Bivalent logic in Xfault simulators

Pavlinka Radoyska and Kamen Fillyov
College of Energetic and Electronics at Technical University of Sofia
8 Kl. Ohridski Blvd
Sofia 1000, Bulgaria
pradoiska@abv.bg, kfillyov@ecad.tu-sofia.bg.

ABSTRACT: *We have introduced the concurrent X fault simulator in this work. It is found that many issues are identified with the fault nature of X fault model and rules. The main issue is the problem of using bivalent logic for bad gates. We have fixed this issue by the use of trivalent logic and bit presentation. Another issue is the particular fault propagation rules based one. One more issue is that Xfault version generation. We have solved the issue by a set of using a sub-gate for every bad gate. We have solved the issue of Xfault source line presentation with the use of binary arrays for source lines description.*

Keywords: X-fault Model, VLSI Fault Simulator, Concurrent Fault simulation, Algorithm, Bivalent Logic, Trivalent Logic, Bitwise operations

Received: 11 March 2022, Revised 28 June 2022, Accepted 7 July 2022

1. Introduction

X-fault model is described in [1], [2] and [3]. This is a multiple fault model. It is proper for modeling some complex defects such as Byzantine defect, bridge defect and others. Each X-fault is described by the gate name, the version and the set of source lines. X-fault is injected on each gate output (figure 1). X-fault versions are different for each output branch. The set of source lines includes one element - the branch line. X-fault has the single fault nature, if the gate has one output line, and multiple fault nature, if the gate has fanout on the output. Fault propagation rules are described by the equations from (1) to (12).

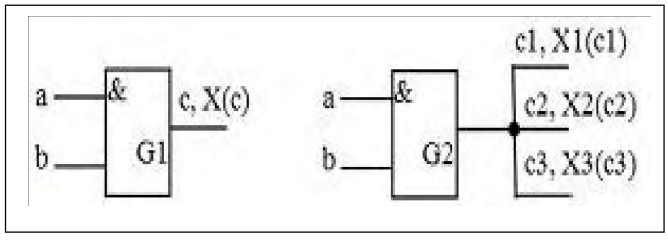

Figure 1. X-fault injection for fanout-free and fanout gates

$$NOT(Xi(b)) = \overline{X}i(b) \quad (1)$$

$$AND(Xi(b), 0) = 0 \quad (2)$$

$$AND(Xi(b), 1) = Xi(b) \quad (3)$$

$$AND(Xi(b), \overline{Xi}(b)) = 0 \quad (4)$$

$$OR(Xi(b1), 0) = \overline{Xi}(b1) \quad (5)$$

$$OR(Xi(b1), 1) = 1 \quad (6)$$

$$OR(Xi(b), \overline{Xi}(b)) = 1 \quad (7)$$

$$XOR(Xi(b), 0) = \overline{Xi}(b) \quad (8)$$

$$XOR(Xi(b),1) = \overline{Xi}(b) \quad (9)$$

$$XOR(Xi(b), \overline{Xi}(b)) = 1 \quad (10)$$

$$XOR(Xi(b), \overline{Xi}(b)) = 0 \quad (11)$$

$$G(X1(B1), X1(B2),.., Xk(Bk)) = X_{k+1}(\bigcup_{i=1}^{k} Bi) \quad (12)$$

Many authors works on developing effective simulation algorithms [5][6][7], based on stuck-at fault model. The most efficient of them is concurrent fault simulator. Concurrent simulation algorithm is based on fast bivalent logic and eventdriven principles. Fault-free simulations are performed on circuit model. Each fault-free gate is called good gate. A set of bad gates is associated with each good gate. Bad gates correspond to the faults, observable on the input lines and simulate the faulty behavior. One bad gate corresponds to one fault. A gate is called bad if at least one line (input or output) has different value from the corresponding line in the good gate. A bed event is generated on the gate output if the output signal level for the bad gate is different from the output signal level for the good gate. Bad gates, which generate bad events on primary outputs, determine the set of faults, detected by applied input vector. The full calculations for the good gates and for the set of bad gates are performed for first simulation cycle. Only gates (good or bed) with input lines, affected by an event are recalculated on the next simulation cycles. Event is called every change of line signal value.

X-fault propagation principles, have analytic nature and there are deductive approach is the closest to this description. Deductive X-fault simulator is presented in previous papers. Another type of X-fault simulator - parallel X-fault simulator, is developed by Tallinn University of Technology, Estonia [8]. Our aims are to develop X-fault simulator of concurrent type. Developing the concurrent X-fault simulator is supported by some problems. The first problem comes from more complex propagation rules and impossibility to use simple bivalent logic. The second problem comes from the multiple fault nature and necessity to define unique fault version identifier. The third problem comes from requirement to make union between the source lines sets. We try to resolve these problems in the next section.

2. Problems and Decisions

As described above, a set of bad gates must be associated to each good gate (figure 2). Every bad gate corresponds to one X-fault. The good gate is G. On its input lines are detectable three X-faults: A, B and C. Associated bad gates are three: GA for X-fault A, G-B for X-fault B and G-C for X-fault C.

2.1. Bivalent logic

Effectiveness and fastness of the classic concurrent simulator is comes especially from bivalent logic and fast bitwise operations. If we replace signals for every faulty lines in bad gates (figure 2) with opposite value for gate G-A will get AND(0,0,1)=0, for gate G-B will get AND(1, 1, 1)=1 and for gate G-C – AND(1, 1, 0) = 0. Output signal level for G-A is 0, which means that this

gate bad does not generate bad event. This behavior corresponds to the X-fault propagation rules. Output signal level for G-B is 1, which means that this bad gate generates bad event. This behavior corresponds to the Xfault propagation rules too. Output signal level for G-C is 0, which means that this bad gate does not generate bad event. This behavior is in conflict with the X-fault propagation rules. Therefore bivalent logic is not applicable for the X-fault model.

Problem can be solved by using trivalent logic in combination with bitwise operations. Signal values are 3 types: 0 and 1 for fault-free inputs and X, for faulty signal level. Bad gate generates bad event if output signal level is X. Bad gates behavior in trivalent logic are described as follows: for G-A – AND(X,0,1)=0; for G-B – AND(1,X,1)=X; for G-C – AND(1,X,X)=X. Hence, bad gate G-A does not generate bad event and bad gates G-B and G-C generate bad events.

Trivalent values can be presented by two bits to use the fast bitwise operations. Value "0" is presented by (00), value "1" – by (11) and value "X" – by the rest two combinations: (01) and (10). Trivalent logic and bitwise operations for bad gates of AND, OR, NAND and NOR types are presented by equations (13)-(24). Output X values for normal gates are presented by (01). Output X values for inverted gates are presented by (10). This is the reason to use two binary presentations for *X* value.

$$\mathrm{AND}((00),(01)) = (00) \equiv \mathrm{AND}(0, X) = 0 \quad (13)$$
$$\mathrm{AND}((11),(01)) = (01) \equiv \mathrm{AND}(1, X) = X \quad (14)$$
$$\mathrm{AND}((01),(01)) = (01) \equiv \mathrm{AND}(X, X) = X \quad (15)$$
$$\mathrm{OR}((00),(01)) = (01) \equiv \mathrm{OR}(0, X) = X \quad (16)$$
$$\mathrm{OR}((11),(01)) = (11) \equiv \mathrm{OR}(1, X) = 1 \quad (17)$$
$$\mathrm{OR}((01),(01)) = (01) \equiv \mathrm{OR}(X, X) = X \quad (18)$$
$$\mathrm{NAND}((00),(01)) = (11) \equiv \mathrm{NAND}(0, X) = 1 \quad (19)$$
$$\mathrm{NAND}((11),(01)) = (10) \equiv \mathrm{NAND}(1, X) = X \quad (20)$$
$$\mathrm{NAND}((01),(01)) = (10) \equiv \mathrm{NAND}(X, X) = X \quad (21)$$
$$\mathrm{NOR}((00),(01)) = (10) \equiv \mathrm{NOR}(0, X) = X \quad (22)$$
$$\mathrm{NOR}((11),(01)) = (00) \equiv \mathrm{NOR}(1, X) = 0 \quad (23)$$
$$\mathrm{NOR}((01),(01)) = (10) \equiv \mathrm{NOR}(X, X) = X \quad (24)$$

The main purpose of trivalent logic is to determine whether there is a bad event on the bad gate output or not. This logic is not applicable for XOR/NXOR gates. XOR(X,X) must be X, but XOR((01),(01))=(00). The problem will be discussed in the next sub-section.

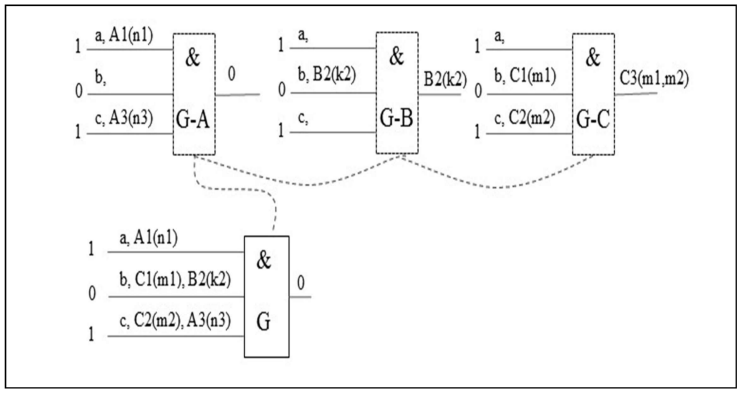

Figure 2. Gate description for X-fault concurrent simulator

2.2. X-fault Version Identifier Generation

Calculations for the new X-fault version identifier (vID) are more complex and time consuming. There are 3 subproblems:

a) How to generate unique version ID for every new combination of input X-fault versions;

b) How to generate the same version ID for the same combination of input X-fault versions;

c) How to determine if there is a dominant X-fault version and decide to use it as output X-fault version.

The first approach for new vID generation is to use integer value and to care highest version ID (hvID). The value of dvID is increase by one for any new version. This approach is very fast, but does not satisfy sub-problems b) and c). A circuit segment is shown on fig.3. Fault version 3 is the same as fault version 5, but it is not visible by vID. This approach does not permit to associate the same vID for X3(l1,l2) and X5(l1,l2). Therefore we give up first approach.

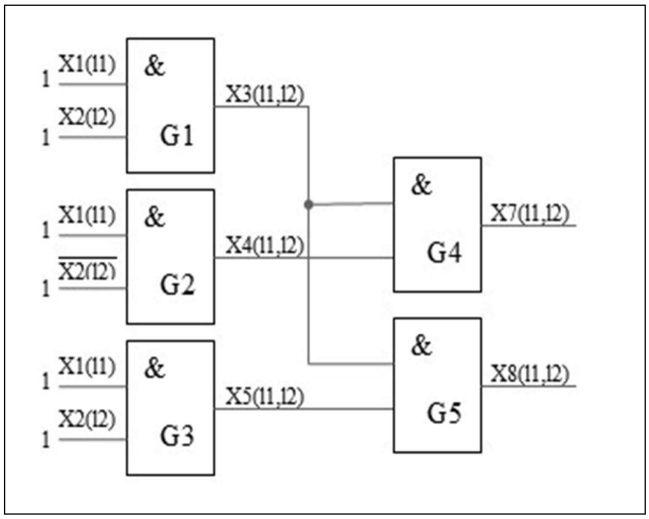

Figure 3. X-fault Version Generation

The second approach is to use string values for vID and generate it by concatenation the source versions. In this manner the vID for faults X3 and X5 will become identical: vID=”12”. But if the new version generates only by source versions concatenation, the vID for X4 will look like the same as the faults X3 and X5: vID=”12”. This is the reason to add symbol “i" before every X-fault version ID, which has inversion flag. The vID for X4 will become “1i2”, which is different.

Concatenation approach follows to two other problems: versions order problem and duplication problem. Versions “12” and “21” are represent by different strings, but actually they are the same. The set of input vID must be sorted before concatenation to avoid version order problem. The vID for X7 (firg.3) is “12”, according to the X-fault propagation rules, but after concatenation vID becomes “1212”. Problem can be solved by reducing the duplicate versions after sorting and before concatenation. The duplicate versions are neighbors in ordered set and reducing process is not time consuming.

It is more complex operation to find dominant X-fault versions, received by equations (4), (7), (10) and (11). There are analytical and bivalent approaches for solving the problem. Analytical approach is close connected to describe above solution for solving duplicate version name problem. Inverse flags are compared for every pair of identical versions. If flags are the same, the second version name is extracted from the set. If flags are different, there is a dominant version.

The second approach has concurrent fault simulation nature. If a bad gate generate bad event, for each X-fault version is created a new sub-bad gate. Signal level for each fault free input line is the same as in good gate. Signal level for each faulty input line with the same fault version is X(01). Signal levels for other input lines are recessive for the gate type – 1 for AND and NAND gates and 0 for OR and NOR gates. Duplicate versions are affected the same sub-bad gate with X value on the gate output. The version is dominant if output signal level for the sub-bad gate is opposite of the good gate. The output version is dominant if there is a dominant version among the sub-bad gates. Otherwise output version is obtained by applying the concatenation between sub-bad gates versions. Sub-bad gates must be ascending sorted by version ID. A gate with input vector (111) is shown on fig.4. Faultfree output signal is 1. Two X-faults are cause effect on the gate – A and C. Two bad gates are associated with a good gate. The two bad gates generate bed events.

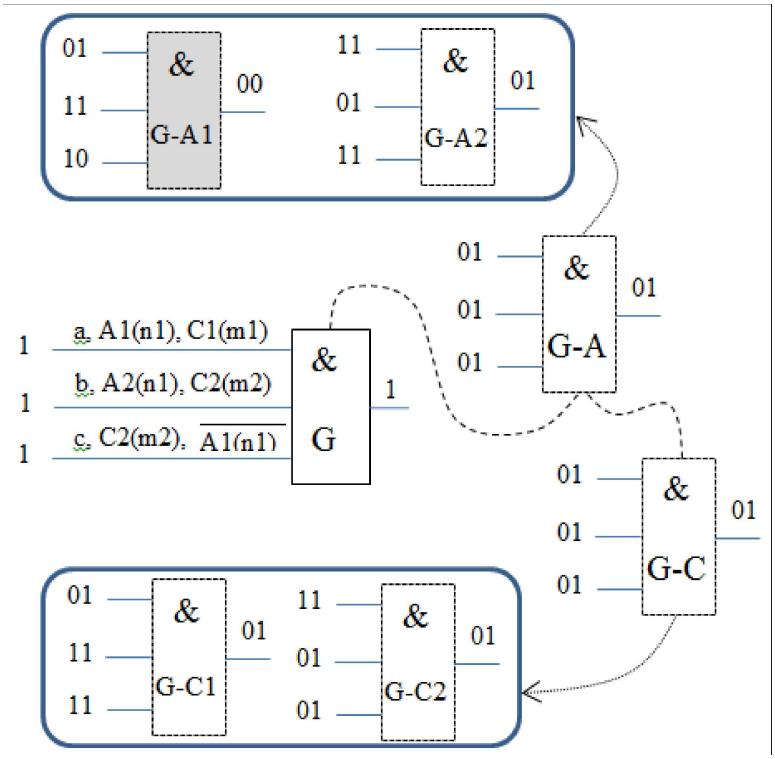

Figure 4. Sub-bad gates for AND-type gate

X-fault A has two versions – "1" and "2". Two sub-bad gates (G-A1 and G-A2) are generated for this bad gate. Version "1" cause effect on lines a and c. Input signal level for line a is (01) and for line c – (10), because there is an inversion. Input signal for line b is (11). Output signal for subbad gate G-A1 is (00) and therefore this is a dominant fault version.

X-fault C has also two versions – "1" and "2". Two sub-bad gates (G-C1 and G-C2) are generated for this bad gate. Version "1" cause effect on line and its input signal is (01). Input signals for lines b and c are (11). Output signal for subbad gate G-C1 is (01) and therefore this is a normal fault version, which will be included in concatenation for new vID. Version "2" cause effect on lines b and c and their input signals are (01). Input signal for line a is (11). Output signal for sub-bad gate G-C2 is (01) and therefore this is a normal fault version, which will be included in concatenation for new vID.

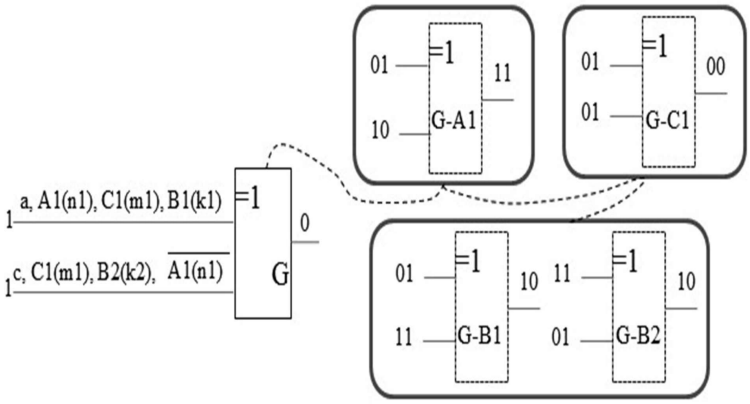

Figure 5. Sub-bad gates for AND-type gate

In this way, the duplication and dominant problems are solved in a natural way with very fast bitwise operations. Moreover, this approach gives solution for the XOR problem, discussed in section A. Only sub-bad gates are generated for XOR and NXOR gates. Duplicate input X-fault versions follow to value (00) on the gate output, independently of input signal levels. Mutually inverse X-fault versions follow to value (11) on the gate output, independently of input signal levels. Dominant X-fault version is determined only after comparison with the output signal for good gate. Input vector for the gate on fig. 5 is (11). Input X-faults are A, B and C. Xfault A cause the two input lines with the same vID, but with inversion on line c. Therefore one sub-bad gate G-A1 is created for fault A. Output line level for G-A1 is (11), which is opposite then the good gate. Therefore there is a bad event and dominant fault version. X-fault C cause the two input lines with the same vID. Therefore one sub-bad gate G-C1 is created for fault C. Output line level for G-C1 is (00), which is the same as for the good gate. Therefore there is not a bad event. X-fault B cause the two input lines with different vIDs. Therefore two sub-bad gates (G-B1 and G-B2) are created for fault B. Output line level for the two sub-bad gates is (01), which means that the two versions generate bad events and vID on the gate output is "12". If The input vector for fig.5 is (01), output signal for good gate will be 1, output signals for sub-bad gates G-A1 and G-C1 will be the same. Therefore GC1 will become dominant version and G-A1 will not generate bad event. Output signals for G-B1 and G-B2 will be (01), which is again X-value.

In conclusion, the best solution for X-fault version ID value type is string type. The best solution for version ID generation is to generate sorted list of sub-bed gates for each version and concatenate version IDs, if there are not dominant version.

2.3. Union operations for set of X-faults base lines

There are two solutions for description the source lines for current X-fault version. Description by string values, which are the lines names, is used in the first solutions. An array of source lines' names is associated with every X-fault version. The union of source lines makes by multiple scanning the base arrays, which will slow down the simulator.

Description by binary values is used in the second solutions. Source lines for any X-fault version are described by a binary array. The size of array is the number of fanout branches on the output of the gate, for which X-fault is injected. The bit, corresponding to the faulty source branch number has true value, others have false value. Each binary array has only one true value on fault injection stage. The union of two base lines' sets comes down to bitwise AND.

For example let us see the gate on fig. 1.b). Gate has three output branches. Therefore the source lines array for this Xfault has three elements. Injected faults are X"1"(100) for line c1, X"2"(010) for line c2 and X"3"(001) for line c3. Source lines union for equation (25) is presented by binary arrays in equation (26). Source lines union for equation (27) is presented by binary arrays in equation (28). Source lines union for equation (29) is presented by binary arrays in equation (30).

$$\text{AND}(X\text{"1"}(c1), X\text{"2"}(c2)) = X\text{"12"}(c1, c2) \quad (25)$$

$$\text{AND}(X\text{"1"}(100), X\text{"2"}(010)) = X\text{"12"}((100) \,\&\, (010)) == X\text{"12"}(110) \quad (26)$$

$$\text{AND}(X\text{"1"}(c1), X\text{"3"}(c3)) = X\text{"13"}(c1, c3) \quad (27)$$

$$\text{AND}(X\text{"1"}(100), X\text{"3"}(001)) = X\text{"13"}((100) \,\&\, (001)) == X\text{"13"}(101) \quad (28)$$

$$\text{AND}(X\text{"12"}(c1,c2), X\text{"13"}(c1,c3)) = X\text{"1213"}(c1, c2, c3) \quad (29)$$

$$\text{AND}(X\text{"12"}(110), X\text{"13"}(101)) = X\text{"1213"}((110) \,\&\, (011)) == X\text{"1213"}(111) \quad (30)$$

The second approach is time and memory consuming and we have embedded it in the concurrent X-fault simulator.

3. Experimental Results

Concurrent X-fault simulator is developed on C# and .NET framework as web-based tool. The deductive X-fault simulator has been presented in previous papers. Simulations are performed with the two simulators on the same benchmark circuits and test sets. Measured simulation time for the deductive simulator (DS) and for the Concurrent simulator (CS) are presented in table 1. Abbreviation CS/DS indicates the simulation time ratio of Concurrent simulator to Deductive simulator. The results show that the Concurrent simulator is more effective than the Deductive simulator.

Experiments are made on computer with 3 GB RAM, Pentium Dual-Core CPU T440 2.20GH and 32-bit operating system. The results are indicative, despite the weak performance of the machine, because the relative, not absolute results are important.

Circuit	DS (sec.)	CS (sec.)	CS/DS
C17	0.0109	0.0099	0.91
74L85	0.0183	0.0162	0.88
74181	0.0192	0.0163	0.85
74182	0.0047	0.0038	0.80
74283	0.0090	0.0066	0.73
C432	0.1186	0.0903	0.76
C499	0.1140	0.0964	0.85
C880	0.2616	0.2129	0.81
C1908	1.2350	0.9594	0.78
C1355	4.6758	4.3871	0.94
C3540	6.5588	5.1301	0.78

Table 1. Simulation Times For Deductive and Concurrent Simulators

4. Conclusions

The main goal of this paper is to discuss the problems, related to development of concurrent simulator for X-fault model and to offer the best solutions for them. Proposed concurrent simulator uses trivalent logic for bad gates simulations. Each value is presented by two bits and performances are implemented by fast bitwise operations. A bad gate generate bad event if output signal level is *X*. Subbad gates, working in trivalent logic, are used for any X-fault version. X-fault source lines are described by binary values and lines' sets union is performed as bitwise AND.

References

[1] Huisman, L.(2004). Diagnosing Arbitrary Defects in Logic Designs Using Single Location at A Time (SLAT), *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, 2004, pp. 91–101

[2] Wen, X., Miyoshi, T., Kajiihara, S., Wang, L., Saluja, K., Kinoshita, K. (2004). On per-test fault diagnosis using the Xfault model. *In Int'l Conf. on CAD*, 2004, pp. 633–640.

[3] Polian, I., Miyase, K., Nakamura, Y. , Kajihara, S., Engelke, P., Becker, B., Spinner, S., Wen, X. (2008). Diagnosis of Realistic Defects Based on the X-Fault Model, *11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, DDECS 2008., pp.1-4

[4] Zhang, Y., Guan, Y., Wang G.(2009). Analysis and Comparison of Fault Simulation, *International Symposium on Intelligent Ubiquitous Computing and Education*, 2009, pp. 503 – 506

[5] Shen, Li, (2003). RTL Concurrent Fault Simulation, Proceedings of the 12th Asian Test Symposium

[6] Lu, W., M. Radetzki (2011). Efficient Fault Simulation of SystemC Designs, 14th Euromicro Conference on Digital System Design, *IEEE, Computer Society*, pp. 487-494

[7] Bosio, A., G. Natale (2008). *LIFTING: a Flexible Open-Source Fault Simulator*, 17th Asian Test Symposium, pp. 36 – 40

[8] Ubar, R., Devadze, S., Raik, J., Jutman, A. (2010). Parallel Xfault simulation with critical path tracing technique, Proceedings of the Conference on Design, *Automation and Test in Europe* DATE '10, pp. 879-884