Ordered Ternary Decision Diagrams and the Multivalued Compiled Simulation of Unmapped Logic

Glenn Jennings, Joachim Isaksson and Per Lindgren
Division of Computer Engineering
Luleå Institute of Technology
S-971 87 Luleå, Sweden
glenn@sm.luth.se

Abstract

We describe a method for generating logic simulation code which correctly responds to any number of undefined logic values at the code inputs. The method is based on our development of the Ordered Ternary Decision Diagram, itself based on Kleenean ternary logic, which explicitly and correctly manages the unknown logic value 'U' in addition to the '1' and '0' of conventional OBDDs. We describe the OTDD and how to implement its reduction, application, and restriction operations. This method avoids expensive technology mapping, producing highly efficient 'U'-correct compiled logic simulation code in seconds rather than in hours. Our experiments toward confirming the validity of the method are reported.

1 Introduction

The simulation of register-transfer and gate-level models remains an important design methodology, especially for the earlier stages of digital system design. Many such simulators, VHDL as an example [Jen91b] [Ins88], permit *multivalued* simulation [Urq86]. In this paper we are concerned with values which represent an "unknown" logic value 'U'. In this context 'U' does not mean "an illegal logic voltage," neither are we trying to define a pseudo-continuum '0'→'U'→'1'. Rather, 'U' means that we have a *valid* logic one or logic zero, but we don't know *which.* This "unknown" value is used for example in [L+88] and [Jen91b] as the initial value for all binary signals in the simulation.

During circuit design, we prefer describing logic components (such as adders, ALUs etc.) in behavioral terms, using high-level descriptions such as BDS or state-machine languages [B+86] [BC87], since this permits us to express our intent more quickly to both the simulator and the implementation tools. However, the value 'U' must be correctly managed during simulations based on these descriptions. Using a state-machine behavior as an example, an overly-conservative simulator might generate 'U' at *all* the outputs if it sees a 'U' enter *any* of the inputs, but this frustrates any attempt to simulate the reset sequence for the design. At the other extreme, the simulator could gratuitously replace each incoming 'U' by (for example) a logic one; but not only do we now have misleadingly purged behavior at the outputs, but also the unfortunate loss of symbolic output information ('U' might not emerge any more) which could have tested the sensitivity of "downstream" components to unknown logic values. 'U' furthermore must be properly treated as a genuine (although unknown) logic value. For example, the Boolean expression "X AND NOT(X)" should evaluate to logic zero, even if a 'U' enters signal X during simulation. Finally, we prefer *compiled simulation*, since it gives us a typical two order of magnitude speed improvement over event-driven simulation [Jen91a].

We considered two approaches to generating "U-correct" compiled simulation code from our behavioral descriptions. The first was to pass the behavior through the implementation tool (including logic and state minimization, technology mapping and so forth) and obtain a netlist of gates. That is, we "mapped" the logic description into a small repertoire of gates, having a corresponding library of gate-level simulation models which correctly handle all possible cases of incoming '1', '0', and 'U' at the gate inputs. All that remained was to transform the resulting mapped-logic netlist into simulation code. This is a straightforward application of topological sort [WM89] [Jen91a] which generates a sequence of "subroutine calls" to the low-level gate models. The result is a complete simulation code module for that particular logic component. However, there are significant disadvantages to this approach:

- logic synthesis ("mapping" to gates) can be expensive, with times easily measured in hours.
- synthesis tools tend to map away "don't-care" information in the original logic description, which again can mask the overall design's inherent vulnerability to 'U' values as discussed above.
- regardless of available abstractions in the original behavioral description, the resulting simulation code is a (potentially long) list of calls to primitive gate-level subroutines, that is, each and every "gate" generally has to be evaluated.
- in our case we faced an additional obstacle, in that our use of certain signal abstractions sometimes *prevented* logic synthesis [Jen91b] [JLI94].

The second possible approach seeks to avoid implementing ("mapping") the logic description by instead generating simulation code more directly from the original logic description, yet which correctly reacts to 'U' values. Emitting simulation code as a decision diagram yields very fast simulation code [CG85]. This is because evaluation of a function simply means that we trace a path in the diagram, whose length is bounded by the number of input variables (generally *much* smaller than the number of "gates" needed to describe the same function). However, all prior such work in logic simulation has been based on ordered binary decision diagrams (OBDDs, [Bry86]). These require that all input variables have firm '1' or '0' values during evaluation. We could find no simple way to use the OBDD directly for correctly managing incoming 'U' values during evaluation (that is, during simulation). Therefore we sought to develop a new decision diagram which could manage the "unknown" logic value.

This paper reports our development of the Ordered Ternary Decision Diagram (henceforth OTDD) as well as its first application to generating compiled simulation code for a conventional PLA and state-machine language. We briefly describe OTDDs and their implementation. We then describe our experiments into

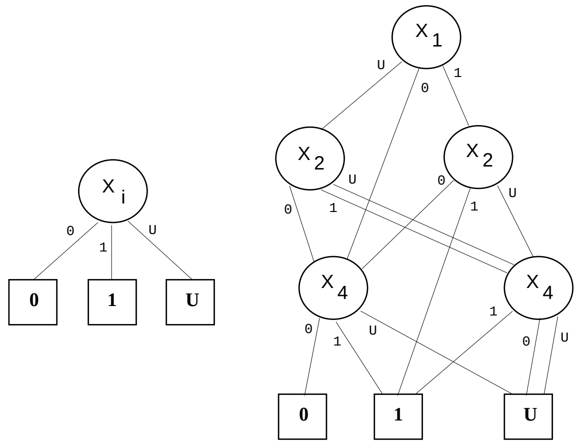

Figure 1: Simple Ordered Ternary Decision Diagrams. To left: the identity function for variable x_i. To right: the function $(x_1 \wedge x_2) \vee x_4$.

the validity of OTDDs in dealing with 'U'. The resulting state-machine language has been integrated into the GRTL compiled simulation RTL design platform [Jen91c] [Jen91b] in use at the Luleå Institute of Technology. By parsing the input descriptions directly into the new decision diagram (avoiding technology mapping) we can now obtain efficient 'U'-correct compiled simulation code in seconds rather than hours.

2 OTDDs and Earlier Work

Although we do not intend to give a formal exposition of the OTDD (since we felt this would be inappropriate in the present forum) nevertheless by claiming to ourselves the designation "Ordered Ternary Decision Diagram" we have been requested to defend this, given that there are other works which have used designations which are very similar to this. Here we will cite those efforts which we are aware of, and will explain that our OTDD is indeed distinct from all of these. Furthermore much of the controversy centers around the word "ternary," and we will assure these other authors that we do not simply mean "three-branched," but rather that we are proposing a very specific decision diagram, having *very* specific semantics, based on *strong ternary logic* [Kle52] [HNY93].

The Ordered Binary Decision Diagram, or OBDD, we attribute to Bryant [Bry86]. It is important to recognise that the OBDD is a decision diagram for functional manipulation of *Boolean logic,* and *not* just a syntactical data structure devoid of any such semantics. Together with the development of the shared, re-

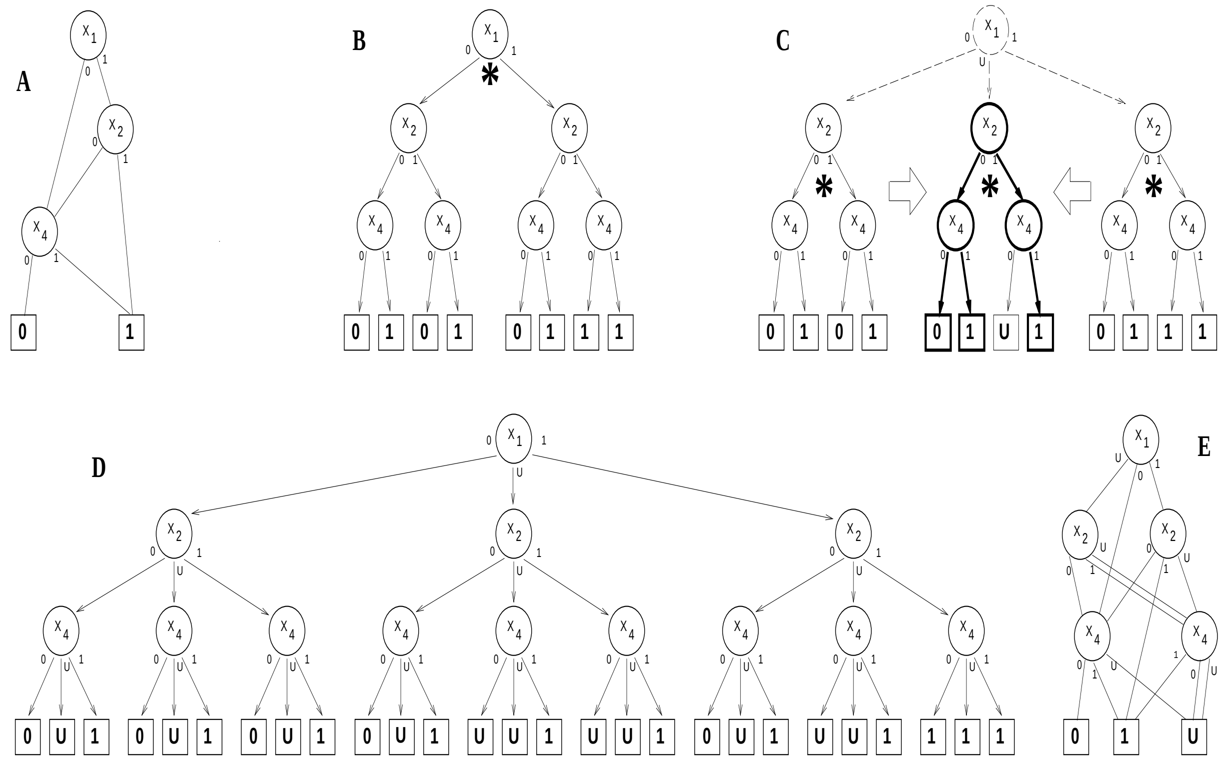

Figure 2: Developing the OTDD for $(x_1 \wedge x_2) \vee x_4$ from its OBDD (A) by first expanding into the corresponding decision tree (B) and preparing for the topmost extraction (asterisk, B) of '0' and '1' subtrees. The resulting extraction subtree (center, C) receives a terminal 'U' wherever the corresponding terminal in the '0' subtree differs from that of the '1' subtree. The extraction subtree becomes the new 'U' child to the top node. The process is then repeated recursively down the tree (asterisks, C). The resulting full ternary decision tree (D) can then be reduced to a directed acyclic graph (E) which is the final OTDD.

duced OBDD representation having a *strong canonical form* (in which isomorphism can be tested in constant time) [BRB90] we first begin to see the notion of a *binary* decision diagram (that is, each nonterminal vertex still having only *two* branches, labelled '0' and '1') but which now has three *terminal* vertices, namely '0', '1' and "undefined" as in the "ternary-valued BDD" of [MIY90], the Ordered Partial Decision Diagrams (OPDD) of [RB+91] and most recently the Modified Ordered Binary Decision Diagram (MBD) of [JT93]. In none of the above cases are we aware of any proposal to further process such diagrams against each other by using Kleenean ternary logic, much less the particular semantics we give to the 'U' branch for our nonterminal vertices (below). More abstract efforts which seem to lay claim to general "n-ary decision diagrams" for any cardinal n can be cited, such as the MDD of [SKMB90] and the NDD of [KP+92]. Here we first see *nonterminal* vertices having *more* than two branches, although the definitions tend to be syntactical leaving us often to wonder whether any *meaning* [YM88] can be attached in general to such an "n-ary decision diagram."

By our designation "Ordered Ternary Decision Diagram" we did not intend to encroach on any of the efforts cited above. Our "ternary" was meant to convey our *semantical* foundation, namely ternary Kleenean logic. We cite [Kle52] although Kleene himself cites his earlier work from 1938; see also [Urq86]. With the recent publication of [HNY93] we have also become aware that there is considerable interest in Kleenean ternary logic within current research into "fuzzy logic," yet even here we are unaware of any previous work for functional manipulation of Kleenean ternary logic functions by decision diagram.

We now give an intuitional exposition of the OTDD for a specific example. In Figure 1 we show OTDDs corresponding to the simple OBDDs given in Figure 1 of [Bry86]. The first conceptual extension over the OBDD is that each nonterminal always represents a three-way decision, depending on whether its value is '0', '1', or unknown ('U'), with the same three values also available as terminal vertices. The second conceptual extension is in the particular semantics which we assign to the 'U' branch in nonterminals, which we will

```
char op_eval( a, b, op ) {
  if (op == AND) {
     if ((a == '0')||(b == '0')) {
        res = '0';
     } else if ((a == 'X')||(b == 'X')) {
        res = 'X';
     } else if ((a == '1')&&(b == '1')) {
        res = '1';
     } else {
        res = 'U';
     }
  } else if (op == XOR) {
      :
      :
  } else if (op == A_AND_NOT_B) {
     if ((a == '0')||(b == '1')) {
        res = '0';
     } else if ((a == 'X')||(b == 'X')) {
        res = 'X';
     } else if ((a == '1')&&(b == '0')) {
        res = '1';
     } else {
        res = 'U';
     }
  } else if (op == OR) {
     if ((a == '1')||(b == '1')) {
        return '1';
     } else if ((a == 'X')||(b == 'X')) {
        return 'X';
     } else if ((a == '0')&&(b == '0')) {
        return '0';
     } else {
        return 'U';
     }
 } else if (op == MAKE_U) {
     if ((a == 'U')||(b == 'U')) {
        return 'U';
     } else if ((a == 'X')||(b == 'X')) {
        res = 'X';
     } else if (a == b) {
        res = a;
     } else {
        res = 'U';
     }
  }
}
```

Figure 3: Pseudo-"C" code fragment for *op* evaluation.

now demonstrate by example. We will show how the ternary decision diagram for $(x_1\ and\ x_2)\ or\ x_4$ shown in Figure 1 is derived from the corresponding OBDD. We first expand the binary function $(x_1\ and\ x_2)\ or\ x_4$ (Figure 2 A) into its full binary decision *tree* [RB$^+$91], see Figure 2 B. Consider now the top variable; it has two child subtrees, one for each of its '0' and '1' branches. We will create a new third subtree (for the new 'U' branch) as a function of these first two subtrees, a function which we at present (perhaps unfortunately) call *extraction.* The primitive ternary function for extraction is apparent in Figure 3 as the case "MAKE_U". The result is a third subtree which is isomorphic to the original two *except* that a 'U' terminal appears in the new tree wherever the corresponding original two terminals do not match each other, see Figure 2 C. The same extraction procedure is now carried out on all three sons of the top variable, and so on recursively down the entire tree. The result is a ternary decision tree as in Figure 2 D which can then be reduced to a directed acyclic graph giving the final result of Figure 2 E. Of course in practise we do not explicitly expand OBDDs and OTDDs into trees, but operate on them directly. Again we point out that the 'U' branch at any nonterminal has a very specific relationship to the '0' and '1' branches for the same nonterminal, namely that the 'U' branch represents the *extraction* of its '0' and '1' brothers. An arbitrary 3-branch diagram which does not have this *particular* semantics is *not* an OTDD, therefore we have set forth a very particular definition for the OTDD, however informally we may have stated it here.

3 Basic OTDD Implementation

Since we are addressing the practitioner, we choose simply to describe the implementation of Ordered Ternary Decision Diagrams in terms of direct modifications to the pseudocode given in Bryant's landmark article on basic Ordered Binary Decision Diagrams [Bry86]. This involves both extensions to the OBDD data structures defined in [Bry86] as well as modifications to the algorithms for the *reduction, apply,* and *restriction* operations, with subroutines *traverse* and *op* used by those operations. The data structure for *type vertex* [Bry86] is extended by adding an additional member *"unk: vertex"* to provide the new outgoing edge for the case 'U' (in addition to the *"low, high: vertex"* for the cases '0' and '1'), and by extending the enumerated type for *val* to *"val: (0,1,U,X)".* With those changes to the *vertex* structure, the extensions to the algorithms described in [Bry86] are readily

```
apply_step( a, b, op )
  res = op_eval( a.val, b.val, op );
  if  res <> 'X'  then return [constant-‘res' OTDD];
  if  a.index < b.index
    then  low_b = high_b = b;
    else  low_b = b.low,   high_b = b.high,
  if  a.index > b.index
    then  low_a = high_a = a;
    else  low_a = a.low,   high_a = a.high,
  f_0 = apply_step( low_a,  low_b,  op );
  f_1 = apply_step( high_a, high_b, op );
  f_u = apply_step( f_0,  f_1,  MAKE_U );
  return [create OTDD, index = min(a.index,b.index),
          pointers low = f_0, high = f_1, unk = f_u];
end apply_step;
```

Figure 4: Pseudo-code fragment for “Apply-Step” [Bry86] showing computation of the *unk* branch using extraction function “MAKE_U”.

apparent: in all cases where there is a decision based on *val,* a new case for the ‘U’ terminal must be provided; where actions are taken on both *low* and *high,* a third action is added for pointer *unk.* All necessary modifications will be obvious to the implementor of OTDDs, with four critical exceptions.

The first exception is the subroutine *op* which evaluates results in the Kleenean (“strong”) ternary logic system $\{0, 1, U\}$ [Kle52]. We provide guidance to the reader in the form of pseudo-code for representative operations, please see Figure 3. The second exception is the structure of the “Apply” procedure, see Figure 4, where the extraction computation used for the *unk* pointer is shown.

The third and fourth exceptions both occur in the *reduction* operation. While one of these is a programming pitfall, the other exception goes deep into the peculiar nature of the OTDD. The programming pitfall is as follows: when sorting the selected set of elements Q by key (see function *Reduce* in [Bry86]), the terminal ‘U’ must be treated as having a value *less than* both terminals ‘0’ and ‘1’, otherwise the case “matches existing vertex” will not always work as intended.

The remaining exception, also in the *reduction* algorithm and having a more fundamental nature, is in the determination of redundant vertices. In the original OBDD algorithm, this is determined by comparing *u.low.id* against *u.high.id*, and the reader may wonder whether one must now bring *u.unk.id* into the comparison. The answer is no: it is only necessary to compare *u.low.id* against *u.high.id* as in the original OBDD algorithm. It is here that the OTDD treats ‘U’ as a true logic value, not as some poorer third

```
   $ON     reset       => full_idle $GOTO main

a: $IF (! write) & hit => read_hit  $REPEAT
   $ELSEIF write & hit
           & (! mwait) => write_hit $REPEAT
   $ELSEIF write & hit => mem_wait  $GOTO easy
   $ELSEIF mwait       => mem_wait  $GOTO poor
   $ELSE                  init_rd_1 $GOTO hard
       :
       :
   $IF mwait           => mem_wait  $REPEAT
   $ELSE                  load_2    $GOTO main
```

Figure 5: Fragment of a state machine description demonstrating a typical if–then–else structure.

cousin: intuitively, one can reason that if the ‘0’ case and the ‘1’ case are the same, then we have obviously covered the ‘U’ case as well, since ‘U’ must be (in actuality) either one or the other; so if we find that ‘0’ and ‘1’ behave the same, then we can discard the ‘U’ case altogether. Analytically, one can see that *extraction* on isomorphic *zero* and *one* branches will always yield an *unk* branch which is also isomorphic to both. When using *shared reduced* decision diagrams as in [BRB90], this means that we can avoid computing the *unk* branch altogether, should we find that the *zero* and *one* branches are equal.

4 A State Machine Language

Our first application of OTDDs was to parse the input condition space used in a conventional state machine (“PLA”) language. Figure 5 shows a typical such description, providing an ordered *IF – ELSIF* structure where an input configuration is associated with the first condition which covers it. Both the test conditions and the descriptions which keep track of the remaining portions of the input space can become very complex. Our problem was to determine which, if any, of the cases covered by the state machine description should be executed when one or more ‘U’ values appeared at the inputs during simulation.

This was solved using OTDDs by translating each case's condition into an OTDD. Each such OTDD, when evaluated by assigning ‘0’, ‘1’ or ‘U’ to each *input* variable, would lead to one of three possible values as its *output:* ‘1’ meaning that the configuration was covered by this expression, that is, execute this case; or, ‘0’, meaning that the configuration was not covered by this expression, so test against the next case; and finally, 'U', meaning that neither could be determined, that is, an ambiguity which could not be resolved (which we treated as an "error" condition). This was the most conservative use of OTDDs which we were able to devise, completely excluding any loss of "don't care" information at the outputs since it avoids any semblance of technology mapping. Furthermore the resulting simulation code produces output values which the designer can immediately recognize in his original description, making it easier for him to identify which case had been selected by the simulator.

If such a direct case correspondence can be sacrificed, then a more direct method of generating the outputs is to construct an OTDD for each individual PLA *output* bit. This is done by constructing the input case OTDDs as above, and then observing whether the output bit for that case is to evaluate to '1', '0', or else is not specified ("don't care" is a valid interpretation of 'U' [Kle52]). Using an "accumulation" OTDD initialized to constant 'U', the cases are accumulated by using *Apply* [Bry86] having the accumulator as first operand, input cube as second operand, and either function OR (see Figure 3) if the output evaluates to '1', or function A_AND_NOT_B if the output evaluates to '0' (no application if the output evaluates to 'U'). The simulation code is created by simply emitting the resulting output-bit OTDDs directly. An example of such generated code is shown in Figure 6.

5 Results

We set out to validate both OTDDs and their application in these contexts. Since GRTL [Jen91b] already included a compiler for a more or less conventional finite state machine language, we modified this to use OTDDs during parsing. This permitted us to assemble a test suite of finite state machine ("PLA") input descriptions from several full-scale digital system designs already at hand. We also included some very simple descriptions into the test suite to test the basic Boolean operations supported by the "PLA" language. This "PLA" test suite was processed by the new OTDD-based implementation, and the resulting simulation code was subjected to a number of different tests.

First we verified the {0,1} behavior of the OTDD-based code against the old implementation's code (in which we have full confidence) by exhaustive generation of {0,1} input test vectors. These were presented to both the old and new compilers' code and the outputs from the two were compared. We then generated

```
        /* output bit n_3: */

_2: if (res == '0') {goto _3;} else
    if (res == '1') {goto _4;} else {goto _5;}
_3: if (p_3 == '0') {goto _6;} else {goto _5;}
_6: if (p_2 == '0') {goto _7;} else
    if (p_2 == '1') {goto _8;} else {goto _5;}
_7: if (p_1 == '0') {goto _9;} else
    if (p_1 == '1') {goto _4;} else {goto _5;}
_9: if (p_0 == '1') {goto _4;} else {goto _5;}
_8: if (p_1 == '0') {goto _4;} else {goto _5;}
_4: n_3 = '0'; goto _1;
_5: n_3 = 'U'; goto _1;
_1: ;
```

Figure 6: Direct-output-bit simulation code fragment generated by GPLA.

test vectors containing unknown logic values 'U' in random positions, and obtained the result from the new code; and this result was checked against the results obtained (also) from the new code by replacing the set of 'U's in each test vector by all possible combinations of '0' and '1'. With this we gained confidence that the 'U' value was being correctly handled for all cases we had tested so far, and that both OTDD theory and implementation were correct. These tests were concluded by exhaustive {0,1,U} input vector generation for all "PLA"s in the test suite; no errors were found, and both generation methods (test-case-by-case and direct) were subjected to testing. This OTDD-based state machine compiler is now fully integrated into the GRTL register-transfer design tool, so that it will be subjected to continual testing by computer engineering students [JLI94].

6 Conclusions

We have presented a method to generate logic simulation code which correctly evaluates even in the presence of any number of indeterminate input logic values, applicable in multivalued simulators which support an "unknown" logic value. This method is the first application of a new decision diagram which we have called the Ordered Ternary Decision Diagram (OTDD). We have described the implementation of the OTDD and described two possible ways to emit simulation code from an input description parsed into OTDDs. The first treats input conditions as characteristic functions which are sequentially searched, while the second synthesizes each output bit directly.

Generated simulation code for both methods exhibits 'U'-correct behavior in all experiments we have been able to devise, over a suite of finite state machine descriptions taken from actual full-sized digital constructions.

References

[B$^+$86] R. Brayton et al. Multiple-Level Logic Optimization System. In *IEEE ICCAD*, 1986.

[BC87] M. C. Browne and E. M. Clarke. SML - a High Level Language for the Design and Verification of Finite State Machines. In Dominique Barrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 269–292. Elsevier Science Publishers B.V. (North-Holland), 1987.

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient Implementation of a BDD Package. In *Proceedings 27th DAC*, pages 40–45, June 1990.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CG85] E. Cerny and J. Gecsei. Simulation of MOS Circuits by Decision Diagrams. *IEEE Transactions on CAD*, CAD-4(4):685–693, October 1985.

[HNY93] Yutaka Hata, Kyoichi Nakashima, and Kazuharu Yamato. Some Fundamental Properties of Multiple-Valued Kleenean Functions and Determination of Their Logic Functions. *IEEE Transactions on Computers*, 42(8):950–961, August 1993.

[Ins88] Institute of Electrical and Electronics Engineers, New York, NY USA. *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-1987 edition, 1988.

[Jen91a] G. Jennings. A Case against Event-Driven Simulation for Digital System Design. In Alan H. Rutan, editor, *Proceedings, 24th Annual Simulation Symposium*, pages 170–176, April 1991.

[Jen91b] G. Jennings. GRTL – a Graphical Platform for Pipelined System Design. In *Proceedings of the 1991 European Conference on Design Automation (EDAC 1991)*, pages 424–428, February 1991.

[Jen91c] G. Jennings. Reversible Functional Simulation for Digital System Design. In *Proceedings of the IEEE 1991 Custom Integrated Circuits Conference (CICC '91)*, pages 8.2.1–8.2.4, May 1991.

[JLI94] G. Jennings, P. Lindgren, and J. Isaksson. A State Machine Language Supporting Integer Inequalities Implemented with Ordered Ternary Decision Diagrams. In Philip A. Wilsey and David Rhodes, editors, *Proceedings, 1994 Int'l Conference on Simulation and Hardware Description Languages (ICSHDL)*, pages 23–28. Society for Computer Simulation (SCS), January 1994.

[JT93] Ricardo P. Jacobi and Anne-Marie Trullemans. A New Logic Minimization Method for Multiplexor-Based FPGA Synthesis. In *Proceedings, European Design Automation Conference (EURO-DAC '93)*, pages 312–317, September 1993.

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics.* Wolters-Noordhoff, North-Holland Publishing, 1952. Chapter 12, Section 64: The 3-valued Logic.

[KP$^+$92] Ken Kubiak, Steven Parkes, et al. Exact Evaluation of Diagnostic Test Resolution. In *Proceedings 29th DAC*, pages 347–352, 1992.

[L$^+$88] M. Loughzail et al. Experience with the VHDL Environment. In *Proceedings of the 25th Design Automation Conference*, pages 28–33, June 1988.

[MIY90] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation. In *Proceedings 27th DAC*, pages 52–57, 1990.

[RB$^+$91] Don E. Ross, Kenneth M. Butler, et al. Fast Functional Evaluation of Candidate OBDD Variable Orderings. In *Proceedings of the 1991 European Conference on Design Automation (EDAC 1991)*, pages 4–10, February 1991.

[SKMB90] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for Discrete Function Manipulation. In *Proceedings ICCAD '90*, pages 92–95, November 1990.

[Urq86] Alasdair Urquhart. Many-Valued Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, chapter III.2, pages 71–116. D. Reidel, 1986.

[WM89] Zhicheng Wang and Peter M. Maurer. Scheduling High-Level Blocks for Functional Simulation. In *Proc. 26th DAC*, pages 87–90, June 1989.

[YM88] Yoshinori Yamamoto and Masao Mukaidono. Meaningful Special Classes of Ternary Logic Functions – Regular Ternary Logic Functions and Ternary Majority Functions. *IEEE Transactions on Computers*, 37(7):799–806, July 1988.