Abstract—We investigate the properties of the Ordered Ternary Decision Diagram (OTDD) in order to develop an efficient general OTDD package. The OTDD is a three-branched three-terminal diagram based on Kleenean strong ternary logic. The OTDD can represent functions having nontrivial don’t-care sets in a single diagram and is capable of provably correct evaluation in the presence of unknown input values. We propose a number of improvements to both OTDD computational methods and data structures. Furthermore we introduce the purged form OTDD which unifies the abbreviated and full form OTDD into a single diagram. A package exploiting these OTDD specific properties is presented and we show the computational advantages of this improved package for LGSynth93 standard benchmarks.

I. INTRODUCTION

In computer engineering applications we are often confronted with complex Boolean functions containing nontrivial don’t-care sets. Furthermore to accurately and efficiently handle undefined logic input values is another problem related to logic simulation applications, synthesis and testability, see [JIL94], [CP89] and [CA87]. The Ordered Ternary Decision Diagram (OTDD) as proposed in [Jen95] solves the accurate evaluation problem in the presence of unknown inputs, and is capable of representing functions having nontrivial don’t-care sets in a single diagram. The OTDD is based on Kleenean strong ternary logic permitting well defined logical operations on OTDD functions. For any number of undefined input logic values the output can be evaluated in $O(n)$ time, where $n$ is the number of inputs. In [Jen95] the basic algorithms for building and processing OTDDs are defined. However, efficiency and optimization of the actual implementation are only briefly discussed.

This paper discusses our experience from implementing a general OTDD package. Based on our observations we propose a number of improvements to the basic computational methods proposed in [Jen95]. We introduce the purged form OTDD which supports lazy evaluation of the OTDD. To evaluate our refinements of the original OTDD computational methods, a number of experiments were performed on standard benchmarks (LGSynth93). The experiments involve both building the OTDDs from the PLA-descriptions in LGSynth93 and performing logical operations on them. Our results show that the proposed improvements reduce memory requirements and increase computational speed.

The paper is organized as follows: Section II gives a short tutorial on the basic properties of the OTDD. In Section III we discuss the properties of the basic OTDD package, and propose a number of improvements exploiting OTDD specific properties. Section IV motivates our choice of experiments and contains a detailed description of these. Section V shows and discusses the results obtained from computations run on a Sun SPARC ELC.

II. A SHORT TUTORIAL ON THE OTDD

This section gives a short tutorial on the OTDD. For an in depth description we refer the reader to [Jen95] where the theoretical aspects of the OTDD are discussed.

The OTDD was introduced to satisfy the following criteria. First: representing functions having nontrivial don’t-care sets in a single diagram; second: allowing evaluation of these functions even when given (any number of) undefined logic input values; and third: allowing operations to be performed on the functions. The OTDD is an ordered decision diagram with terminal values $\{0, 1, U\}$, where $U$ represents a don’t-care output value corresponding to the first criterion. From the second criterion follows that any nonterminal must be able to handle an undefined logic input value ‘$U$’. This problem was solved by giving the nonterminals children ‘low’, ‘high’ and ‘und’ corresponding to input values $\{0, 1, U\}$.

Removal of all nonterminals where ‘low’ = ‘high’ yields a reduced form OTDD. If ‘und’ is removed from all nonterminals an abbreviated form OTDD is obtained. Although the resulting abbreviated OTDD can not directly handle ‘$U$’ input values, all necessary information to retrieve a full form OTDD is preserved. The last crite-
tion, to allow operations on the functions represented as OTDDs, is addressed by using the semantics of Kleenean strong logic for evaluation. The abbreviated form is suitable for representing intermediate results during calculation on OTDDs, since it requires less space and computations to obtain. Note that the abbreviated form OTDD is still capable of representing functions having nontrivial don’t-care sets.

Figure 1 shows the identity function for variable $x_1$ (to the left) and the function $((x_1 \land x_2) \lor x_4)$ as a full form OTDD (to the right). Evaluation is done by traversing the OTDD from the root, for example, the input vector $x_1 = 'U'$; $x_2 = 'U'$; $x_4 = '1'$ corresponding to ("U" and "U") or "1"), traverses $x_1$, $x_2$, $x_4$ and finally arrives at the terminal "1". Though the input vector contains undefined logic values, the output will be correctly evaluated.

III. OTDD Computational Methods

We started out by implementing a basic OTDD package as described in [Jen95]. From this exercise we gained insight into some of the properties unique to the OTDD. Exploitation of these properties resulted in the optimizations proposed in the following sections.

A. The Basic OTDD Package

Our first OTDD package implemented the computational methods suggested in [Jen95]. The package includes Kleenean AND, OR, XOR and COMPOSE as well as the OTDD specific operations OR_B_TO_U, TERNATE and ALIGNMENT. The COMPOSE operation $G_{k,F}$ replaces the input variable $v_i$ of function $G$ with the function $F$, and yields a single (full form) OTDD. The operation OR_B_TO_U is used to create an abbreviated form OTDD from ON and DC set OTDDs. The TERNATE operator transforms an abbreviated form OTDD into a full form OTDD by recursively visiting each node, and deriving their ‘und’ branches by applying ALIGNMENT on that node’s ‘low’ and ‘high’ branches. We refer the reader to [Jen95] for detailed description and pseudocode for these operations.

The node description used in our basic package can be found in Figure 2. The ‘index’ field indicates to which input variable this specific node applies. Note that value ‘X’ is given to all nonterminals and that the value ‘U’ indicates a don’t-care terminal. All nodes are cached in a unique-table as described in [BR90].

To perform operations on the OTDDs, the $\text{Apply}_\text{step}$ routine first described by [Bry86] and adapted for the OTDD in [Jen95] is used. $\text{Apply}_\text{step}$ is called recursively, traversing the operands towards their terminals. $\text{Apply}_\text{step}$ uses an “Op” function, which given the ‘val’ fields of the top node operands, can determine terminal and controlling cases, the latter preventing further traversal. In [Jen95] a flag is used to indicate whether $\text{Apply}_\text{step}$ will calculate a full or an abbreviated form OTDD. In our package $\text{Apply}_\text{step}$ yields an abbreviated form OTDD, a full form OTDD can be derived post facto by an explicit TERNATE operation. A memory function proposed in [BR90] insures that each unique operation is only performed once.

B. Refinement of the “Op” Function

The “Op” function in our basic package forces evaluation of the operands all the way down to the terminals if no controlling cases are found. We seek a way to prevent that kind of exhaustive evaluation.

Under Kleenean logic $(f \land f) = f$, which intrigues us to consider isomorphic operands. In these cases we might be able to prevent further traversal of the operands.

The OTDD has a strong canonical form, proven in [Jen95]. This together with the unique-table representation of OTDDs allows us to detect isomorphism between OTDDs as pointer equality, that is, in constant time.

Based on these observations an improved “Op” function exploiting isomorphism is proposed in [Jen95]. The modified “Op” function is shown in pseudocode in Figure 3. The $\text{iso}$ flag indicates that the two operands are isomorphic. The OR operation is modified to handle $(f \lor f) = f$ by adding $(\text{iso}) \Rightarrow \text{return} F$, the AND operation is modified to handle $(f \land f) = f$ in the same way. The
char Op(a, b, opcode, iso) {
    (opcode == OR):  
    (a == '1') || (b == '1') => return '1' else return '0';
    (a == 'X') || (b == 'X'):
    (iso)  => return 'F' else return '0';
    (a == '0') => return '0' else return 'P';
    (b == '0') => return 'P' else return 'F';
    return 'X';
    (a == '0') && (b == '0') => return '0' else return 'U';
    (a == '1') && (b == '1') => return 'U' else return '0';
    (a == '1') && (b == '0') => return 'U' else return '0';
    (b == '1') && (a == '0') => return 'U' else return '0';
    return 'X';
    (a == '1') && (b == '1') => return 'U' else return '0';
    (a == '0') && (b == '0') => return 'U' else return '0';
    return 'X';
    (a == 'X') || (b == 'X'):
    (iso)  => return 'F' else return '0';
    (a == '0') => return '0' else return 'P';
    (b == '0') => return 'P' else return 'F';
    return 'X';
    (a == '1') && (b == '1') => return 'U' else return '0';
    (a == '0') && (b == '0') => return 'U' else return '0';
    return 'X';
    (a == 'X') || (b == 'X'):
    (iso)  => return 'F' else return '0';
    (a == '0') => return '0' else return 'P';
    (b == '0') => return 'P' else return 'F';
    return 'X';
    (a == 'b')  => return a else return 'U';
}

Fig. 3. The improved “Op” function, [iso] indicates that the operands are isomorphic.

ALIGNMENT operator (used to create the ‘und’ branch), conforms to \( f(\text{alignment}\ f) = f \), see [Jen95]. This allows the same modification of the ALIGNMENT operation as described for the OR operation.

Under Kleenean-strong logic the XOR operation does not conform to \( f(xor\ f) = 0 \), hence adding \( (iso) => return 'U' \) would violate the Kleenean semantics.

C. The Purged form OTDD

At this point we made two critical observations.

First observation: the OTDD is canonical with respect only to the ‘index’ of the input variable and its ‘low’ and ‘high’ branches.

Second observation: the ‘und’ branch can be derived for any node at any time by computing the ALIGNMENT of that node’s ‘low’ and ‘high’ branches.

The first observation immediately leads to the conclusion that the very same node pointed out in the unique-table can hold both the abbreviated and full form OTDD. We introduce the purged form OTDD allowing any mix of abbreviated and full form OTDD nodes in a single diagram. By using this new form we hope to reduce both the number of unique OTDD nodes allocated, and the size of the unique-table. The reader must notice that although a node might contain a defined ‘und’ pointer it is not to be considered as a full form OTDD, because the ‘low’ and ‘high’ branches are in no way guaranteed to be of full form. Note that the ALIGNMENT operation will effect all OTDDs that contain the aligned node.

To implement the purged form OTDD some modifications of the basic package are necessary. We propose a modified “uniqueness” criterion to be used for the unique-table (involving only ‘index’, ‘low’ and ‘high’ branches, opposed to the criteria used in [Jen95] which involves the ‘und’ branch). Furthermore we propose a “ut_insert” function, see [Jen95], which provides ‘und’ branch updating of existing abbreviated OTDD nodes. These modifications cause Apply_step to produce purged form OTDDs rather than abbreviated OTDDs.

The second observation is to some extent exploited in [Jen95]. The Apply_step function, as proposed in [Jen95], takes a parameter that indicates if the result is to be calculated as an abbreviated or a full form OTDD. Intermediate calculations can be performed producing abbreviated OTDDs (which require less memory and CPU resources) and the final result can be expanded to the full form OTDD. However Apply_step calculates either a full form OTDD or else a fully abbreviated OTDD.

Furthermore the second observation leads to the conclusion that the ALIGNMENT operation should not be issued until absolutely necessary, i.e. when required to perform an operation. If done earlier, ‘und’ nodes might be computed which are neither required for the computation, nor part of the final result. All such work is wasted!

The Compose algorithm, suggested in [Jen95], requires the downstream function to be represented as a full form OTDD (though Jennings briefly discusses the possibility of expanding the ‘und’ branches only when needed). This means we would expect significant improvements by using the new purged form OTDD for lazy evaluation.

As a result of our observations we propose an improved package including the new unique-table criteria, a node update function, and a new Compose algorithm, see Figure 4. The pseudo-code actually holds both the original compose_step as proposed in [Jen95], shown as (1), and our proposed optimized function, shown as (2). Observe that (1) excludes lines corresponding to (2) and vice versa. The optimization can be broken down in several steps. To prevent unnecessary computation of ‘und’ branches we allow both functions to be represented in purged form, relaxing the full form assumption in [Jen95]. If and only if the ‘und’ branch ‘vul’ is required, see (1b), we derive it from ‘vhl’ and ‘vh1’ by applying ALIGNMENT, see (2a). This operation derives ‘vul’ as...
node *compose_step(v1l, vhi, vul, v2) {
    /* v1[m] = v2 */
    (v1l.index == m) => v1l = v1l.low
    (vhi.index == m) => vhi = vhi.high
  1a) (vul.index == m) => vul = vul.und

    /* terminal in v1 */
    (v2.val == '0') and (v1l.val terminal) => return(v1l.val)
    (v2.val == '1') and (v1l.val terminal) => return(v1l.val)
  1b) (v2.val == 'U') and (v1l.val terminal) => return(v1l.val)
  2a) (v2.val == 'V') => (v = apply_step(v1l,vhi,ALIGNMENT) =>
    and (v1l.terminal) => return(vul.val))

    u.index = Min(v1l.index,vhi.index,vul.index,v2.index)
    (v2.index == u.index) => (v2low == v2.low; v2high == v2.high)
    else => (v2low = v2; v2high = '2')

    (v1l.index == u.index) => (v1l = v1l.low; v1hi = v1l.high)
    else => (v1l = v1l; v1hi = v1l)

    (vhi.index == u.index) => (vhi = vhi.low; vhi = vhi.high)
    else => (vhi = vhi; vhi = vhi)
  1a) (vul.index == u.index) => (vul = vul.low; vul = vul.high)
    else => (vul = vul; vul = vul)

    u.low = compose_step(v1l, vhi, vul, v2low)
    u.high = compose_step(v1l, vhi, vul, v2high)
    r = st_insert(u.index, u.low, u.high)
  1c) r.und = apply(u.low, u.high, ALIGNMENT, FULL)
    return (r);
}
node *Compose(v1, v2) {
    return compose_step(v1, v1, v1, v2);
}

Fig. 4. The optimized compose_step function. (1) shows the original function, (2) shows the optimized function.

a **purged** form OTDD, which is exactly what we need at that time.

Furthermore we benefit from neglecting the incoming ‘vul’ branch and from avoiding unnecessary calculations of ‘vul’ and ‘vulh’ by eliminating (1a). The second observation is further exploited by eliminating (1c), thus the result of compose_step will be a **purged** OTDD.

IV. **Description of the Experiment**

In order to measure the efficiency of our package with respect to memory usage, CPU usage etc we used test cases consisting of PLA-function descriptions from LGSynth93. The PLA descriptions contain both the ON set which evaluates to logic one and the explicit DC set (if any) which evaluates to logic ‘U’. The strong Kleenean ‘U’ means a valid logic one or logic zero, but we don’t know or don’t-care which.

A. **Building of the abbreviated form OTDDs**

During parsing we initially represent the ON and DC sets as OBDDs, see [Bry86]. Building the OBDDs involves both AND and OR operations. To represent both ON and DC set in one OTDD is done by applying the OR_{B_TO_U} operator to the ON and DC sets. The result of the OR_{B_TO_U} operation is an **abbreviated** form OTDD.

Several properties are measured during the procedure:

- Memory usage – the total number of nodes allocated
- Wasted nodes – the number of nodes allocated that are not part of the result
- CPU time used when performing logical operations

By this experiment we hope to gain insight in the performance of the basic OTDD and furthermore to evaluate the improved “Op” function.

B. **Expansion to full form OTDDs**

This is done by applying the unary operator TERMINATE, which recursively applies the ALIGNMENT operator to all nodes in the abbreviated form OTDD, the result is a **full** form OTDD. The ALIGNMENT operator processes the output of the ‘low’ and ‘high’ branches into the ‘und’ branch. We refer the reader to [Jen95] for further details.

The same measurements of memory usage and CPU load, as described above, are taken during the operations. By this experiment we seek further understanding of the properties regarding the basic and improved “Op” function.

C. **Retrieval of the full form OTDDs**

Now we can apply any logic function (AND, OR, NOT or XOR) to the OTDDs, and study the performance of our package. For this experiment we chose the NOT operator (implemented as F XOR ‘1’). We performed two consecutive NOT operations, thus yielding the original function.

We will be able to probe the package’s ability to retrieve a **full** OTDD from an **abbreviated** form OTDD in the case that TERMINATE already has been applied to the original function, see B.

D. **Composing OTDD functions**

Any two functions F and G represented as OTDDs can be composed, G|_{v_i=F}, where input variable v_i of downstream function G is replaced by the function F. Since the PLA descriptions in LGSynth93 correspond to functions F with different numbers of outputs F_i, we chose only to compose the first output F_1 with the first input of the downstream function, G|_{v_i=F_1}. Of course any other combination would do as well, but this choice serves our
purposes. The ability to handle upstream functions having nontrivial don’t-care sets is examined by PLAs ‘spla’, ‘miser3c’ and ‘ex1010’. This forces the proposed lazy compose_step algorithm to derive the ‘und’ branch while composing the functions, see (2a) in Figure 4. The input vectors to functions F and G can have any number of common variables. We test the ability to handle both completely disjoint sets of input variables as well as the case of maximum possible common variables.

V. RESULTS OF THE EXPERIMENTS

We present our refinements of the OTDD package from our first implementation towards the final version. To probe the efficiency gain of the separate improvements and how they affect each other our package allows us to turn the features on and off.

During the experiments several measurements were taken. Table I shows the number of nodes wasted when consecutively performing experiments IV A, B and C, as well as the total number of nodes allocated during the suite. When building the abbreviated form OTDD, see IV A, we typically get an improvement of more than 30% (peak at 75%) with respect to the number of wasted nodes. When expanding to full form, see IV B, the total improvement varies from 10% to 50%. After performing the two consecutive NOT operations, see IV C, the total improvement varies from 5% to 50%.

These results fulfill our expectations for the improved “Op” function. We successfully prevent exhaustive evaluation of the OTDDs by detecting isomorphic operands and utilizing the evaluation under Kleenean strong logic.

Table II compares the total CPU time when the experiments IV A, B and C are executed on a Sun SPARC ELC. The columns marked ‘lazy’ indicates that our unique criterion proposed in Section III C is used. The purged form OTDD increases computational speed by up to 30%.

From the results we conclude that the critical section is the building of the OTDDs, this result also applies to Table I. When the abbreviated OTDD is calculated once and for all, operations on the OTDDs are performed almost instantly. This proves the usability of the OTDD in practical applications.

Table III compares our improved Compose \((G|_{x,F})\) function to the basic implementation suggested in [Jen95]. The input vectors to functions F and G can have any number of common variables. For test purposes the input sets are either disjoint or have the maximum possible common variables, see column ‘disjoint inputs’. The next column ‘abbrev’ shows the number of nodes allocated to build the purged form OTDDs. The next two columns show the total number of nodes allocated and the number of ALIGNMENT entries in the memory function after performing the compose_step operation. The results after expanding the purged form OTDD to a full OTDD are shown in the next two columns. Finally the two rightmost columns show measurements taken on the basic OTDD package for comparison. Note that only full form OTDDs can be obtained from the original package.

When presented upstream functions having nontrivial don’t-care sets, our proposed compose_step derives the required ‘und’ branches under lazy evaluation. ‘alv4]=spla’ and ‘miser3c]=ex1010’ show that composition deriving the purged form only requires a fraction of the ALIGNMENT operations necessary for obtaining the full form. Our experimental results show that our proposed compose_step can derive a purged form OTDD mainly by using already allocated memory resources. The new compose_step compares favorably to the basic implementation even when expanding the purged form OTDD to full form. However, the essence of this paper is to show the advantages of using the purged form representation for all intermediate computations. The presented results verify this.

VI. CONCLUSIONS

We investigate the properties of The Ordered Ternary Decision Diagram (OTDD) in order to develop an efficient general OTDD package. We exploit OTDD specific properties into both improvements of the computational methods and data structures. We propose the purged form OTDD which allows any mixture of abbreviated and full form OTDD nodes in a single diagram, and can furthermore be utilized for lazy evaluation of the OTDD. A package exploiting these OTDD specific properties has been presented and we have demonstrated the computational advantages of this improved package on LG Synth93 standard benchmarks.

REFERENCES


### TABLE I

The number of wasted nodes (nodes allocated but not part of the result) during the experiments, see Section IV A, B and C. The total number of allocated nodes is found in the rightmost column.

<table>
<thead>
<tr>
<th>out</th>
<th>PLA → abbreviated OTDD</th>
<th>abbreviated → full OTDD</th>
<th>2*XOR</th>
<th>(\sum) nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>basic</td>
<td>“Op”</td>
<td>lazy</td>
<td>basic</td>
</tr>
<tr>
<td>9sym</td>
<td>9</td>
<td>2700</td>
<td>1765</td>
<td>2675</td>
</tr>
<tr>
<td>5xp</td>
<td>10</td>
<td>794</td>
<td>532</td>
<td>879</td>
</tr>
<tr>
<td>dule</td>
<td>29</td>
<td>6294</td>
<td>3921</td>
<td>19335</td>
</tr>
<tr>
<td>table5</td>
<td>15</td>
<td>6797</td>
<td>3867</td>
<td>28716</td>
</tr>
<tr>
<td>alu4</td>
<td>8</td>
<td>46810</td>
<td>34547</td>
<td>66979</td>
</tr>
<tr>
<td>spla</td>
<td>46</td>
<td>160711</td>
<td>134587</td>
<td>163435</td>
</tr>
<tr>
<td>misex &amp; ex1010</td>
<td>14</td>
<td>40861</td>
<td>29719</td>
<td>58231</td>
</tr>
<tr>
<td>ex1010</td>
<td>10</td>
<td>61930</td>
<td>57435</td>
<td>65134</td>
</tr>
</tbody>
</table>

### TABLE II

CPU times in seconds for consecutive experiments, see Section IV A, B and C, run on a Sun SPARC ELC.

<table>
<thead>
<tr>
<th>out</th>
<th>abbreviated</th>
<th>full OTDD</th>
<th>2*XOR</th>
<th>retrieve full</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>basic</td>
<td>“Op”</td>
<td>lazy</td>
<td>basic</td>
</tr>
<tr>
<td>9sym</td>
<td>1</td>
<td>0.16</td>
<td>0.17</td>
<td>0.15</td>
</tr>
<tr>
<td>5xp</td>
<td>7</td>
<td>0.06</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>dule</td>
<td>29</td>
<td>0.39</td>
<td>0.26</td>
<td>0.37</td>
</tr>
<tr>
<td>table5</td>
<td>15</td>
<td>0.62</td>
<td>0.63</td>
<td>0.57</td>
</tr>
<tr>
<td>alu4</td>
<td>8</td>
<td>2.64</td>
<td>2.07</td>
<td>2.36</td>
</tr>
<tr>
<td>spla</td>
<td>46</td>
<td>22.32</td>
<td>21.84</td>
<td>21.89</td>
</tr>
<tr>
<td>misex &amp; ex1010</td>
<td>14</td>
<td>2.20</td>
<td>2.05</td>
<td>2.65</td>
</tr>
</tbody>
</table>

### TABLE III

Composition of functions, see Section IV D, where ‘spla’, ‘misex3c’ and ‘ex1010’ have nontrivial don’t-care sets. ‘abbrev’ shows the number of nodes allocated to build the abbreviated form OTDDs. ‘alloc nodes’ shows the total number of nodes allocated, corresponding to each operation. ‘mem’a shows the number of ALIGNMENT entries in the memory function.

<table>
<thead>
<tr>
<th>disjoin</th>
<th>abbrev</th>
<th>purged compose</th>
<th>full compose</th>
<th>basic compose</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs</td>
<td>alloc nodes</td>
<td>mem</td>
<td>alloc nodes</td>
<td>mem</td>
</tr>
<tr>
<td>alu4[1] = 5xp</td>
<td>no</td>
<td>183</td>
<td>235</td>
<td>0</td>
</tr>
<tr>
<td>alu4[1] = spla</td>
<td>no</td>
<td>471</td>
<td>520</td>
<td>104</td>
</tr>
<tr>
<td>spla[1] = alu4</td>
<td>no</td>
<td>471</td>
<td>516</td>
<td>0</td>
</tr>
<tr>
<td>misex3c[1] = ex1010</td>
<td>no</td>
<td>10176</td>
<td>10424</td>
<td>413</td>
</tr>
<tr>
<td>alu4[1] = 5xp</td>
<td>yes</td>
<td>183</td>
<td>183</td>
<td>0</td>
</tr>
<tr>
<td>alu4[1] = spla</td>
<td>yes</td>
<td>471</td>
<td>471</td>
<td>18</td>
</tr>
<tr>
<td>spla[1] = alu4</td>
<td>yes</td>
<td>471</td>
<td>471</td>
<td>8</td>
</tr>
<tr>
<td>misex3c[1] = ex1010</td>
<td>yes</td>
<td>10176</td>
<td>10176</td>
<td>272</td>
</tr>
</tbody>
</table>