result.

The package implementation supports the domains Boolean, Integer, and Sets. Supported graph types are BDDs, FDDs, OKFDDs, (*)BMDs, p*BMDs, MTBDDs, EVBDDs, HDDs, K*BMDs and ZDDs (Minato 1993). New graph types can be easily added if they are based on Kronecker decompositions. The package also supports multiple instantiations of the same graph type in order to have multiple decomposition type lists for OKFDDs, HDDs and K*BMDs.

A serious restriction of DD packages is that graphs must share a total ordering of variables. A convenient package extension is therefore the ability to dynamically adapt the ordering of variables (Yajima *et al.* 1991) without user intervention and without being visible to the user's application. This method called 'sifting' has been first introduced for BDDs (Rudell 1993) and has been extended to OKFDDs (Drechsler *et al.* 1995). However, efficient implementations of this functionality are missing for important graph types like *BMDs or K*BMDs.

This paper presents a modification of *BMDs, called positive *BMDs (p*BMDs), that allows to apply dynamic reordering techniques. Based on this extension, the presented DD package supports local exchange of variables even if different graph types share sets of variables. With the exception of *BMDs and K*BMDs, local variable reordering is supported for all graph types mentioned above.

2 ALGEBRAIC DOMAINS AND DECISION DIAGRAMS

Current decision diagram packages describe operations in terms of graphs and restrict arguments of operations to a single graph type. However, from a mathematical point of view, this is not required. Arguments of operations must be from a particular algebraic domain, not of a particular type of representation. In order to support a more flexible manipulation scheme, the notion of the native domain of a decision diagram is introduced. The native domain of a decision diagram is the "largest" domain where this graph type can represent all possible domain functions.

Figure 1 shows common decision diagrams and their native domains. Examples of boolean decision diagrams (native domain **B**) are BDDs, FDDs and OKFDDs. DDs representing integer function (native domain **Z**) are MTBDD, EVBDD, *BMD, HDD or K*BMD. Zero-suppressed DDs (ZDD) are representations of sets. This scheme could be extended to super-domains of the integers like rational or complex numbers.

Although decision diagrams have a fixed native domain, they can be used to describe functions in other domains as well. For example, a *BMD describing

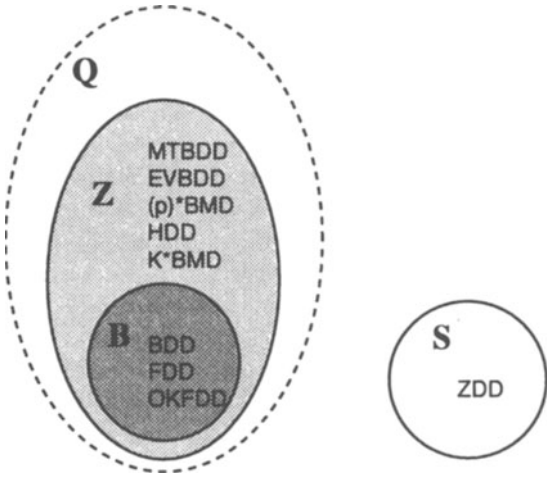

Figure 1 Algebraic Domains and Decision Diagrams.

a 0/1 integer-valued function is interpreted as a representation of a boolean function if its current domain is **B**. Analogously a BDD can be treated as a representation of a function in the integer domain. In general, any graph type from a super-domain can be used to describe any function in a (true) sub-domain. Clearly, graph types from sub-domains can only represent a restricted set of functions in a super-domain.

The presented DD package allows any combination of graph types for package operations, as long as argument graphs have proper current domains. Operations are classified according to the supported domains and can only return graph types with proper native domain. Additionally, there exist functions to explicitly change the domain and/or the type of graph representations.

Note that the current domain of a DD is a graph property that is kept at the user level – it is not part of the graph representation.

2.1 Basic Definitions

This section defines basic notations and manipulation functions used for the description of algorithms working on different graph types. In the sequel, only reduced graphs are considered. Graph structure is defined in the usual way, but – in addition – inner nodes are labelled with the graph type and with the decomposition type. Edges denote functions and can contain edge labels. Terminal nodes simply contain symbols representing constant values. All graph types share the representation of constant symbols. In particular, the representation of the boolean constants *true* and *false* is identical to the

integer constants '1' and '0', respectively. Constant symbols are interpreted depending on their current domain or by the operation that encounters them.

Semantics are given to the graph structure by associating nodes with function decompositions. In the following, a binary decomposition that can be described in terms of equation (1), will be called a Kronecker decomposition. Functions $d_{low}(x_i)$ and $d_{high}(x_i)$ only depend on the boolean variable x_i and (or) on constant values. Operations + and * represent addition and multiplication in the domain of function f.

$$\begin{aligned} f \;=\; & d_{low}(x_i) * f_{low}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) + \\ & d_{high}(x_i) * f_{high}(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \end{aligned} \tag{1}$$

For Kronecker decompositions, not all pairs of functions $d_{low}(x_i)$, $d_{high}(x_i)$ are allowed or useful(Drechsler *et al.* 1995). In the sequel, a valid tuple

$$dt(x_i) := [+, *, d_{low}(x_i), d_{high}(x_i)]$$

will be referred to as the *type* of the decomposition. Decomposition types are associated to the variables x_i with the help of a decomposition type list (DTL):

$$dtl := (dt(x_1), \ldots, dt(x_n)).$$

Typically, only a limited number of decomposition types are used in a DTL. For example, the DTL for BDDs consist of Shannon decompositions only (i.e. for all variables $dt_{BDD}(x_i) := [\oplus, \wedge, \overline{x_i}, x_i]$) while BMDs only use the integer Davio decomposition $dt_{BMD}(x_i) := [+, \cdot, 1, x_i]$. Note that all graph types of figure 1 are based on Kronecker decompositions.

While the decomposition type is used to describe the function of nodes, attributed edges are often used to modify the function represented at the edges of the graph.

Minato ((Sasao 1996), pg.10) describes a general method to define an attributed edge. It is based on mappings $m_k : \mathcal{F} \rightarrow \mathcal{F}$ where $\mathcal{F}$ is the set of functions to be represented:

1. Partition $\mathcal{F}$ into a number of subsets $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_n$.
2. For any $1 \leq k \leq n$, define a mapping $m_k : (\mathcal{F} \rightarrow \mathcal{F})$, such that for any function $f_k \in \mathcal{F}_k$ there is a unique $f_0 \in \mathcal{F}_0$ to satisfy $f_k = m_k(f_0)$.

Consequently, an edge label m_k modifies the function represented at a node into the function $f_{edge} = m_k(f_{node})$.

Let $\mathcal{M}$ be the set of possible mappings m_k supported by the graph representation. Based on the definitions for the set $\mathcal{M}$ and decomposition type list

dtl, the graph type τ can be defined as the pair

$$\tau := [\mathcal{M}, dtl]. \tag{2}$$

For example, the graph type BDD is easily described by the pair

$$\tau_{BDD} = [\{\overline{f}\}, \text{S-dtl}],$$

where S-dtl denotes a DTL with Shannon decompositions only.

Based on the definitions for the graph type, basic graph manipulation functions can be introduced.

The function $edge^\tau$ is used to construct graphs of type τ. It reduces the node consisting of variable x and the successors F^τ_{low}, F^τ_{high} and returns a labelled edge F^τ:

$$F^\tau = edge^\tau(x, F^\tau_{low}, F^\tau_{high}).$$

Other basic graph operations are functions to compute the representation of cofactors $F^\tau_{x=0}$, $F^\tau_{x=1}$ from the graph representation of F^τ by replacing variable x in F^τ with constant symbols. Finally, the inverse function $succ^\tau$ is used to obtain graph successors from cofactor representations:

$$[F^\tau_{low}, F^\tau_{high}] = succ^\tau(x, F^\tau_{x=0}, F^\tau_{x=1}).$$

Note that functions $edge^\tau$ and $succ^\tau$ as well as the cofactor functions only depend on a single graph type τ. Their implementation is discussed in the original work for the respective graph type and is typically very simple. For example, the successor function for BDDs uses identity to obtain the pair $[F^\tau_{low}, F^\tau_{high}]$ from $[F^\tau_{x=0}, F^\tau_{x=1}]$.

2.2 Hybrid Graph Algorithms

Manipulation of decision diagrams is often based on depth-first algorithms. An important class of these algorithms are so-called Apply algorithms which proceed by recursively applying an operation to graph predecessor functions.

This section describes the core of the presented package, which is based on three different flavours of Apply algorithms.

Virtually any graph operation is implemented as a recursive function that consists of a termination test and a recursive part (the Apply step). Whenever only a single graph type is involved, algorithms from the original work can be used. Otherwise, arguments of an operation are expanded recursively until a return value is obtained. If the graph type of the result is different from the

graph type(s) of the arguments, the function expansion is continued until the return value is a constant function.

The most general (functional) approach to evaluate an operation is to follow a domain-partitioning paradigm. This involves computing orthogonal expansions for arguments and result and than convert the result to the desired decomposition type, i.e. use function $succ^\tau$ to obtain graph successor functions.

Theorem 1 (Functional Apply) *Any binary operation* $\circ$ *can be evaluated based on orthogonal (Shannon type) expansions of argument functions:*

$$\begin{aligned} r &= f \circ g \\ &= x * (f_{x=1} \circ g_{x=1}) + (\neg x) * (f_{x=0} \circ g_{x=0}) \end{aligned}$$

Functions $f_{x=0}$, $f_{x=1}$ and $g_{x=0}$, $g_{x=1}$ are the cofactors of f and g, respectively. $\neg x$ denotes the complement of boolean variable x, i.e. $\neg x = \overline{x}$ if r is a boolean function and $\neg x = (1 - x)$ if the operation returns an integer result.

Figure 2 outlines the Apply step based on theorem 1. The recursive function computes the graph representation R^τ of function r with graph type τ. F^ϕ, G^ψ are graph representations of type ϕ and ψ for functions f and g, respectively. $\bullet$ denotes an arbitrary binary graph operation. The algorithm assumes, that

```
1  funct functional_apply(•, τ, F^φ, G^ψ) ≡
2    comment: computes R^τ = F^φ • G^ψ
3    ⌈if (lookup_CT([•, τ, F^φ, G^ψ]) → SUCCESS)
4      then
5          R^τ = CT_entry_for([•, τ, F^φ, G^ψ]);
6      else
7          x = top_variable(F^φ, G^ψ);
8          R^τ_{x=0} = F^φ_{x=0} • G^ψ_{x=0};
9          R^τ_{x=1} = F^φ_{x=1} • G^ψ_{x=1};
10         [R^τ_low, R^τ_high] = succ^τ(x, R^τ_{x=0}, R^τ_{x=1});
11         R^τ = edge^τ(x, R^τ_low, R^τ_high);
12         insert_in_CT(R^τ, [•, τ, F^φ, G^ψ]); fi;
13   R^τ; ⌋.
```

Figure 2 Functional Apply Algorithm.

F^ϕ, G^ψ are graphs with proper current domain and that type τ can hold the result. In the implementation, these conditions can easily be checked before calling the operation.

Functional Apply is very similar to Apply algorithms known from literature but in addition it handles graph types. The algorithm first checks a cache (the so-called Computed-Table, CT) if a previous operation can be reused. Otherwise it selects the next variable x (found on top of either F^ϕ or G^ψ) and recursively evaluates the operation based on cofactoring. The final result is obtained through functions $succ^\tau$ and $edge^\tau$.

Please observe that any graph constructing sub-function of the functional Apply algorithm only depends on a single graph type. As a consequence, the algorithm can handle arbitrary types and new types can easily be added by providing the corresponding graph constructing functions.

While this algorithm is sufficient to compute any binary graph operation, it can be improved depending on operation $\bullet$. Simplified versions exist for addition and multiplication in the domain of function r. In both cases, computation of cofactors (lines 8,9 in figure 2) as well as computation of the successor function $succ^\tau$ (line 10) can be avoided.

Theorem 2 (Structural Apply for Addition) *If both argument graphs and the result use the same decomposition type for variable x, addition in the native domain of graphs can be simplified according to:*

$$\begin{aligned} f+g &= (d_{low}(x) * f_{low} + d_{high}(x) * f_{high}) + \\ &\quad (d_{low}(x) * g_{low} + d_{high}(x) * g_{high}) \\ &= d_{low}(x) * (f_{low} + g_{low}) + d_{high}(x) * (f_{high} + g_{high}). \end{aligned}$$

Note that neither the graph types used for f and g, nor the whole DTLs need to be the same.

As a special case of theorem 2, computation of function $succ^\tau$ can also be avoided if decomposition types of arguments can be converted before applying addition.

Example 1 *Consider addition of two functions f and g, where f is given as a moment decomposition ($\rightsquigarrow$ BMD), while g is represented using integer Shannon expansion ($\rightsquigarrow$ MTBDD). After changing the decomposition type of variable x in g, for the moment expanded result r one has*

$$\begin{aligned} r &= r_{low} + x \cdot r_{high} \\ &= [f_{low} + x \cdot f_{high}] + [x \cdot g_{high} + (1-x) \cdot g_{low}] \\ &= (f_{low} + g_{low}) + x \cdot [f_{high} + (g_{high} - g_{low})]. \end{aligned}$$

Furthermore, if function g is independent of variable x, conversion of the

decomposition type can be skipped since $g_{high} = g_{low} = g$. In this case, theorem 2 can be applied, even if the native domain of g differs from the native domain of r. An important application of this observation is addition and subtraction of a boolean graph type and an integer graph type (s.a. section 5.2).

Similarly, there is a simplification for multiplication if one argument is independent of variable x:

Theorem 3 (Structural Apply for Multiplication) *Multiplication in the native domain of the result graph can be simplified, if at least one argument has the same native domain as the result and if the other argument is independent of variable x:*

$$\begin{aligned} f * c &= (d_{low}(x) * f_{low} + d_{high}(x) * f_{high}) * c \\ &= d_{low}(x) * (f_{low} * c) + d_{high}(x) * (f_{high} * c) \end{aligned}$$

Thus, multiplication of Kronecker functions is an 'easy' operation, if both functions depend on disjoint sets of variables.

As demonstrated in example 1, the decomposition type of f can be changed if it doesn't match the decomposition type used in the result graph.

Please observe, that both structural Apply algorithms and their variants for the special cases are advantageous to functional Apply. For all structural Apply algorithms, at least the computation of function $succ^{\tau}$ is avoided. In some cases, decomposition types must be changed instead of computing cofactors. On the assumption that the target graph type is adequate, arguments of structural Apply are converted to the decomposition type of the result in an early stage of the traversal.

On the contrary, functional Apply is restricted to orthogonal (Shannon-type) expansions for the graph traversal phase, while different decompositions might be used in the graph construction phase. Since exponential gaps exist between graph types using different expansions, functional Apply is potentially an exponential operation.

3 MULTIPLE GRAPH TYPES AND VARIABLE REORDERING

DD packages maintaining strong canonical forms are restricted to a total ordering of input variables. An important feature of a DD package is therefore the ability to dynamically change variable ordering during or in-between operations without user intervention. Variable sifting has been shown to be very effective for BDDs (Rudell 1993) and has been extended to OKFDDs (Drechsler *et al.* 1995).

However, not all graph representations can be sifted. From the graph types of figure 1, *BMDs and K*BMDs are such a case. A sufficient condition to

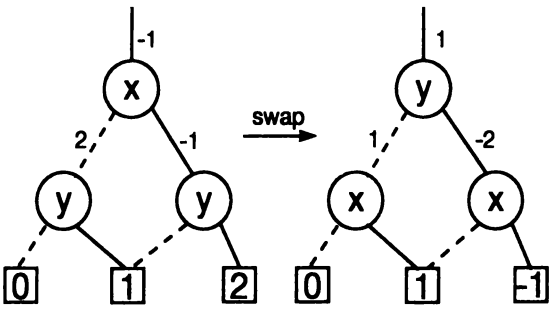

Figure 3 *BMD for $f = x - 2y + 2xy$

make dynamic reordering an automated (background) process is that incoming edges to a sifted node must not change during variable swapping. If this condition holds, the users application can safely copy and manipulate graph pointers (edges) without interfering with the reordering mechanism.

For *BMDs, a function obtains its root label by computing the greatest common divisor (*gcd*) of the root labels of successor functions. Conversely, the function represented at an edge is obtained by multiplying the edge value m with the node function:

$$f = m \cdot (\frac{f_{low}}{m} + x \cdot \frac{f_{high}}{m}).$$

Starting from terminal values, the root label is obtained in a bottom-up manner. Since for *BMDs hold $f_{low} = f_{x=0}$ and $f_{high} = f_{x=1} - f_{x=0}$, and

$$\begin{aligned} m &= gcd(f_{low}, f_{high}) \\ &= gcd(f_{x=0}, f_{x=1} - f_{x=0}) \\ &= gcd(f_{x=0}, f_{x=1}) \end{aligned}$$

the root label of a *BMD can also be obtained from the Shannon expanded tree representation of the graph. As a consequence, the root label of a *BMD is simply the greatest common divisor of all function values. Its absolute value is independent of variable ordering and will not change during sifting.

However, the sign of the root label is obtained from the sign of the successor in the low direction. If f_{low} is the constant function zero, the sign is obtained from f_{high}. Since f_{low} might change during sifting, the sign of the root label will probably change as well.

Example 2 *In Figure 3 two *BMDs for function $f = x - 2y + 2xy$ under different variable orderings are given. As can easily be seen the root node of the two *BMDs differ, but the ordering of the left *BMD can be transformed to the ordering of the right *BMD by only swapping one pair of neighbour-*

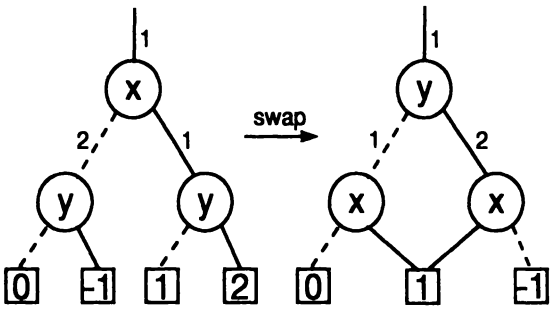

Figure 4 p*BMD for $f = x - 2y + 2xy$

*ing variables. Thus, dynamic variable reordering is not a local operation for *BMDs.*

To overcome this limitation, it is sufficient to restrict root labels to positive values. By this modification, the size of the resulting graph is doubled at most. Multiplying a positive *BMD (p*BMD) with a positive number is still a constant-time operation, but multiplication with negative numbers additionally requires to copy the graph and to negate the terminal values.

Example 3 *The p*BMDs for the function from Example 2 are given in Figure 4. The value at the root node does not change by an exchange of the variables.*

4 DATA STRUCTURES

In the previous sections it has been shown, that hybrid graph manipulation can be based on a small number of Apply algorithms. Well-known functionality like sifting can be supported even in the presence of multiple graph types. It turns out that also memory requirements for graph representation are nearly the same as for conventional DD packages.

In the package implementation two byte per node are spent to represent the additional information. The decomposition type and the type of edge labels are kept in one byte each, limiting the maximum number of supported label types and decomposition types very liberally.

Since the same data structures are used for all graphs, the size of an edge depends on the most complex graph type being configured. Currently, up to two edge weights are supported increasing the size of an edge from 4 byte to either 8 byte for arithmetic DDs or 12 byte for K*BMDs. Depending on the value of two lower bits, an edge weight can either be interpreted as the address of a multiple precision number or as an 30 bit integer value; the remaining bit is used for the sign. Constant functions are represented as a labelled edge to

a single terminal node. All together, node requirements are either 20, 28 or 36 byte including all memory management overhead.

5 APPLICATIONS AND EXPERIMENTS

Since the presented approach is a true superset of any approach based on Kronecker decision diagrams, applications of the package exist virtually in any environment where decision diagrams can be applied. This section demonstrates its usefulness for manipulation of bit-level and word-level functions and for bit selection from arithmetic expressions. Experiments have been made on a Sun UltraSPARC-170 workstation.

5.1 Control and Data Variables

It is well known, that orthogonal (BDD-like) expansions are very efficient for the representation of control logic while moment (*BMD-like) decompositions are often effective for representing data path logic at the word-level.

Hybrid Decision Diagrams are a means to represent arithmetic expressions that contain control and data variables. As a prerequisite, the variable space has to be partitioned and decomposition types are used according to the functionality of the individual variable.

The approach presented in this paper naturally supports representation and manipulation of those expressions even if variables are both part of the control logic and part of the data path of a circuit.

Example 4 *In figure 5 the Hybrid Decision Diagram for the expression* **if** $(a[0] \oplus a[1])$ **then** $b[0:1]$ *and for the multiplier function* $a[0:1] * b[0:1]$ *are shown. The HDD for the conditional expression uses moment decompositions (pD) to represent the word encoded vector b and integer Shannon decompositions (S) to represent the conditional part. In a conventional approach, this would collide with the representation of the multiplier function which also depends on variables* $a[0], a[1]$. *In the approach presented in this paper, this is easily solved by using graphs defined over appropriate DTLs. Formally, both decision diagrams are treated as different graph types.*

5.2 Bit-Level and Word-Level Operations

In a first experiment, the word-level representation of a multiple-output function is computed starting from different graph representations of bit-level logic. Column Bit-level compares the efficiency of BDDs, FDDs, MTB-

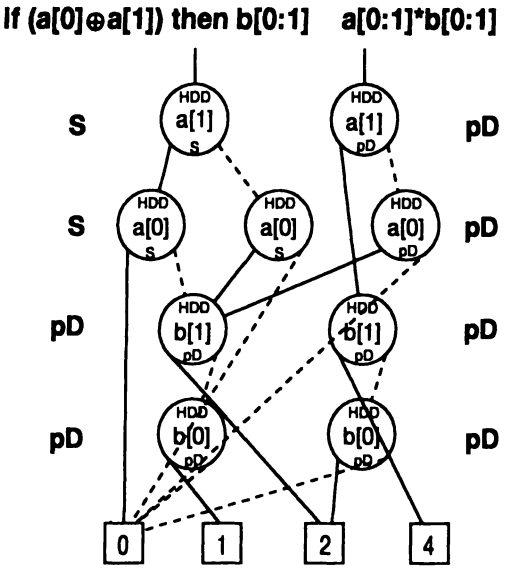

Figure 5 Multiple Decomposition Types for Individual Variables.

DDs, *BMDs and p*BMDs for representing bit-level output functions. Based on these differing bit-level representations, the unsigned integer encoding $o = \sum_i 2^i \cdot o_i$ has been computed from the vector of output functions o_i. The arithmetic function o is represented using MTBDDs, *BMDs and p*BMDs. Columns time and size depict construction time and final graph size of bit-level and word-level expressions, respectively. Machine limits have been set to 128 MByte and 1 CPU hour.

Table 1 shows examples with different runtime and space requirements. As expected, graph types differ significantly for their word-level and bit-level behaviour. While MTBDD construction at the word-level is often very fast, *BMDs and p*BMDs tend to be more compact at reasonable runtimes. The situation is often reversed at the bit-level, where orthogonal expansions are advantageous.

It is interesting to see, that in all of the examples a combination of different graph types obtains the best results for runtime and/or graph size.

5.3 Bit Selection

Bit selection is another example where multiple graph types are useful. Suppose we want to select the least significant bit (LSB) from the BMD representation of an arithmetic function f. If f is given as an unsigned integer representation, selecting the LSB is equivalent to computing the parity of f:

$$odd(f) \quad = \quad odd(f_{low} + x \cdot f_{high})$$

Circuit	in	out	Bit-level		MTBDD		Word-level *BMD		p*BMD	
mul8	16	16	time	size	time	size	time	size	time	size
BDD			1.7s	9257	8.1s	65280	40.4s	16	39.0s	16
FDD			17.9s	16588	13.4s		47.5s		45.3s	
MTBDD			3.6s	11033	8.1s		31.4s		31.6s	
*BMD			138.2s	51571	39.4s		6.1s		13.1s	
p*BMD			124.7s	56314	33.0s		12.1s		6.0s	
C880	60	26	time	size	time	size	time	size	time	size
BDD			0.9s	8654	$> 128M$		319.9s	57634	63.2s	113162
FDD			6.9s	27326			319.9s		58.1s	
MTBDD			1.8s	8676			364.3s		167.0s	
*BMD			352.7s	51591			171.0s		481.5s	
p*BMD			139.3s	99488			174.9s		16.2s	
too_large	38	3	time	size	time	size	time	size	time	size
BDD			19.2s	7082	0.7s	14122	27.8s	11126	12.8s	22241
FDD			18m10s	13926	3.1s		31.6s		17.1s	
MTBDD			58.81s	7158	0.6s		47.9s		31.2s	
*BMD			$> 1h$							
p*BMD			$> 1h$							
div8	16	9	time	size	time	size	time	size	time	size
BDD			2.1s	3195	0.5s	4981	4.6s	7263	4.4s	9012
FDD			20.34s	3062	1.1s		5.4s		5.4s	
MTBDD			3.72s	3295	0.5s		2.2s		2.2s	
*BMD			99.2s	7244	4.1s		0.5s		1.6s	
p*BMD			118.6s	9193	4.5s		1.2s		0.6s	

Table 1 Multiple graph types in bit- and word-level operations.

Since the parity of a sum is equal to the exclusive-or of the parities, one has:

$$odd(f) = odd(f_{low}) \oplus odd(x \cdot f_{high}) \tag{3}$$

With $x \in \mathbf{B}$, $odd(x \cdot f_{high}) = x \wedge odd(f_{high})$:

$$odd(f) = odd(f_{low}) \oplus (x \wedge odd(f_{high})).$$

This is already the Reed-Muller decomposition of function $odd(f)$. Consequently, the FDD representation of $odd(f)$ is obtained by (structurally) traversing the BMD for f until the parity of a terminal case (i.e. the LSB of an integer number) can be returned. After reducing the graph, the result cannot be larger than the BMD for f. Typically the FDD will be much smaller, since the number of different leaf values has been reduced.

Arditi (1996) defines the *odd* function for *BMDs. For arithmetic graph types, the exclusive-or in equation (3) can be replaced according to $a \oplus b =$

$a + b - 2 \cdot a \cdot b$ (cf. Chen *et al.* (1994)).

$$\begin{aligned} odd(f) &= odd(f_{low}) + odd(x \cdot f_{high}) - 2 \cdot odd(f_{low}) \cdot odd(x \cdot f_{high}) \\ &= odd(f_{low}) + x \cdot [odd(f_{high}) - 2 \cdot odd(f_{low}) \cdot odd(f_{high})] \end{aligned}$$

Therefore, computing the *BMD of $odd(f)$ requires the multiplication $odd(f_{low}) \cdot odd(f_{high})$. Since multiplication of *BMDs can be an exponential operation, time and/or space requirements to compute $odd(f)$ can be exponentially large as well.

Table 2 summarizes the results for bit selection from arithmetic expressions.

For the presented examples and based on the multiplication algorithm proposed in (Bryant *et al.* 1995) we have not been able to get beyond word sizes of 64 bit. The time requirements for bit selection grow exponentially with the word size. Using the Apply algorithms presented in this paper, word sizes of up to 256 bit could easily be handled.

The following program has been executed to successively select bits of the arithmetic expressions $msbs_0 = a + b$ and $msbs_0 = a \cdot b$ (a, b are n-bit vectors given as an unsigned integer encoding):

$$\begin{array}{l} \lceil \underline{\textbf{for}}\ i = 0\ \underline{\textbf{to}}\ n\ \underline{\textbf{do}} \\ \quad lsb_i = odd(msbs_i); \\ \quad msbs_{i+1} = (msbs_i - lsb_i)/2;\ \underline{\textbf{od}}; \rfloor \end{array}$$

Table 2 compares results for representing bits lsb_i with BDDs, FDDs, BMDs and *BMDs. All functions $msbs_i$ are represented with *BMDs. The subtraction $msbs_i - lsb_i$ is a hybrid operation between the different graph types used in the bit level and in the word level representation.

Columns *lsb* show the total time for bit selection as well as the maximum size of a single bit. Columns *msbs* present time and space requirements for computing the (word encoded) remaining bits. Since, in the example, division by two is a constant time operation, nearly all of the *msbs* time is spent for the subtraction of the word-level *BMD and the bit-level graph type in the corresponding row.

The examples show, that the hybrid approach clearly outperforms methods based on a single graph type. As expected, FDDs are ideally suited for bit selection from *BMDs. If the size of the selected bit is small compared to the size of a BMD, this approach even outperforms *BMD/*BMD subtraction, which is implicitly based on a BMD expansion.

a + b		32		64		128		256	
		lsb	*msbs*	*lsb*	*msbs*	*lsb*	*msbs*	*lsb*	*msbs*
BDD	time	0.25s	0.6s	1.2s	2.7s	7.5s	12.2s	52.0s	52.4s
	size	95	126	191	254	383	510	767	1022
FDD	time	0.11s	0.8s	0.4s	3.3s	1.7s	13.9s	7.1s	61.7s
	size	95	126	191	254	383	510	767	1022
BMD	time	9.9s	0.7s	327.8s	5.6s	$> 1h$		$> 1h$	
	size	1521	126	6113	254				
*BMD	time	1.15s	0.6s	7.9s	5.1s	65.2s	49.1s	584.8s	490.7s
	size	126	126	254	254	510	510	1022	1022

a * b		7		8		9		10	
		lsb	*msbs*	*lsb*	*msbs*	*lsb*	*msbs*	*lsb*	*msbs*
BDD	time	0.23s	4.6s	0.9s	21.5s	3.3s	93.8s	13.3s	410.1s
	size	744	3490	1774	14157	4221	56653	9955	228135
FDD	time	0.07s	4.9s	0.2s	21.9s	0.8s	98.6s	3.5s	432.9s
	size	602	3490	1842	14157	5958	56653	20049	228135
BMD	time	3.6s	0.3s	21.6s	1.4s	120.4s	6.6s	625.5s	32.9s
	size	4183	3490	16284	14157	63825	56653	250232	228135
*BMD	time	4.7s	0.3s	28.9s	1.2s	164.3s	5.8s	899.7s	26.2s
	size	3301	3490	13186	14157	52773	56653	211047	228135

Table 2 Bit selection and hybrid operations.

6 CONCLUSION

Theoretical results support the conjecture, that there exist no single graph type that can represent boolean and arithmetic functions with the same efficiency. Therefore, it is important to provide a framework for the integration of multiple graph types.

This paper presented an implementation of a multiple domain graph manipulation package that integrates a variety of decision diagrams. Hybrid manipulation algorithms have been given, so that type conversion of graphs is an implicit operation. New graph types can easily be added to this concept if they are based on Kronecker decompositions.

The package implementation has been shown to be very effective in applications that manipulate bit-level functions as well as word-level expressions. From a theoretical point of view, the presented approach overcomes many exponential gaps that exist between different graph types and between graph types using different DTLs.

Furthermore, this paper introduced dynamic variable reordering for *BMDs, that is the most promising minimization technique for decision diagrams. Dynamic reordering is supported even if multiple graph types are used together.

There are a number of applications where the presented work is also very promising. Current research concentrates on verification of assembly instructions of microprocessors.

REFERENCES

L. Arditi. *BMDs can delay the use of theorem proving for verifying arithmetic assembly instructions. *Formal Methods in Computer-Aided Design*, pages 34–48, November 1996.

B. Becker and R. Drechsler. How many decomposition types do we need? *European Design and Test Conference*, pages 438–443, 1995.

B. Becker, R. Drechsler, and R. Enders. On the representational power of bit-level and word-level decision diagrams. *Asian and South Pacific Design Automation Conference*, 1997.

R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C(35):677–691, August 1986.

R. E. Bryant and Y.-A. Chen. Verification of arithmethic functions with binary moment diagrams. Technical Report CMU-CS-94-160, Carnegie Mellon University, May 1994.

R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. *ACM/IEEE Design Automation Conference*, pages 535–541, 1995.

E. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams - overcoming the limitations of MTBDDs and BMDs. *International Conference on Computer Aided Design*, November 1995.

E. Clarke, K. L. McMillan, X. Zhao, and J. C.-Y. Yang. Spectral transforms for large Boolean functions with application to technology mapping. *ACM/IEEE Design Automation Conference*, pages 54–60, June 1993.

E. Clarke, M. Kaira, and X. Zhao. Word level model checking – avoiding the pentium fdiv error. *ACM/IEEE Design Automation Conference*, pages 645–648, June 1996.

R. Drechsler and B. Becker. Dynamic minimization of OKFDDs. In *International Conference on Computer Design*, pages 602–607, 1995.

R. Drechsler, B. Becker, and S. Ruppertz. K*BMDs a new data structure for verification. *European Design and Test Conference*, 1996.

R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams. *ACM/IEEE Design Automation Conference*, pages 415–419, 1994.

N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. *International Conference on Computer-Aided Design*, pages 472–475, 1991.

U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. *European Conference on Design Automation*, pages 43–47, 1992.

Y.T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. *ACM/IEEE Design Automation Conference*, pages 608–613, June 1992.

S. Minato. Zero-suppressed BDDs for set manipulation in combinational problems. *ACM/IEEE Design Automation Conference*, pages 272–277, 1993.

R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. *International Conference on Computer Aided Design*, pages 42–47, 1993.

T. Sasao and M. Fujita (Editors). *Representations of Discrete Functions*. Kluwer Academic Publishers, 1996.