Concurrent D-Algorithm on Reconfigurable Hardware

Fatih Kocan and Daniel G. Saab
Electrical Engineering and Computer Science Department
Case Western Reserve University, Cleveland, Ohio
(kocan,saab)@eecs.cwru.edu

Abstract

In this paper, a new approach for generating test vectors that detects faults in combinational circuits is introduced. The approach is based on automatically designing a circuit which implements the D-algorithm, an Automatic Test Pattern Generation (ATPG) algorithm, specialized for the combinational circuit. Our approach exploits fine-grain parallelism by performing the following in three clock cycles: direct backward/forward implications, conflict checking, selecting next gate to propagate fault or to justify a line, decisions on gate inputs, loading the state of the circuit after backup. In this paper, we show the feasibility of this approach in terms of speed, and how it compares with software based techniques.

1. Introduction

ATPG is the process of either finding input vectors that detect a fault in digital circuits by distinguishing the faulty and fault-free circuit behavior at Primary Outputs (PO) or flagging a fault redundant when no such vector exists. This process requires a large amount of CPU time and in many cases they abort many of the hard-to-detect faults. It is known that the ATPG is NP-complete even for combinational circuits[13].

Most existing deterministic ATPG techniques employ a branch-and-bound [1] technique to examine all input combinations. The D-algorithm, in [12], examines all input combinations by making decisions at internal circuit nodes as well as primary inputs and alternates between fault propagation and line justification processes until some faulty values appear at the primary circuit output (PO) (The fault is detected) or the search space is exhausted. In this later case, the fault is flagged as being redundant. The PODEM, in [7], examines all input combinations by making decisions only on primary inputs (PI). This way the number of nodes appearing in the search tree is reduced. To achieve this, all decisions on internal lines are traced back to the Primary Inputs (PI). The FAN algorithm, in [9], presents the following improvements to the basic PODEM algorithm: tracing of objectives stops at some internal lines (head-lines) in addition to PIs and multiple objectives are back-traced instead of a single objective back-tracing as used in PODEM. Further improvements are made to FAN to have a better performance by finding mandatory assignments based on dominators and by finding nodes where values can be assigned independent of other nodes [10]. SOCRATES utilizes a unique sensitization technique based on dominators and implication learning to speed the justification process [11]. Recursive learning that avoids the use of decision tree is proposed in [14]. Other improvement to speed the ATPG process is found in [16].

Emulation systems are being used increasingly in the design, verification, and in rapid prototyping of digital systems [3]. To increase the use of these emulation systems, several methods are proposed to emulate Computer-Aided-Design (CAD) algorithms such as fault simulation [4,5], Automatic Test Pattern Generation (ATPG) [1], Satisfiability (SAT) [1,6], and Fault diagnosis [8,15]. In [4], a method is proposed to emulate serial fault simulation. In [5], a method is proposed to emulate critical path-tracing algorithm. In [1], a method is proposed to emulate PODEM algorithm with its application to SAT. In all of those algorithms, a significant speed-up was obtained over software based implementation.

In this paper, we present a new method to emulate the D-algorithm on a reconfigurable hardware. The method achieves significant speed-up over software-based ATPG techniques with similar or better results. The quality of the results is measured in terms of fault coverage. This is achieved by utilizing reconfigurable hardware that provides a way to exploit the fine-grain parallelism in the D-algorithm.

This paper is organized as follows: In section 2, the concurrent D-algorithm is presented. In section 3, the overall architecture of the implementation is given. In section 4, we present results. Finally, we present conclusion and future work in section 5.

2. Concurrent D-Algorithm

The concurrent D-algorithm is shown in Fig. 1. The algorithm generates a test for a fault F or declares it redundant. The algorithm associates with each gate G a list of fan-in states stored in Fanin(G,l) and four signals DF[G], JF[G], DIMP[G], VAL[G] indicating that a gate is on the D-frontier, J-frontier, has a direct implication, and the logic value of the direct implication respectively. For buffers and inverters, only DIMP[G] and VAL[G] are generated. In addition, for each line l in the circuit we associate a state S[l].

The algorithm starts by activating a fault F and initializing S[l] from the gate fan-in states stored in Fanin(G,l) (Initially, all fan-in states are set to undefined X and they may be changed during the execution of the algorithm). The initialization and fault activation are performed in three clock cycles. The algorithm proceeds to perform forward implications and during this step sets the indicators DF[G], JF[G], DIMP[G], and

0-7803-5832-X /99/$10.00 ©1999 IEEE.

VAL[G]. The state S[l] of a line l that is connected to the fan-in of a gate G with DIMP[G] set is updated with a VAL[G]. The Conflict signal is set if the implied value in VAL[G] is different from the binary value stored in S[l]. Conflict signal is also set during the Forward implication where the implied value is different from the binary value stored in S[l]. The forward implication and the backward direct implication processes are repeated until a conflict occurs or no more direct implication exists. In our implementation, each iteration requires one clock cycle. In the case of a conflict the program transfers control to a backtrack procedure which reverses the previous decision. In the other case, the procedure tries to justify nodes in the J-frontier when error is at a PO or propagates an error in the D-frontier otherwise. If all nodes in the J-frontier are justified then a test is found. In the case where justification fails, the control is transferred to backtrack. During the justification, each gate is pushed on the stack and the state of its fan-in are either initialized (InitFaninState(G)) if the G is pushed for the first time (indicated by the Increment flag) or the next appropriate fan-in state is computed (NextFaninState(G)). These steps are also performed in one clock cycle. Similar steps are performed during the propagation process which is activated when no error is at any POs. Next, the algorithm loads the changes on the fan-in nodes of gate G, due to a decision, into S[l] (LoadLineStateFromFanin(G)) and resets the Increment and BackUpSet flags. These steps are repeated until a test is found or a fault is redundant.

```
Concurrent D-Algorithm()
 Fanin(G,l); Holds the state of fan-in l of gate G.
 S[l] : Holds the logic state of circuit node l.
 DF[g]: A signal that is set when G is on the D-frontier
 JF[g]: A signal that is set when G is on the J-frontier
 DIMP[g]: A signal that is set when G has a direct implication
 {
     Activate_Fault();
     Increment ← BackUpSet ← FALSE;
     LastProcessedGate ← ∅;
 Init: for each gate G do LoadLineStateFromFanin(G);
     while(TRUE){
       repeat
         Forward_imply();
         BackwardDirectImplication(VAL);
         LoadDirectlyImpliedValues();
       until Conflict or all DIMP==0
       if ( Conflict ) goto Backtrack;
       if ( ErrorAtPO ){ /* Process J-frontier */
          G ← SelectGateFromJF(LastProcessedGate);
          if(G==∅){ if ( BackUpSet ) goto Backtrack;
                     TestIsFound; }
          else if (Increment) {Push(G); NextFaninState(G); }
               else{ Push(G); InitFaninState(G); }
       }
       else{  /*Process D-frontier */
           G ← SelectGateFromDF(LastProcessedGate);
           if (G==∅) goto Backtrack;
           Push(G); InitFaninState(G);
     }
     LoadLineStateFromFanin(G);
     Increment ← BackUpSet ← FALSE;
  }
 Backtrack: Pop();
    if(StackIsEmpty)   RedundantFault←TRUE;
    else{ G←TopStack(); Pop();  BackUpSet ← TRUE;
       if (AllFaninState(G) are tried ) SetFaninState(G,X);
                    else Increment ← TRUE;
       LastProcessedGate = G;
       goto Init;
       }
```

Fig. 1. Concurrent D-algorithm.

Note that when selecting a gate from the D-frontier/J-frontier (SelectGateFromDF/JF()) the decision on the fan-in gate which was last pushed onto the stack is examined and if more decisions exist, then gate is selected again; Otherwise a new gate is selected. It should be mentioned that in the implementation the same logic is activated depending on ErrorAtPO signal for selection of a gate from D-frontier or J-frontier sets.

3. Implementation

The overall architecture is shown in Fig. 2. It consists of a Fault Activator[1], Forward Network, Backward Network, Signal Computation, Frontier Selection, Stack, and a Decision Block. The circuit starts by activating a fault that is performed in the fault activator (FACT). FACT sets S[i] to 0/1 for a stuck-at-1/0 fault on line i (F_i). FACT is similar to the circuit in [4] and consists of a shift register where each flip-flop corresponds to a stuck-at fault on a line i in the circuit.

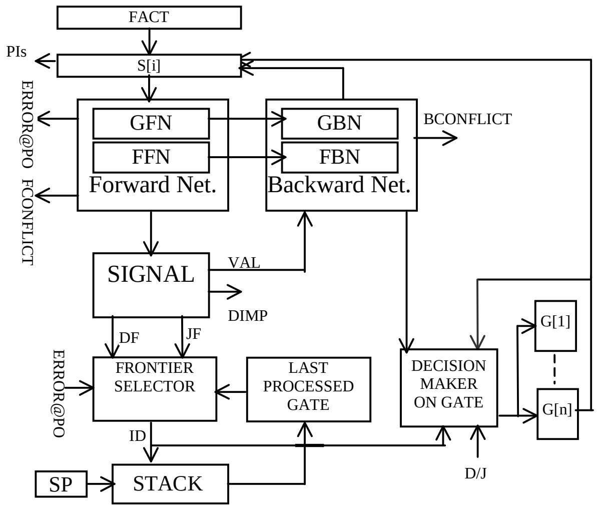


Fig. 2. The overall high-level block of data-path.

The forward network computes the effect of the changes in S[I] on the faulty (FFN) and fault-free (GFN) circuit. The forward network sets the conflict signal in the case where implied values conflict with previous decision values. The Signal block computes for each gate its DF, JF, DIMP, VAL according to equations (3-8). These equations are explained later. The backward network propagates directly implied gate VAL one level backward in the faulty (FBN) and fault-free (GBN). These values are stored in corresponding S[I]. These steps are repeated until either the FCONFLICT/ BCONFLICT is set or none of DIMP is set. The Frontier Selector Block consists of a priority encoder that selects either a DF gate if error is not at the PO or a JF gate otherwise. The gate identification that was processed previously is stored in LastProcessedGate. This is used to be passed to the stack if more decision can be made on this gate

[1] All equations uses the convention described in Table 1.

otherwise it is used to exclude gates with lower priorities from the selection process because the search space associated with these gates has already been explored. The selected gate is pushed onto the STACK and the associated decision on its Fanin(G,I) are computed in the Decision Maker block, and the decisions are stored in corresponding G[I]. In the case of either a conflict, error is not at the output and all DF are not set, all justification associated with set JF gates failed the gate on the top of the stack is popped and stored in LastProcessedGate. All these steps are repeated until a test is found or no more decision can be made which signals a redundant fault.

The forward network consists of the circuit under test where the gates are interconnected with lines. Each line is modeled as in Fig. 3(a)-(c). The forward networks compute the *final* values of a line by considering the state value (S_i), the *implied* value computed by the fan-in gate (I_i), and the fault injection signal (F_i). The *final* value of line is computed using (Eq. 1.a). A conflict signal is also computed for every line using (Eq. 1.b). In the case where a fault (F_i) is not injected at the line the expression in Eq. (1.a) is simplified. Fig. 4 shows a 2-input AND gate and its corresponding model in faulty forward network.

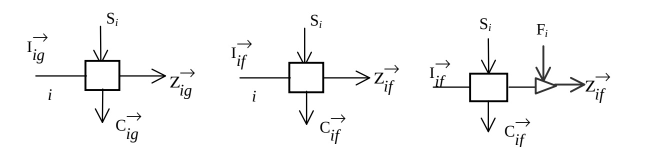
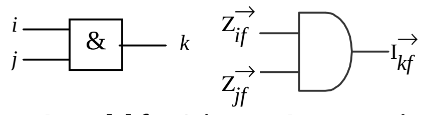

S_i
$\overrightarrow{I_{ig}}$
i
$\overrightarrow{Z_{ig}}$
$\overrightarrow{C_{ig}}$
S_i
$\overrightarrow{I_{if}}$
i
$\overrightarrow{Z_{if}}$
$\overrightarrow{C_{if}}$
S_i
F_i
$\overrightarrow{I_{if}}$
$\overrightarrow{Z_{if}}$
$\overrightarrow{C_{if}}$
a) Good line in GFN b) Good line in FFN c) Faulty line in FFN

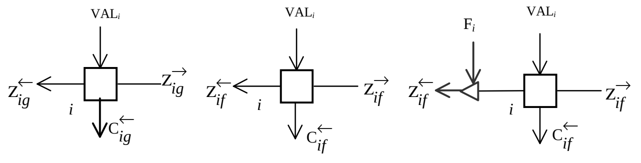
Fig. 3. Line models in forward networks.

$$\overrightarrow{Z_{if}} = (\overrightarrow{I_{if}} \vee S_i) \oplus F_i \tag{1.a}$$

$$\overrightarrow{C_{if}} = \overrightarrow{I_{if}} \neq S_i \wedge S_i \neq x \tag{1.b}$$


i
j
&
k
$\overrightarrow{Z_{if}}$
$\overrightarrow{Z_{jf}}$
$\overrightarrow{I_{kf}}$
Fig. 4. A model for 2-inputs AND gate in FFN.

Similarly, backward network (GBN/FBN) propagates the fault-free/faulty backward implications of the circuit. The backward networks use the line models in Fig. 5(a)-(c) to propagate the direct implication values backward. The *final implication* value of a line is computed from the signal (VAL_i), *final* value computed by the forward network and the fault injection signal (F_i) and given in equation (2.a). The conflict signal is set according to (Eq. 2.b) in the case of conflicting implied values on a line.



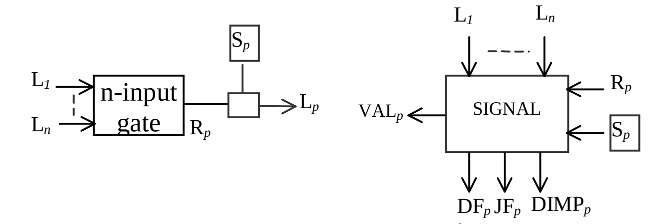
a) Good line in GBN b) Good line in FBN c) Faulty line in FBN

Fig. 5. Line models in backward networks.

$$\overleftarrow{Z_{if}} = (\overrightarrow{Z_{if}} \vee VAL_i) \oplus F_i \tag{2.a}$$

$$\overleftarrow{C_{if}} = \overrightarrow{Z_{if}} \neq VAL_i \wedge VAL_i \neq x \tag{2.b}$$

The signal block computes for each gate *i* the following: JF_i, DF_i, $DIMP_i$, and VAL_i. Those are computed for all gates with more than one fan-in. DIMP and VAL are computed for all gates. The equations (3-8) compute JF_i, DF_i, $DIMP_i$ and VAL_i using 5-valued logic (D and ¬D represent 1/0 and 0/1 respectively). The gate type is encoded into these equations by the inclusion of its controlling value c. Fig. 6 shows the other parameters used in these equations. The *final* line values in the good and faulty forward networks are combined and mapped into a 5-valued logic represent with L_i and R_p in Fig. 6.



a) A n-input gate b) Model for an n-input gate

Fig. 6. The frontier/implication model for n-input gate (n>1).

Table 1. Nomenclatures.

t: t∈{f = faulty, g=good}
d: d∈{→ = forward, ← = backward }

Z_{lt}^{d} : the *final* value of a line *l* in the *td* network
I_{lt}^{d} : the *implied* value of a line *l* in the *td* network
C_{lt}^{d} : the conflict value of a line *l* in the *td* network
R_i : the *implied* value of a line *i* in the forward network
L_i : the *final* value of a line *i* in the forward network
VAL_l: direct implication value on line *l*
G_i^c : the inputs of the gate *i* from the circuit
G_i^g : the inputs of the gate *i* from its gate fan-in state

$$JF_p \leftarrow R_p = x \wedge S_p \neq x \wedge (\exists_{i,j}\ L_i = x \wedge L_j = x) \quad 1 \leq i < j \leq n \tag{3}$$

$$DF_p \leftarrow R_p = x \wedge S_p = x \wedge (\exists_{i,j}\ L_i = D \oplus \exists_j\ L_j = \neg D) \quad 1 \leq i < j \leq n \tag{4}$$

$$DIMP_p \leftarrow \begin{cases} R_p = x \wedge S_p = c \wedge [(\sum_i L_i = x) = 1] \vee R_p = x \wedge S_p = \neg c \\ \qquad 1 \leq i \leq n \text{ and } p\text{-type} \in \{AND, OR\} \\ R_p = x \wedge S_p = \neg c \wedge [(\sum_i L_i = x) = 1] \vee R_p = x \wedge S_p = c \\ \qquad 1 \leq i \leq n \text{ and } p\text{-type} \in \{NAND, NOR\} \\ R_p = x \wedge S_p \neq x \qquad p\text{-type} \in \{INV, BUF\} \end{cases} \tag{5}$$

$$VAL_p \leftarrow \begin{cases} S_p / \neg S_p & \text{if } DIMP_p = 1 \text{ and p is BUF/INV} \\ x & \text{if } DIMP_p = 0 \text{ and p is INV/BUF} \end{cases} \tag{6}$$

$$VAL_p \leftarrow \begin{cases} 0 & R_p = x \wedge S_p = c \wedge [(\sum_i L_i = x) = 1] \\ 1 & R_p = x \wedge S_p = \neg c \quad p\text{-type} \in \{AND, OR\} \\ x & \text{otherwise} \end{cases} \tag{7}$$

$$VAL_p \leftarrow \begin{cases} 0 & R_p = x \wedge S_p = \neg c \wedge [(\sum_i L_i = x) = 1] \\ 1 & R_p = x \wedge S_p = c \ p\text{-type} \in \{NAND, NOR\} \\ x & \text{otherwise} \end{cases} \tag{8}$$

Fig. 7 shows the *decision* block. The D/J signal indicates the frontier type and the G-type signal indicates gate type. The G_i^c / G_i^g

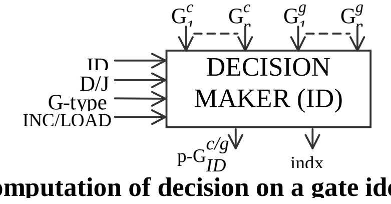


Fig. 7. Computation of decision on a gate identified with ID.

lines are inputs to gate *i*. If LOAD is set then we are processing the first decision on the gate. In this case, the inputs to this gate are supplied from the forward network. In the case where INC is set then we will be making the next decision on the gate. In this case, the previous decision values are loaded from Fanin(G,l). Next decision is computed after loading and index is set to the location where reverse decision may be made.

3. Results

To compute the efficiency of this approach, we compare the run time of a software based D-Algorithm with that of a hardware implementation. For the hardware implementation we include the following parameters: Preprocessing time (T_P), Reconfiguration time (T_C) and test generation time (T_G). The test generation time (T_G) is computed in terms of the number of faults (F), the number of direct parallel implications (I), the number of assignments (A) (i.e., the number of frontiers' selections) and the number of backtracks (B). The test generation procedure generates a test for each testable fault and does not fault simulate these vectors. For each fault, the test generation circuit requires 3 clock cycles to initialize and inject the fault. During the execution of the algorithm, the circuit requires 2 clock cycles to process direct implications, 2 clock cycles to process an assignment, and three clock cycles to process a backtrack. Thus, the total test generation time required by the generated circuit is given by $T_G = 3*F + 2*I + 2*A + 3*B$. Therefore, the total time to finish the test generation for all faults is $T_P + T_C + T_G$.

To perform the experiment we use the ISCAS85 benchmark. Characteristics of these circuits are shown in Table 2. For example, circuit c432 consists of 160 gates and a total of 524 faults were injected into this circuit. When all faults are targeted using the D-Algorithm, the number of assignments, of backtracks and of implications are 60366, 5980 and 2350 respectively. These numbers are used to compute the number of clock cycles required by the circuit.

Table 2. Descriptions of the benchmark circuits.

	# Faults	#Gates	#Assign	#Back	#Imp
C432	524	160	60366	5980	2350
C880	942	383	55624	4740	1893
C1908	1879	880	252605	13135	10301
C6288	7744	2416	939128	80131	21452
C7552	7550	3513	833121	90235	33452

Table 3 shows for each of the circuits the time required by a software D-algorithm, the preprocessing time, the time required by the hardware running at 1 MHz and the speed up. In speed-up computation, the configuration time is assumed 10 sec. Also, although most of the implications are computed in parallel in our approach, we count them sequential in our approximation. For these we can see a speed-up over software ranging between 1.3 and 3.2 times for large circuits.

Table 3. The speedups for benchmarks.

	Soft CPU (sec)	T_p (sec)	Hardware (T_G) (1 MHz)	Speed-up
c432	2.34	2.3	0.144	0.188
c880	2.8	4.7	0.132	0.188
c1908	27.64	9.2	0.57	1.398
c6288	113.02	37.3	2.184	2.281
c7552	183.02	43.5	2.026	3.296

4. Conclusion

We presented a new approach for generating test vectors that detects faults in combinational circuits. The approach is based on automatically designing a circuit which implements the D-algorithm, an Automatic Test Pattern Generation (ATPG) algorithm, specialized for the combinational circuit. Our approach exploits fine-grain parallelism by performing the following in three clock cycles: direct backward/forward implications, conflict checking, selecting next gate to propagate fault or to justify a line, decisions on gate inputs, loading the state of the circuit after backup. We showed the feasibility of this approach in terms of speed, and how it compares with software based this approach in terms of speed, and how it compares with software based techniques. For large circuits, we achieve high speed-up.

5. References

[1] M. Abramovici and D. G. Saab, "Satisfiability on Reconfigurable Hardware," *Seventh Intn'l. Workshop on Field Programmable Logic and Applications*, Sept. 1997.

[2] M. Abramovici, M. A. Breuer and A. D. Friedman, Digital Systems Testing and Testable Design, *IEEE Press*, 1994.

[3] M. Butts, J. Bacheler, and J. Varghese, "An Efficient Logic Emulation System," *ICCD-92*, pp. 138-141.

[4] K.-T Cheng, S.-Y Huang, and W.-J Dai, "Fault Emulation: A New Approach to Fault Grading," *ICCAD-95*, pp. 681-686.

[5] M. Abramovici and P. Menon, "Fault Simulation on Reconfigurable Hardware," *Proc. IEEE Symp. On Field-Programmable Custom Computing Machines*, April 1997.

[6] T. Suyama, M. Yokoo, and H. Sawada, "Solving Satisfiability Problems on FPGAs," *Proc. Intn'l. Workshop on Field-Programmable Logic and Applications*, 1996.

[7] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. On Computers*, Vol. C-30, No.3 pp. 215-222, March 1981.

[8] F. Kocan and D. G. Saab, "Dynamic Fault Location for Sequential Circuits on Reconfigurable Hardware," *IEEE ICCD-98*, pp. 214-215.

[9] H. Fujiwara and T. Shimono, "On the accelaration of test generation algorithms," *IEEE Trans. on Computers*, vol. C-32, pp. 1137-1144, Dec. 1983.

[10] T. Kirkland and M. R. Mercer, "A topological search algorithm for ATPG," *DAC-87*, pp. 502-508.

[11] M. H. Schulz, E. Trischler, and T. M. Sarfert, "SOCRATES: A highly efficient automatic test pattern generation system," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 126-137, Jan. 1988.

[12] J. P. Roth, W. G. Bouricius and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Trans. On Electronic Computers,* Vol. EC-16, No. 10, pp. 567-579, Oct. 1967.

[13] H. Fujiwara and S. Toida, "The complexity of fault detection: An approach to design for testability," in *Proc. 12th Int. Symp. Fault-Tolerant Compt.*,pp.101-108,June 1982.

[14] W. Kunz and D. K. Pradhan, "Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits," *ITC-92*, pp. 816-825.

[15] F. Kocan and D. G. Saab, "Dynamic Fault Diagnosis on Reconfigurable Hardware," *ACM/IEEE DAC-99*, pp. 691-696.

[16]Y.Matsunaga and M. Fujita, "A fast test pattern generation for large scale circuits," *SASIMI*, pp. 263-271, 1992.