SAT-Based Fault Equivalence Checking in Functional Safety Verification

Ai Quoc Dao, Mark Po-Hung Lin, *Senior Member, IEEE*, and Alan Mishchenko

***Abstract*—Detecting equivalence classes of injected faults for functional verification of electronic systems is an important task because it helps reducing the number of faults to qualify a verification environment, and hence, improves the performance of qualification process and the validation time required for large-scale electronic systems. This paper describes an efficient way of detecting equivalent injected faults in a mapped netlist in order to speedup the qualification process of a verification environment for functional safety. The presented fault models include general faults resulting in arbitrary functional changes, in addition to conventional stuck-at faults. The solution is based on structural pruning and functional analysis performed by a synergistic combination of iterative Boolean satisfiability and guided simulation. It should be noted that traditional brute-force-like methods would take many hours or even days to identify thousands of equivalent functional faults injected to a small circuit with only hundred of cells. The proposed approach can achieve excellent fault reduction ratios within few seconds for such small circuits. The implementation also scales well for the largest ISCAS'89 and OpenCores benchmarks containing over 35K gates and 490K general functional faults.**

***Index Terms*—Equivalence checking, equivalence classes, equivalent fault, fault collapsing, fault injection, fault reduction, functional safety, functional verification, general fault, identical fault, ISO 26262, qualification acceleration, stuck-at fault.**

I. Introduction

QUALIFICATION of functional verification environments is very important to ensure functional safety of various electronic systems, especially for automotive and Internet-of-Things applications. Failure modes and effects analysis (FMEA) is a step by step approach for identifying all possible failures in a design, a manufacturing/assembly process, a product, or service, while a broadening of FMEA, called failure modes effects and diagnostics analysis, is a detailed analysis of different failure modes and diagnostic capability for electronic systems. Fig. 1 shows a general functional safety verification environment. One of the key challenges is to qualify whether the verification environment can achieve 100% functional safety for automotive systems on chip, consisting of many intellectual properties (IP) and verification IP (VIP). The automotive functional safety verification environment involves six major components: 1) requirement management; 2) prototyping; 3) traceable verification; 4) automotive protocols and memory VIP; 5) fault injection and simulation; and 6) customized safety reports [3].

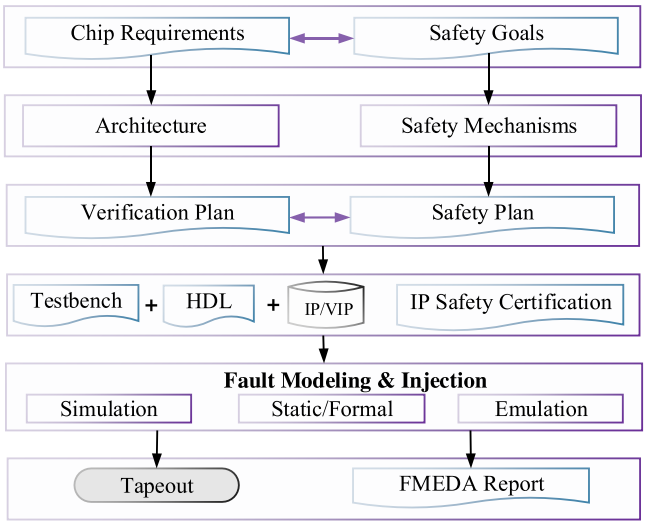

Fig. 1. Automotive functional safety verification environment.

The international standard for functional safety of electrical and electronic systems, ISO 26262 [4], is considered as the best practice framework for achieving functional safety in road vehicles. The functional safety assessments defined in ISO 26262 help designers to reach target automotive safety integrity levels. ISO 26262 recommends fault-injection testing to verify the effectiveness of the safety mechanisms [3]. A general ISO 26262 functional safety verification flow is shown in Fig. 2, which involves three major steps: 1) failure mode effects analysis; 2) fault injection campaign; and 3) metric calculation and export.

To achieve qualified safety verification, fault injection enables verification qualification to save time and effort

Manuscript received July 3, 2017; revised October 27, 2017; accepted December 20, 2017. Date of publication January 9, 2018; date of current version November 20, 2018. The work of D. A. Quoc and M. P.-H. Lin was supported by the Ministry of Science and Technology, Taiwan under Grant 104-2628-E-194-001-MY3. The work of A. Mishchenko was supported by NSF/NSA Grant "Enhanced Equivalence Checking in Cryptoanalytic Applications" at the University of California, Berkeley. This work received the 1st Place Award of Problem A "Identical Fault Search" in ICCAD'16 CAD Contest [1]. The implementation is currently available as a new command "faultclasses" in ABC [2]. This paper was recommended by Associate Editor Q. Xu. *(Corresponding author: Mark Po-Hung Lin.)*

A. Q. Dao and M. P.-H. Lin are with the Department of Electrical Engineering and Advanced Institute of Manufacturing with High-tech Innovations, National Chung Cheng University, Chiayi 621, Taiwan (e-mail: aiquocvt@gmail.com; marklin@ccu.edu.tw).

A. Mishchenko is with the Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA 94720 USA (e-mail: alanmi@berkeley.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2018.2791465

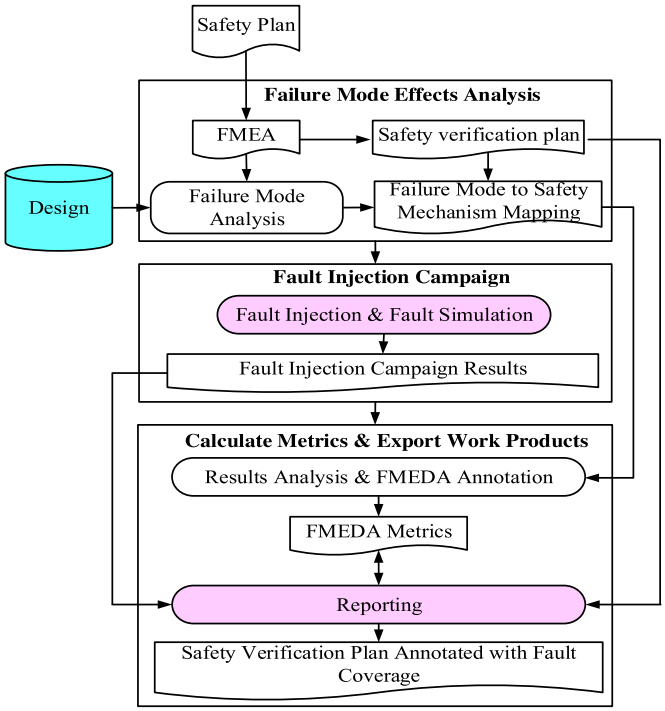

Fig. 2. ISO 26262 functional safety verification flow [3].

when testing all safety requirements of electronic systems. Verification engineers need to take their functional verification environment and essentially replay pieces of it while injecting faults into their system. For example, Sherer *et al.* [5] presented a verification flow to ensure safety compliance for ISO 26262 using simulation. Some verification environments use fault lists derived for a design to be tested. Injecting faults into designs is a way to qualify a verification environment. The design of safety systems contains redundancy and checkers. Safety engineers must implement requirement checkers tracing from the system to components, and ensure that their development flow aligns with the confidence level. However, the large number of injected faults may have equivalence. To avoid running the redundant dynamic simulations or formal checks for those equivalent faults, and hence, improve the performance of the qualification process, it is essential to remove all identical faults [6]. Detecting equivalence classes of faults is an important task to reduce the large number of injected faults.

In the testing field, detecting equivalence classes of faults, also known as fault collapsing, is an important task because it helps reduce the number of faults targeted by an automatic test pattern generation (ATPG) tool. This, in turn, helps reduce the number of test-patterns, the runtime of ATPG, and the time required for design validation and fault diagnosis. It is also important that the computation of equivalent faults is fast. Otherwise, it may substantially increase the runtime of an ATPG flow. Conventional fault collapsing methods based on either structural analysis or functional dominance techniques [7] only focus on stuck-at faults. For large-scale safety-related electrical and electronic systems, such as automobiles, higher safety standard, has been developed to control various systematic and random hardware failures, and to mitigate their effects. In addition to stuck-at faults, it is desirable to consider various general functional faults leading to behavioral and functional changes of the whole system, and hence, functional safety of these electrical and electronic systems can be guaranteed.

This paper focuses on efficient computation of classes of equivalent faults for general fault types in order to reduce the number of injected faults and speedup the qualification process of a verification environment for functional safety. The input includes a mapped netlist and a fault list, with each fault specified as a location in the netlist and a change introduced by the fault. The proposed algorithm will return the set of equivalence classes of faults, represented as a set of pairs of faults proved to be equivalent. Inspired by efficient and scalable applications, which combine circuit simulation and Boolean satisfiability [8]–[12], the proposed solution is a new algorithm for the computation of fault equivalence classes based on a tight integration of the following.

1) *Structural pruning* to efficiently rule out a large number of nonequivalent faults.
2) *Functional analysis* by the SAT solver to prove or disprove that any two faults are indeed equivalent.
3) *Circuit simulation* to refine candidate equivalence classes of faults using counter-examples produced by the SAT solver.

The use of Boolean satisfiability (SAT) in the proposed algorithm is based on the understanding that SAT is more scalable than BDDs [13] when dealing with large industrial circuits. A similar observation regarding the scalability of SAT was made in past work on combinational equivalence checking (CEC) [10]. The reason why SAT-based methods [9]–[11] typically outperform BDD-based ones [14], [15], is because BDDs are a canonical data-structure that needs to be fully constructed before it can be used, but it is often difficult or impossible to construct it for large netlists. For example, arithmetic logic, which is often present in industrial circuits, is notoriously hard for BDDs. In particular, multipliers result in exponential BDD size for any variable order [16]. Meanwhile, SAT solvers rely on a noncanonical conjunctive normal form (CNF) that can be easily constructed and used immediately after construction.

The rest of this paper is organized as follows. Section II contains necessary background. Section III gives an overview of the algorithm. Experimental results are given in Section IV. Section V concludes this paper.

II. Background

A. Boolean Function

In this paper, function refers to a completely specified Boolean function $f(X) : B^n \rightarrow B, B = \{0, 1\}$. The *support* of function f is the set of variables X, which influence the output value of f. The support size is denoted by $|X|$.

B. Boolean Network

A Boolean network (or circuit) is a directed acyclic graph (DAG) with nodes corresponding to Boolean functions and edges corresponding to wires connecting the nodes. A node

n may have zero or more fanins, i.e., nodes driving *n*, and zero or more fanouts, i.e., nodes driven by *n*. The primary inputs (PIs) are nodes without fanins. The primary outputs (POs) are a subset of nodes of the network, connecting it to the environment. A transitive fanin (fanout) cone (TFO) of a node is a subset of nodes of the network, which are reachable through the fanin (fanout) edges of the node. The TFO support of a node is the set of POs reachable through the fanouts of the node.

C. Mapped Network and Fault Description

A Boolean network can be mapped using a standard cell library. Mapping is performed by a technology mapper, a software package, which takes a technology-independent representation of the network called a subject graph and returns a mapped network. In a mapped network, each node is assigned a gate from the library. In this paper, we assume that the network is mapped. Furthermore, we do not consider any technology-dependent information associated with the gates, except their Boolean functions. A fault is a malfunction in the mapped network happening due to a manufacturing error. A fault changes functionality at an internal node or wire. The change may propagate to one of the POs and make the functionality of the whole network incorrect. A fault can be characterized by a location and a type. Fault location describes where the change has occurred. Fault type describes how the correct functionality has changed. For example, a given two-input AND-node in the network may become a two-input NAND-node, or a given wire may become stuck-at-zero (stuck-at-one), meaning that the function of the wire remains always constant 0 (constant 1) for any value of the driving node. Two or more faults are equivalent if the errors they produce at the outputs of the network are the same. In this paper, we discuss an efficient implementation of the computation of fault equivalences in a mapped network, and hence the number of injected faults for verification qualification can be reduced.

D. Boolean Satisfiability

A satisfiability problem (SAT) is a decision problem that takes a propositional formula representing a Boolean function and answers the question whether the formula is satisfiable. The formula is satisfiable (SAT) if there is an assignment of variables that evaluates the formula to 1. Otherwise, the propositional formula is unsatisfiable. A software program that solves SAT problems is called an SAT solver. When an SAT problem is satisfiable, the solver returns a satisfying assignment, which is often useful to reproduce and understand the reason of a malfunction.

E. Conjunctive Normal Form

To represent a propositional formula for the SAT solver, important aspects of the problem are encoded using Boolean variables. Presence or absence of a given aspect of the problem is represented by a positive or negative literal of the variable. A disjunction of literals is called a clause. A conjunction of clauses is called a CNF. The CNF can be processed by a mainstream CNF-based SAT solver, such as MiniSAT [17], which is used in this paper.

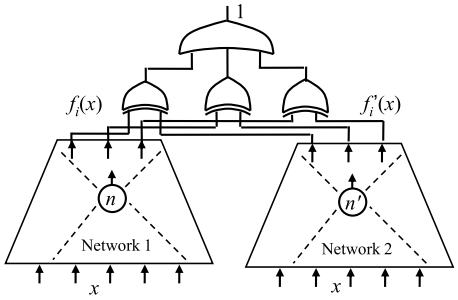

Fig. 3. Equivalence checking miter for two copies of the network with different faults injected.

F. Equivalence Checking

Two combinational Boolean networks are equivalent if and only if, for each combination of PI values, pairs of matching POs produce equal values [8]. The equivalence of two Boolean networks is checked by building another network called miter. The miter contains both Boolean networks with shared PIs, as shown in Fig. 3. The miter also contains additional gates pairwise comparing the POs of the original networks. The miter has one PO whose value is 1 if and only if at least one pair of the POs produces different values. The PI values producing value 1 at the miters output, can be used as a counter-example to demonstrate why equivalence between the original networks has failed.

To prove equivalence of two faults, we build the miter for two networks, Networks 1 and 2, derived by injecting each fault into an original fault-free network, as shown in Fig. 3. The node n in Network 1 corresponds to the injection of the first fault into the fault-free network. Similarly, node *n*' in Network 2 corresponds to the injection of the second fault into the same fault-free network. After constructing the miter, it is converted into a CNF and solved by the SAT solver, as described below.

III. Proposed Algorithm

This section describes the proposed algorithm in detail. A high-level overview of the algorithm is shown in Fig. 4. The algorithm takes a mapped netlist and a fault list containing faults considered for equivalence checking. Initially, structural pruning is used to divide all faults into candidate equivalence classes. These preliminary classes need further refinement by functional methods. To this end, SAT solving and circuit simulation are iterated for each candidate equivalence class, resulting in a set of true equivalence classes. Detailed descriptions of general fault types and all computations for checking fault equivalence are given in the sections below.

A. Fault List With General Fault Types

Each fault in a fault list is represented as a triple.

1) A unique integer number called fault ID, which is used to identify the fault in the resulting file containing the list of fault equivalence classes.
2) The name of a node where the fault occurs.
3) The type of the fault, characterizing how the node's functionality changes when the fault is present.

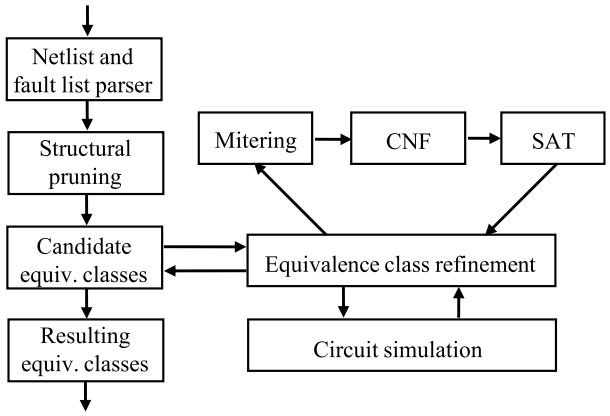

Fig. 4. Overview of the algorithm.

TABLE I
FAULT TYPES GIVEN IN [1]

Fault Type	*Description*
SA0	Stuck at 0
SA1	Stuck at 1
NEG	Negate the value of a signal
RDOB_AND	Replace driver operator by AND
RDOB_NAND	Replace driver operator by NAND
RDOB_OR	Replace driver operator by OR
RDOB_NOR	Replace driver operator by NOR
RDOB_XOR	Replace driver operator by XOR
RDOB_XNOR	Replace driver operator by XNOR
RDOB_NOT	Replace driver operator by NOT (only when the driver operator is BUF)
RDOB_BUF	Replace driver operator by BUF (only when the driver operator is NOT)

Algorithm 1 Fault Injection

Input: A network, N
Output: Injected fault list, F

```
for each primary input n in network N
    Inject_Basic_Faults(SA0, SA1, NEG);
for each primary output n in network N
    Inject_Basic_Faults(SA0, SA1, NEG);
for each internal node n in network N
    Inject_Basic_Faults(SA0, SA1, NEG);
    Inject_General_Faults_in_Table_I;
    // The general faults correspond to changing
    // the given gate by another gate with
    // the same support size from the same library.
return F;
```

A fault type can be represented in a number of ways. In the ICCAD'16 CAD Contest (Problem A) [1], in addition to stuck-at faults (SA0 or SA1) and negation fault (NEG), the predetermined fault types listed in Table I contain eight replacements of a driver operator at a node in the mapped netlist based on the specific library. Our implementation supports all these fault types, assuming that the network is mapped using the library from the contest.

In this paper, we further extend the notion of a fault type and generalize our implementation to work for a netlist mapped into any standard-cell library containing single-output combinational cells. The proposed implementation, as seen in Algorithm 1, allows not only for three basic fault types (stuck-at-0, stuck-at-1, and negation) but also for any other fault replacing a combinational cell in a library by any other combinational cell from the same library with the same number of inputs. Fig. 5 gives an example of a fault-free netlist and the faulty netlists after injecting five different faults, as shown in Table II. The three-input XOR gate, XOR3, at node n_{16}, may be substituted by a three-input OR gate, OR3, in the same library, as shown in Fig. 5(b), due to the fault type RDOB_OR3 at node n_{16}. Similarly, the four-input OR gate, OR4, at node n_{20} may be substituted by a four-input AND gate, AND4, as shown in Fig. 5(c), due to the fault type *RDOB_AND*4. If the library does not have any other three-input and four-input combinational cells, both faults cannot be represented. Fig. 5(d)–(f)

TABLE II
EXAMPLE FAULT LIST FOR THE CIRCUIT IN FIG. 5(A)

Fault ID	*Node*	*Fault Type*
1	n_{16}	RDOB_OR3
2	n_{20}	RDOB_AND4
3	n_{20}	SA0
4	n_{20}	SA1
5	n_{20}	NEG

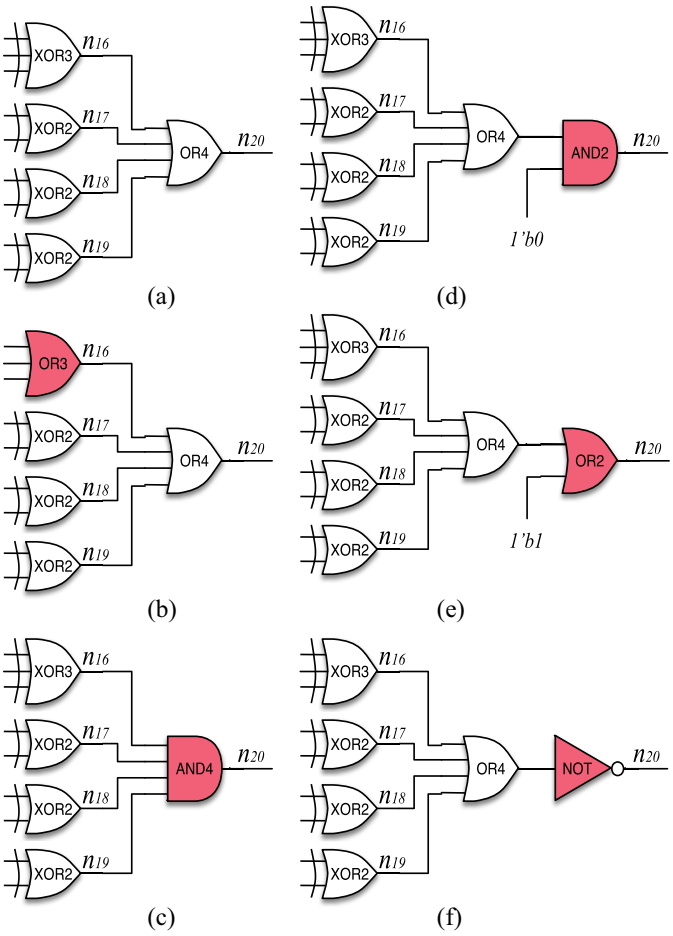

Fig. 5. Example of fault injection. (a) Fault-free netlist. (b)–(f) Faulty netlists after injecting RDOB_OR3 at n_{16}, *RDOB_AND*4 at n_{20}, SA0 at n_{20}, SA1 at n_{20}, and NEG at n_{20}, respectively.

Algorithm 2 TFO Support Computation

Input: Network N
Output: TFO support
1: *For each node n in network N*
2: *result*(n)=$\emptyset$; // *set TFO support to be empty*;
3: *For each primary output n in network N*
4: *result*(n)={n}; // *set TFO support to be the node itself*;
5: *For each internal node n in a topo order from outputs*
6: *For each fanout f of node n*
7: *result*(n)=*result*(n) $\cup$ *result*(f);
8: **return** result;

shows the faulty circuits after injecting the three basic fault types, SA0, SA1, and NEG, at node n_{20}, respectively.

B. Structural Pruning

It is helpful to recall that two faults are equivalent if and only if, for each combination of PI values, the pairs of matching POs produce the same values. Recall also that the TFO support of a node, is the set of all POs of the network which are reachable from the node through the fanout edges. The following observation helps effectively rule out nonequivalent faults.

1) Observation: If two faults have different TFO supports, these faults are likely not equivalent. The observation is motivated as follows. If a PO is in the TFO of one fault and not in the TFO of another fault, then it may be possible for this output to be affected by the first fault and not affected by the second fault. Hence, these two faults are not equivalent. There is a rare situation when this observation is not true.

2) Situation: The TFO support is determined by the structure instead of the functionality of the network. As a result, functional dependency between a target node and a PO may not exist, even if there is structural dependency evidenced by the TFO support. When this happens, two faults are placed into different candidate equivalence classes based on the structural support information, while in fact they belong to the same class. This situation is not considered by our current heuristic algorithm. As a result, a small number of truly equivalent faults may be classified into different candidate equivalence classes by the current algorithm. We believe that the resulting minor loss in quality is compensated by a substantial reduction in runtime, which makes the algorithm applicable to large industrial netlists. The computation of exact equivalence classes of faults will be addressed as part of future work.

The above observation leads to the following strategy for structural pruning. First of all, faults found in a fault list are divided into candidate equivalence classes according to their TFO supports. Each class contains all faults whose TFO supports are the same. Next, each equivalence class is processed by the SAT-based functional analysis to find classes of faults that are truly equivalent. The computation of TFO supports is performed by a dynamic programming algorithm with the pseudo-code in Algorithm 2. The algorithm considers nodes in a topological order from POs to PIs. At each node, its TFO support is computed by unioning the TFO supports of its fanout nodes.

C. Equivalence Class Refinement

Since the candidate equivalence classes obtained from structural pruning may not be true equivalence classes, Boolean satisfiability and simulation are used to prove or disprove the true equivalence between any pair of faults in the same candidate equivalence class. If the equivalence of two faults is disproved, the nonequivalent faults have to be separated into different equivalence classes. For each candidate equivalence class, a pair of faults, which have not yet been proved or disproved, is selected. An equivalence checking miter is then constructed, as described in Section II-F and shown in Fig. 3. The miter is converted to a CNF and loaded into an SAT solver. The solver checks equivalence and returns one of the three outcomes: 1) unsatisfiable, meaning that the two faults are equivalent; 2) satisfiable, meaning that the faults are not equivalent; and 3) undecided, meaning that the two faults could not be proved or disproved. In the latter case, the faults are assumed to be not equivalent. A test-pattern is generated by the SAT solver if two faults are not equivalent. In order to speedup equivalence class refinement, circuit simulation is performed with the generated test-pattern for every fault pairs in the same candidate equivalence class. This may result in splitting the class into several smaller candidate equivalence classes. Sometimes two faults cannot be proved or disproved because the SAT solver times out. In this case, one of the two faults is placed in a separate class without simulation. According to our experimental study, this situation happens rarely, but it is important to handle it to improve robustness of the software system.

D. Management of Equivalence Classes

Candidate equivalence classes are stored in a 2-D array. An equivalence class of faults is an array of integer fault IDs, while a set of equivalence classes is an array of arrays of integers representing equivalence classes. During the refinement of equivalence classes, larger arrays of fault IDs are repeatedly split into smaller ones. The splitting takes place after simulation of a test-pattern generated by the SAT solver, which happens only when the equivalence of two faults is disproved. When the splitting takes place, faults in an equivalence class are divided into two parts: 1) those detected by this pattern and 2) those not detected by it. In our current implementation, the faults in 1) are left in the same array, while the faults in 2) are moved to a new array that is appended at the end. In due time, refinement will reach the new array, which may be further refined, if needed. The pseudo-code of the algorithm described in Sections III-C and III-D is given in Algorithm 3.

IV. Experimental Results

The proposed algorithm was implemented in the C programming language and tested on an Intel Xeon CPU@2.4 GHz with 48 GB memory. The implementation is currently available as a new command "faultclasses" in ABC [2]. Experimental results have been conducted using the following three sets of benchmarks: 1) ICCAD'16 CAD Contest benchmarks; 2) ISCAS'89 benchmarks; and 3) OpenCores benchmarks.

TABLE III
COMPARING THE PROPOSED ALGORITHM AGAINST THE BRUTE-FORCE METHOD AND WITHOUT STRUCTURAL PRUNING ON ICCAD'16 CAD CONTEST BENCHMARKS

Circuit	*Circuit information*					*Brute-force method [1]*		*Our algorithm w/o structural pruning*		*Our algorithm*	
name	*# PIs*	*# POs*	*# FFs*	*# cells*	*# faults*	*# resulting faults (fault reduction ratio %)*	*Time (sec)*	*# resulting faults (fault reduction ratio %)*	*Time (sec)*	*# resulting faults (fault reduction ratio %)*	*Time (sec)*
Case 1	35	32	0	206	1568	1097 (30.0%)	Many hours	1097 (30.0%)	0.2	1101 (29.8%)	0.1
Case 2	35	32	0	242	1848	1313 (29.0%)	Many hours	1313 (29.0%)	0.3	1315 (28.8%)	0.1
Case 3	11	12	7	162	1204	583 (51.6%)	Many hours	666 (44.9%)	1.0	673 (44.1%)	1.1
Case 4	43	40	0	316	2416	1530 (36.7%)	Many hours	1628 (32.6%)	0.6	1637 (32.2%)	0.3
Case 5	46	43	0	401	3098	2029 (34.5%)	Many hours	2029 (34.5%)	0.8	2044 (34.0%)	0.4
Case 6	13	14	16	648	5008	1107 (77.9%)	Many hours	1592 (68.2%)	53.1	1620 (67.7%)	28.6
Case 7	17	18	16	567	4352	1341 (69.2%)	Many hours	1853 (57.4%)	15.7	1870 (57.0%)	15.0
Avg.					**1**	**0.530 (47.0%)**		**0.576 (42.4%)**	**1.8x**	**0.580 (42.0%)**	**1**

TABLE IV
COMPARING THE PROPOSED ALGORITHM AGAINST THE OTHER WINNERS OF ICCAD'16 CAD CONTEST [1]

Circuit	*Circuit information*					*Team A (the 3^{rd} Place)*			*Team B (the 2^{nd} Place)*			*Our algorithm (the 1^{st} Place)*		
name	*# PIs*	*# POs*	*# FFs*	*# cells*	*# faults*	*Correctness (%)*	*Mem. (MB)*	*Time (sec)*	*Correctness (%)*	*Mem. (MB)*	*Time (sec)*	*Correctness (%)*	*Mem. (MB)*	*Time (sec)*
Case 5	46	43	0	401	3098	99.80%	19	1.03	100.00%	3	0.61	99.23%	7	0.42
Case 6	13	14	16	648	5008	79.27%	42	13.14	81.38%	14	41.65	88.23%	11	28.60
Case 7	17	18	16	567	4352	76.97%	37	41.23	81.37%	40	66.03	85.15%	16	14.99
Case 8	n/a	n/a	n/a	n/a	n/a	80.92%	53	34.57	82.20%	5	36.63	92.22%	9	31.13
Avg						**84.24%**	**3.7x**	**1.7x**	**86.24%**	**1.2x**	**2.1x**	**91.21%**	**1**	**1**

Algorithm 3 Equivalence Class Refinement

Input: candidate equiv. classes E
Output: candidate equivalence classes

```
For each candidate equivalence class C in E
  repr = C[0]; //consider C[0] as representative
  for each fault d in C such that d ≠ repr
    status = Check_Using_SAT(d, repr);
    if(status == unsatisfiable)
      continue; // keep fault d where it is in array C
    assert(status == satisfiable || status == undecided);
    if(status == satisfiable)
      cex = Generate_Test_Pattern(d, repr);
      Resimulate_Class_With_Test_Pattern(C, cex);
    Refine_Class(C)
    // The refinement is expected to happen in C.
    // After refinement, the algorithm will keep faults
    // that are similar to repr under cex in C, and
    // move other faults to a new array appended to E.
return result;
```

Motivated by ICCAD'16 CAD Contest [6], this paper focuses on detecting as many identical injected faults as possible to avoid running redundant simulations or formal checks for those identical faults in the verification environment for functional safety. The contest organizers provided eight mapped netlists and the corresponding fault lists, while only cases 1–7 were made public. Table III lists benchmark statistics, including the numbers of PIs, POs, flip-flops (FFs), combinational cells, and injected faults in the given fault lists. We first compare our algorithm against the brute-force method in terms of the numbers of computed equivalent faults and runtime.

The brute-force method used by the contest organizers first collects all potential equivalent fault pairs and then checks whether the two faults in each pair are equivalent using a commercial formal tool. Although such approach computes all equivalent faults in a fault list resulting in the largest fault reduction ratios, it takes many hours or even days for circuits with only hundreds of cells and thousands of faults. Our heuristic algorithm can efficiently identify most of the equivalent faults within a few seconds, while the average fault reduction ratio is only 5% worse than that of the exact brute-force method.

We additionally show the results of our heuristic algorithm without structural pruning in Table III. Without structural pruning, although the number of computed equivalent faults is slightly better than our algorithm, the run time takes 1.84× longer than our algorithm with structural pruning.

We further compare the proposed algorithm against the other winners of ICCAD'16 CAD Contest. According to [1], only cases 5–8 were applied to evaluate the correctness, memory usage, and runtime resulting from different teams of the contest participants. Table IV compares our algorithm against the other two winners, Teams A and B, who received the 3rd and 2nd place awards in the contest, respectively, in terms of the correctness, memory usage, and runtime. The comparisons are publicly available at [1]. The correctness is evaluated according to the number of identified equivalent fault pairs divided by the number of exact equivalent fault pairs. The exact equivalent fault pairs for each benchmark were collected by the contest organizers with a brute-force method which checks whether the two faults in each pair are equivalent using a commercial formal tool. Our algorithm can generate 7% and 5% more

TABLE V
COMPARING THE PROPOSED ALGORITHM AGAINST A STATE-OF-THE-ART COMMERCIAL EDA TOOL FOR SA FAULTS ONLY

	Circuit information						*Our algorithm for general faults*		*TetraMAX ATPG for SA faults only*	*Our algorithm for SA faults only*	
Circuit name	*# PIs*	*# POs*	*# FFs*	*# cells*	*# general faults*	*# SA faults*	*# resulting general faults (reduction ratio)*	*Time (sec)*	*# resulting SA faults (reduction ratio)*	*# resulting SA faults (reduction ratio)*	*Time (sec)*
s5378	35	49	179	2779	21354	884	7219 (66.2%)	8.60	851 (3.7%)	826 (6.6%)	0.52
s9234	19	22	228	5597	44507	994	10710 (75.9%)	42.50	976 (1.8%)	972 (2.2%)	0.96
s13207	31	121	669	7951	62792	2980	14411 (77.0%)	42.30	2813 (5.6%)	2814 (5.6%)	1.02
s15850	14	87	597	9772	78009	2590	17963 (77.0%)	120.80	2497 (3.6%)	2498 (3.6%)	2.13
s35932	35	320	1728	16065	150132	7622	41003 (72.7%)	109.90	7622 (0.0%)	7046 (7.6%)	0.41
s38417	28	106	1636	22179	179320	6812	54119 (69.8%)	142.50	6417 (5.8%)	6415 (5.8%)	5.19
s38584	12	278	1452	19253	165571	6388	63507 (61.6%)	82.70	6358 (0.5%)	6345 (0.7%)	1.04
Avg.							**0.285 (71.5%)**		**0.970 (3.0%)**	**0.954 (4.6%)**	

TABLE VI
EXPERIMENTAL RESULTS FOR OPENCORES BENCHMARKS [18]

	Circuit information					*Our algorithm for general faults*	
Circuit name	*# PIs*	*# POs*	*# FFs*	*# cells*	*# general faults*	*# resulting general faults (reduction ratio)*	*Time (sec)*
pci_bridge32	304	378	1354	12509	174929	71497 (59.13%)	229.20
aes_core	387	258	402	12429	175934	65081 (63.01%)	232.57
wb_conmax	226	218	1775	24212	339971	137992 (59.41%)	734.27
ethernet	192	239	1272	15074	211223	83638 (60.40%)	291.63
des_perf	121	64	1976	35789	491353	190557 (61.22%)	588.09
vga_lcd	223	275	1108	11371	159358	65627 (58.82%)	149.02
Avg.					**1**	**0.397 (60.3%)**	

equivalent fault pairs on average than the methods of Teams A and B, respectively. Also, our algorithm requires substantially less memory and runtime. Consequently, our algorithm outperforms the methods of Teams A and B in all aspects, and hence received the 1st place award in the contest.

In order to show the scalability of our algorithm, we further tested our algorithm on the ISCAS'89 benchmarks [19] with more than two thousands combinational cells, as shown in Table V. The netlist of each benchmark was downloaded from [20]. Only those ISCAS'89 benchmarks that are larger than ICCAD'16 CAD Contest testcases were considered. Table V lists the benchmark statistics, where the number of standard cells range from 2K to 22K. Since the fault lists of the ISCAS'89 benchmarks are not available, we generated two different fault lists for each benchmark, which can be obtained by the command "faultclasses -g" in ABC [2], as described in Section III-A. The first fault list contains all general faults, while the second one contains only stuck-at faults. Since there is no previous work checking fault equivalence for general faults, we simply report our results in Table V. Our method can effectively identify equivalent faults with few minutes, resulting in 71.5% fault reduction ratio on average. We further compare our method with a state-of-the-art EDA tool, Synopsys TetraMAX ATPG, for SA faults only. Our method can result in 1.6% more reduction ratio after identifying equivalent SA faults.

In addition to testing on large ISCAS'89 benchmarks, we expected to perform our algorithm based on the OpenCores benchmarks, which contain more than eleven thousands combinational cells, as shown in Table VI. The netlist of each benchmark is available at [18]. However, the netlists from [18] include blocks with unknown logic specification. We used Berkeley logic interchange format (BLIF) with the capability to represent unknown logic specification [21]. The BLIF of each benchmark is available at [22]. In Table VI, we simply show our result with general faults in this experiment. Our algorithm can achieve 60.3% fault reduction ratios on average for the largest OpenCores benchmarks within only few minutes. The reported runtime in Table VI indicates that our algorithm scales well for a large netlist with 35K combinational cells and 490K faults.

V. CONCLUSION

This paper introduces a new problem formulation to check equivalence of injected faults for the acceleration of qualification process of functional safety verification environment. Based on the presented problem formulation, a new heuristic algorithm to compute equivalence classes of arbitrary (not only stuck-at) faults injected in a given fault list specified for the mapped netlist. The resulting equivalences can reduce the number of injected faults and validation time for functional correctness of large-scale electronic systems. The proposed algorithm is based on a synergistic combination of structural analysis, Boolean satisfiability, and guided simulation. It outperforms the winners of ICCAD'16 CAD Contest and is

applicable to arbitrary mapped netlists. It also achieves excellent fault reduction ratios while substantially reduces search time. The algorithm scales very well for larger ISCAS'89 benchmarks and OpenCores with 35K gates and 490K faults. When comparing with a state-of-the-art commercial EDA tool, it can identify even more equivalent SA faults resulting in higher fault reduction ratio. Our heuristic algorithm achieves excellent fault reduction ratios, which is only 5% worse than the exact brute-force method, while substantially reduces search time from many hours to a few seconds. Our method describes an efficient way of detecting equivalent faults in a mapped netlist, and improves the performance of a qualifying process. The presented fault models include general faults resulting in arbitrary functional changes, in addition to conventional stuck-at faults.

ACKNOWLEDGMENT

The authors would like to thank T. Wei and L. Lin of Synopsys Taiwan Company, Ltd., Hsinchu, Taiwan, for insightful discussions and experimental comparison. The authors would also like to thank the Associate Editor and all anonymous reviewers for the valuable comments which help improve the quality of this paper.

REFERENCES

[1] T. Wei and L. Lin. *Identical Fault Search, 2016 CAD Contest: Problem A*. Accessed: Mar. 28, 2016. [Online]. Available: http://cad-contest-2016.el.cycu.edu.tw/CAD-contest-at-ICCAD2016/index.html

[2] *Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification*. Accessed: Nov. 23, 2016. [Online]. Available: http://wwwcad.eecs.berkeley.edu/∼alanmi/abc

[3] J.-M. Forey, "Automotive safety and security in a verification continuum context," in *Proc. Verification Futures Europe*, 2017. [Online]. Available: http://www.testandverification.com/wpcontent/uploads/2017//Verification_Futures/Jean_Marc_Forey_Synopsys.pdf

[4] *Road Vehicles—Functional Safety*, ISO Standard 26262-1, 2011.

[5] A. Sherer, J. Rose, and R. Oddone, "Ensuring functional safety compliance for ISO 26262," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, Jun. 2015, pp. 1–3.

[6] T. Wei and L. Lin, "ICCAD-2016 CAD contest in large-scale identical fault search," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Austin, TX, USA, Nov. 2016, pp. 1–4.

[7] J. C.-M. Li and M. S. Hsiao, "Fault simulation and test generation," in *Electronic Design Automation: Synthesis, Verification, and Testing*. Amsterdam, The Netherlands: Elsevier, 2009.

[8] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 12, pp. 1377–1394, Dec. 2002.

[9] A. Mishchenko and R. K. Brayton, "SAT-based complete don't-care computation for network optimization," in *Proc. ACM/IEEE Design Autom. Test Europe*, Munich, Germany, 2005, pp. 412–417.

[10] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, San Jose, CA, USA, 2006, pp. 836–843.

[11] J. S. Zhang, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske, "Symmetry detection for large Boolean functions using circuit representation, simulation, and satisfiability," in *Proc. ACM/IEEE Design Autom. Conf.*, San Francisco, CA, USA, 2006, pp. 510–515.

[12] A. Mishchenko *et al.*, "Using simulation and satisfiability to compute flexibilities in Boolean networks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 5, pp. 743–755, May 2006.

[13] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[14] A. J. Hu, "Formal hardware verification with BDDs: An introduction," in *Proc. IEEE Pac. Rim Conf. Commun. Comput. Signal Process. PACRIM. 10 Years Netw. Pac. Rim*, vol. 2. Victoria, BC, Canada, Aug. 1997, pp. 677–682.

[15] S. Jha, Y. Lu, M. Minea, and E. M. Clarke, "Equivalence checking using abstract BDDs," in *Proc. Int. Conf. Comput. Design VLSI Comput. Processors*, Austin, TX, USA, Oct. 1997, pp. 332–337.

[16] R. E. Bryant, "On the complexity of vlsi implementations and graph representations of Boolean functions with application to integer multiplication," *IEEE Trans. Comput.*, vol. 40, no. 2, pp. 205–213, Feb. 1991.

[17] N. Eén and N. Sörensson, *An Extensible SAT-Solver*. Heidelberg, Germany: Springer, 2004, pp. 502–518, doi: 10.1007/978-3-540-24605-3_37. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-24605-3_37#citeas

[18] *OpenCores Benchmarks*. Accessed: Oct. 1, 2017. [Online]. Available: http://opencores.org/

[19] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Proc. IEEE Int. Symp. Circuits Syst.*, vol. 3. Portland, OR, USA, May 1989, pp. 1929–1934.

[20] *ISCAS89 Sequential Benchmark Circuits*. [Online]. Available: https://filebox.ece.vt.edu/ mhsiao/iscas89.html

[21] J. Pistorius and M. Hutton, "Benchmarking method and designs targeting logic synthesis for FPGAs," in *Proc. ACM/IEEE Int. Workshop Logic Synth.*, 2007, pp. 230–237.

[22] *OpenCore Benchmarks in BLIF Format*. Accessed: Oct. 1, 2017. [Online]. Available: https://people.eecs.berkeley.edu/∼alanmi/benchmarks/altera/old/

Ai Quoc Dao received the B.S. and M.S. degrees from the Department of Electrical Engineering, HCMC University of Technology and Education, Ho Chi Minh City, Vietnam, in 2012 and 2015, respectively. She is currently pursuing the Ph.D. degree with the Department of Electrical Engineering, National Chung Cheng University, Chiayi, Taiwan.

Her current research interest includes various electronic design automation problems.

Mark Po-Hung Lin (S'07–M'09–SM'13) received the B.S. and M.S. degrees in electronics engineering from National Chiao Tung University, Hsinchu, Taiwan, and the Ph.D. degree with the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan.

He has been with the Department of Electrical Engineering, National Chung Cheng University, Chiayi, Taiwan, since 2009, where he is currently a Full Professor. He was with SpringSoft, Inc., Hsinchu (acquired by Synopsys, Inc.), from 2000 to 2007, as a Technical Manager. He was a Visiting Scholar with the University of Illinois at Urbana-Champaign, Champaign, IL, USA, from 2007 to 2009, and a Humboldt Research Fellow with the Technical University of Munich, Munich, Germany, from 2013 to 2015. His current research interests include design automation for analog/mixed-signal integrated circuits and low-power circuit and system design optimization.

Dr. Lin was a recipient of the IEEE Best Gold Member Award, the Humboldt Research Fellowship for Experienced Researchers, the JSPS Invitation Fellowship for Research in Japan, the Distinguished Young Scholar Award of Taiwan IC Design Society, and the Distinguished Young Faculty Award of National Chung Cheng University. He has been serving in the technical program committees of several ACM/IEEE conferences, including the Asia and South Pacific Design Automation Conference, the Design Automation Conference, the Design, Automation and Test in Europe, and the International Symposium on Physical Design.

Alan Mishchenko received the M.S. degree from the Moscow Institute of Physics and Technology, Moscow, Russia, in 1993, and the Ph.D. degree from the Glushkov Institute of Cybernetics, Kiev, Ukraine, in 1997.

From 1998 to 2002, he was an Intel-Sponsored Visiting Scientist with Portland State University, Portland, OR, USA. Since 2002, he has been a Research Engineer with the EECS Department, University of California at Berkeley, Berkeley, CA, USA. His current research interest includes developing computationally efficient methods for synthesis and verification.

Dr. Mishchenko was a recipient of the D.O. Pederson TCAD Best Paper Award in 2008 and the SRC Technical Excellence Award in 2011, for his work on ABC.