# Exact Functional Fault Collapsing in Combinational Logic Circuits

Robert Chang, Sep Seyedi, Andreas Veneris
University of Toronto
Dept ECE
Toronto, ON M5S 3G4
{rchang, sep, veneris} @eecg.toronto.edu

Magdy S. Abadir
Motorola
7700 W. Parmer
Austin, TX 78729
m.abadir@motorola.com

## Abstract

Fault equivalence is an essential concept in digital VLSI design with significance in many different areas such as diagnosis, diagnostic ATPG, testability analysis and synthesis. In this paper, an efficient procedure to compute exact fault equivalence classes of combinational circuits is described. The procedure consists of two steps. The first step performs structural fault collapsing and uses fault simulation to return an approximation of the fault equivalent classes. The second step refines these classes with ATPG. Experiments on ISCAS'85 and full-scan ISCAS'89 circuits demonstrate its efficiency.

## 1 Introduction

Computing the complete set of fault equivalence classes is a classical problem in the field of digital VLSI circuit design [1] [2]. Two stuck-at faults are *functionally equivalent* (or *indistinguishable*) if no input test vector can distinguish between them at the primary outputs of the circuit. Therefore, functional fault equivalence is a relation that allows all stuck-at faults in a circuit to be collapsed into sets of disjoint *fault equivalence classes*. In this work, the term fault indicates a stuck-at fault, unless otherwise stated.

There are many reasons why fault equivalence classes are important knowledge to a VLSI designer. An important use is in diagnostic test generation. Since all faults in the same equivalence class can be detected by the same vector, a priori knowledge of equivalence of two faults may save considerable computation in the relatively expensive step of diagnostic test generation [3] [6]. Fault diagnosis also benefits from this knowledge. The effectiveness of diagnosis depends on its resolution, that is, its ability to identify a small set of lines that contain the fault(s) [19]. The shorter the fault list, the better for the test engineer who will probe the circuit to search for the source of failure. Fault equivalence may jeopardize resolution and misguide this search. Hence, prior knowledge of fault equivalence may help improve the diagnostic results by either building accurate fault dictionaries [10] or by reducing the candidates returned by effect-cause algorithms [19].

Another use of fault equivalence is in the field of testability analysis. By definition, each vector that detects a fault in some equivalence class is guaranteed to detect all faults in the same class since the *detectability* of the faults (that is, the set of test vectors that detect them) is the same. In turn, this information can be used by testability enhancement measures such as observation point and control point calculations [1] [2] [15] [17]. Finally, fault equivalence is important in logic optimization. Some existing tools optimize a design through an iterative sequence of appropriate logic rewiring transformations that delete/add some logic to reduce power, increase performance etc [4] [5] [18]. These design rewiring methods usually model logic transformations using stuck-at faults. It has been shown [18], logic addition/deletion is possible if the underlying faults that model these transformations are equivalent. Therefore, knowledge of fault equivalence classes may help develop efficient design rewiring algorithms.

Methods to compute fault equivalence are classified as *structural* and *functional* methods [1] [2]. Structural methods operate on the circuit graph to identify a subset of the complete classes. These methods are fast but they have pessimistic results since they operate in fan-out free circuit regions only. Functional fault equivalence methods are more expensive but they identify more classes [3] [11]. These methods use logic implications to prove equivalence. Since they do not use all implications, they may not return the complete set of classes.

In this work we describe an efficient functional equivalence method that computes the complete set of equivalence classes. The approach is easy to implement, general to apply to other fault types since it models a stuck-at fault and operates as follows. Initially, structural fault collapsing reduces the size of the fault list. Parallel Vector Simulation (PVS) follows for the faults that remain. This procedure returns a set of classes that contain faults that are equivalent only for the test vectors used by PVS. Next, for each class returned by PVS, the algorithm examines each fault pair for equivalence using a novel ATPG-based formal fault equivalence process. This step is motivated by previous efforts of the authors in design rewiring [18]. Experiments presented here demonstrate the efficiency of the approach and suggest different computational trade-offs. They also motivate for future work in the area.

Section 2 presents the two steps of the proposed functional

fault collapsing algorithm. Section 3 present experiments and Section 4 concludes this work.

## 2 Equivalent Fault Identification

In our implementation, identifying equivalent faults is performed in **two steps**. In the first step, an approximation of fault equivalence classes is computed with the use of structural fault collapsing and simulation. The classes are further refined through a formal procedure that employs ATPG. In this Section, we describe each step in detail.

The initial action of the **first step** is to perform structural fault collapsing to quickly reduce the set of faults that need to be considered. Currently, our implementation works along the lines of the structural fault collapsing procedure described in [1]. In the future, we intend to use more advanced structural fault collapsing techniques such as the one described in [12].

The faults that remain at the end of fault collapsing are also the ones that are examined for fault equivalence by *Parallel Vector Simulation (PVS)*. PVS is a simulation-based procedure that classifies these faults into equivalence classes that respect a given input test vector set $T$. In other words, PVS identifies two faults $f_A$ and $f_B$ as equivalent if and only if $f_A$ and $f_B$ give the exact same primary output responses for all test vectors in $T$.

Pseudocode for PVS is given in Fig. 1. The input to PVS is a circuit $C$, the collapsed set of faults $F$ and a set of input test vectors $T$. In experiments, $T$ is a relatively small set of 100-1000 test vectors. This set of vectors consists of random vectors and vectors for stuck-at faults from [7] since a fault detection set is usually available at early stages of the design cycle [14]. As we explain in the next few paragraphs, $T$ is important for the quality of results returned by PVS. We discuss such trade-offs and performance characteristics in Section 3. The output of PVS is a set of fault classes $F_1 \dots F_n$ such that two faults $f_a$ and $f_b$ are in the same class $F_i$ if and only if they have the exact same responses for all vectors in $T$.

The first step of PVS is to simulate (in parallel) all test vectors in $T$ and create an indexed bit-list on every line $l$ in the circuit as in [19] (Fig. 1, line (1)). The $i$-th entry of this bit-list for $l$ contains the logic value of $l$ when the $i$-th input test vector is simulated. Since the test set contains only well defined logic values (`0` and `1`), these bit-lists are conveniently stored as arrays of single 32-bit `unsigned long int` values.

Next, for every stuck-at $v$ fault $f \in F$ on line $l$, value $v$ is injected on $l$ and simulated at the fan-out cone of $l$. The new primary output bit-lists are treated as integers added to produce the *signature* of fault $f$ for test set $T$ (lines 2-6). Once the algorithm computes the signature of $f$, bit-list values are restored at the fan-out cone of $l$ in line 7. This process is repeated for every fault in $F$. Finally, faults that have the exact same signatures are grouped together in line 8 and PVS terminates.

PVS receives a set of faults $F$ and computes fault equivalence classes $F_1, F_2, \dots F_n$ that respect only test vector set

```
Parallel_Vector_Simulation(C, F, T)

(1) Simulate test vectors in T and create
    indexed bit-lists at every circuit line
(2) For every fault f s-a-v on line l do
(3)     fault_signature=0
(4)     set bit-list of l to value v
(5)     simulate at fan-out cone of l
(6)     update fault_signature
(7)     restore bit-lists at fan-out cone of l
(8) Group faults with same signatures together
    into classes F_1 ... F_n
```

Figure 1: Parallel vector Simulation (PVS)

$T$. By construction, these classes are an *overestimation* of the final classes, *i.e.,* there may exist faults in the same class $F_i$ that are not functional equivalent but it will never be the case that two equivalent faults are placed in different classes. This is because PVS bases its results on a small subset of the complete input test vector space and two faults that are not equivalent may have the exact same responses for the test vector set $T$ simply because this set does not contain any distinguishing vectors for these faults. However, faults placed in different classes $F_i$ and $F_j$ are guaranteed not to be equivalent since a distinguished vector already exists in $T$.

PVS is a very fast procedure since it uses a single simulation step at the fan out cone of the fault under consideration. Given two faults $f_A$ and $f_B$ both in $F_i$, **step two** uses ATPG to prove formally whether these faults are equivalent or not. The motivation behind this step originates from our previous work in design rewiring for logic optimization [18] where logic transformations are modeled in terms of stuck-at faults. We present the underlying intuition and the proposed construction with an example. We refer the reader to [18] for further details.

*Example 1:* Consider the circuit in Fig. 2(a), taken from [12], and faults $f_A = I_2 \rightarrow G_1$ s-a-1 and $f_B = G_2 \rightarrow G_4$ s-a-1. Suppose these two faults are placed in the same class $F_i$ by PVS, hence there is high confidence they are equivalent. To formally prove their equivalence, the second step of the algorithm places two multiplexers, shown as boxes in Fig. 2(b), with a *common* select line $S$. The 0-input to the first multiplexer is line $G_2$ and the 1-input of that multiplexer is a constant value of `1` that represent the presense of a s-a-1 fault. Similarly, the 0-input of the second multiplexer is a constant `1` while a branch from $I_2$ feeds the other input. In both cases, the output of each multiplexer connects to the original output of the circuitry.

The reader can verify that when $S = 0$, we operate on a circuit equivalent to the one in Fig. 2(a) under the presence of $f_A$ and when $S = 1$ we operate on a circuit equivalent to the one in Fig. 2(a) under the presense of $f_B$. Therefore, if ATPG for select line $S$ s-a-0 (or, equivalently, s-a-1) returns that the fault is redundant, the two circuits are functional equivalent under the presense of each fault independently which, in turn,

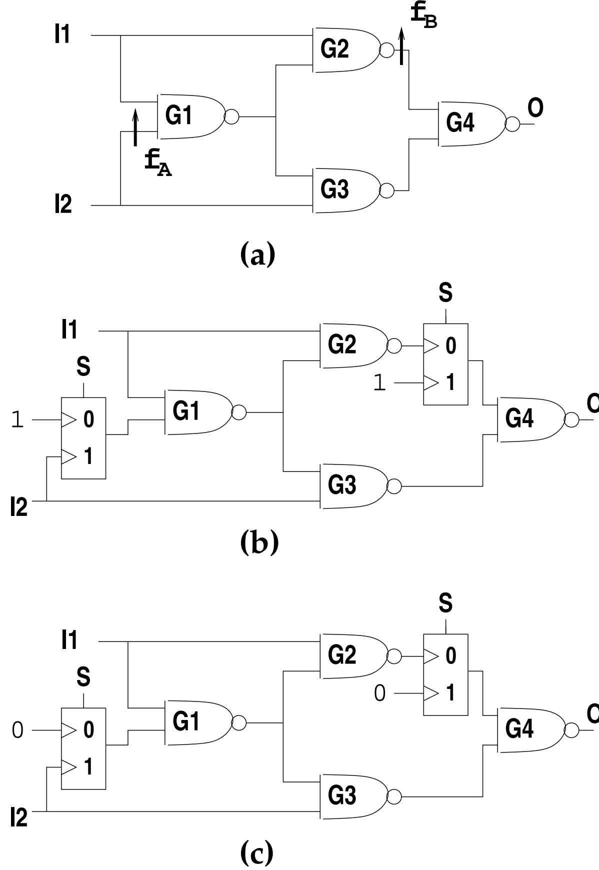


Figure 2: Circuit for Example 1

implies that $(f_A, f_B)$ is an equivalent fault pair [18]. If the fault on $S$ is not redundant, then stuck at faults $f_A$ and $f_B$ alter the functionality of the circuit in Fig. 2(a) differently and they are not equivalent. In this particular example, the stuck-at fault on $S$ is redundant and $(f_A, f_B)$ are indeed equivalent.

*Example 2:* Consider again circuit in Fig. 2(a) and faults on lines $f'_A = I_2 \rightarrow G_1$ and $f'_B = G_2 \rightarrow G_4$ this time both stuck in logic 0. To check whether these new faults are equivalent, we need perform a similar multiplexer construction as in Fig. 2(b) with the difference that a logic 0 is placed on the appropriate multiplexer input to indicate a stuck-at-0 fault. This construction is shown in Fig. 2(c) and ATPG on common select line $S$ s-a-0 returns test vector $(I_1, I_2) = (0, 0)$. This proves that the fault on $S$ is not redundant and faults $f'_A$ and $f'_B$ are not equivalent. This is true since the test vector returned is a vector that detects fault $f'_B$ but does not even excite fault $f'_A$.

Fig. 3 contains pseudocode for the second step. For each class $F_i$ ($i = 1 \ldots n$), a representative $f$ is randomly selected. For each other member $f' \in F_i$, we perform the construction from Example 1 to check whether $f$ and $f'$ are equivalent or not (lines 4-5). If they are not equivalent, $f'$ (and all other such non-equivalent faults from $F_i$) is placed in new class $F_{n+1}$ (lines 6-8) which will be examined later. Observe, that any such fault is guaranteed not to be equivalent with any class $F_1 \ldots F_{i-1} F_{i+1} \ldots F_n$ by PVS. Additionally, faults placed in $F_{n+1}$ may not be equivalent which implies that the algorithm will decompose this class in new classes when it examines it.

```
Formal_Fault_Equivalence(C, F_1, ..., F_n)

( 1) flag=0
( 2) for i=1 to n do
( 3)     randomly select f from F_i
( 4)     for every f' in Fi do
( 5)         perform the MUX construction
( 6)         if f' not equivalent to f do
( 7)             flag=1
( 8)             place f' in F_n+1
( 9)      if flag=1
(10)         flag=0
(11)         n=n+1
```

Figure 3: Proving Fault Equivalence with ATPG

The set of classes returned upon termination of the algorithm are also the exact fault equivalence classes for circuit $C$ and fault set $F$.

# 3 Experiments

We implemented the algorithm outlined in Section 2 in C and ran it on an Ultra 5 SUN workstation with 128 Mb of memory. We tested the approach on ISCAS'85 combinational and full-scan ISCAS'89 sequential benchmark circuits optimized for area using `script.rugged` in SIS [16]. The details of the ATPG engine we employed can be found in [8].

In some experiments we use a complete test vector set for stuck-at faults (ATOM vectors) derived by [7]. This set usually contains less than 200 test vectors and it has a very high fault coverage. We report the average number of the experiments in the next few pages. All run-times are in seconds.

Table 1 contains information on the performance of the algorithm and statistics on the different fault classes. The first column of the table shows the circuit name and the second column contains the total number of stuck-at faults for each circuit. The total number of stuck-at faults is roughly twice the number of lines, including branches, as we avoid injecting faults at primary inputs and outputs. The third column of the table shows the faults after structural fault collapsing. This is also the number of faults input to the two-step algorithm proposed in Section 2. To perform structural fault collapsing, we use the simple method described in [1]. In the future, we intend to use additional structural fault collapsing engines such as the ones described in [3][12] to further reduce the number of faults to be considered. This is expected to improve performance of the formal fault equivalence proving algorithm.

Columns 4–7 of Table 1 contain information that pertain to PVS. In more detail, they show the number of distinct fault classes upon termination of PVS.

Table 1: Results and Statistics

| ckt name | # of initial faults | faults after collaps. | # of fault classes after PVS ATOM vectors | ATOM and 500 random | 500 random | 1000 random | # fault pairs checked | # final fault classes | % of error | PVS time (sec) | ATPG time (sec) | Total time (sec) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c432 | 810 | 431 | 383 | 429 | 425 | 430 | 1 | 431 | 0.2 | 2.4 | 0.01 | 2.4 |
| c499 | 2434 | 1314 | 901 | 1076 | 1027 | 1092 | 231 | 1154 | 5.4 | 21.5 | 12.8 | 34.4 |
| c880 | 1778 | 948 | 863 | 895 | 859 | 874 | 154 | 909 | 3.8 | 4.6 | 3.2 | 7.8 |
| c1355 | 2412 | 1302 | 1046 | 1088 | 1010 | 1079 | 233 | 1142 | 5.5 | 17.6 | 16.6 | 34.2 |
| c1908 | 1802 | 975 | 714 | 748 | 684 | 767 | 561 | 859 | 10.7 | 10.6 | 37.1 | 47.7 |
| c2670 | 3264 | 1753 | 1193 | 1232 | 1212 | 1237 | 24812 | 1521 | 18.7 | 10.1 | 956.8 | 967.0 |
| c3540 | 4520 | 2398 | 1425 | 1566 | 1559 | 1598 | 17828 | 1861 | 14.1 | 175.0 | 400.7 | 575.7 |
| c5315 | 7148 | 3849 | 3019 | 3454 | 3430 | 3465 | 742 | 3615 | 4.1 | 45.6 | 58.1 | 103.7 |
| c6288 | 14314 | 7489 | 6403 | 6603 | 6599 | 6603 | 913 | 6981 | 5.4 | 58.7 | 51.0 | 109.8 |
| c7552 | 10308 | 5541 | 4292 | 4386 | 4285 | 4378 | 44055 | 4988 | 12.2 | 71.8 | 6796.9 | 6868.7 |
| s820 | 1484 | 783 | 565 | 673 | 640 | 682 | 783 | 767 | 11.1 | 1.8 | 6.3 | 8.1 |
| s1196 | 2428 | 1269 | 853 | 1008 | 976 | 1044 | 6180 | 1232 | 15.2 | 4.5 | 84.7 | 89.3 |
| s1238 | 2282 | 1210 | 756 | 947 | 914 | 972 | 10362 | 1185 | 17.9 | 6.3 | 225.2 | 231.5 |
| s1494 | 2792 | 1464 | 1141 | 1319 | 1291 | 1353 | 489 | 1459 | 7.2 | 4.6 | 5.3 | 9.9 |
| s5378 | 6258 | 3452 | 2338 | 2625 | 2289 | 2655 | 21592 | 3186 | 16.6 | 25.3 | 391.3 | 416.7 |
| s35932 | 49160 | 27746 | 24223 | 24429 | 24428 | 24429 | 2742 | 24813 | 1.5 | 1564.7 | 389.8 | 1954.5 |

We compute this quantity for four different cases with respect to the test vector set used by PVS: *(i)* ATOM vectors, *(ii)* ATOM vectors and 500 random vectors, *(iii)* 500 random vectors, and *(iv)* 1000 random vectors. Intuitively, the more vectors we simulate the more accurate results we expect in terms of number of classes, as discussed earlier in the paper.

A close examination of these numbers indicates that a relatively small set of random vectors (case *(iv)*) gives sufficient resolution. We observe, in most cases, random simulation alone outperforms case *(ii)* and there is little to gain by using a pre-computed set of test vectors. Therefore, we use the classes from case *(iv)* as input to the second step. This result is also confirmed if we examine the exact set of fault classes computed by ATPG, as discussed in the paragraphs that follow.

Columns 8 and 9 show the number of fault pairs checked by ATPG and the final number of fault classes after formal fault equivalence, respectively. The relative error between PVS (Step 1), a simulation-based process, and ATPG (Step 2), a formal proving engine, is found in column 10. It is seen that in many cases the relative error is rather small (less than 10%) and it is expected to improve if more advanced structural fault collapsing methods are employed. To appreciate this result, one need consider the CPU time dedicated to each task alone (Steps 1 and 2), shown in columns 11 and 12. As expected, ATPG time dominates that of PVS which suggests that one may afford a small relative error in computing fault classes as a trade-off for run-time efficiency. The last column of Table 1 contains the total time for both steps.

The time for ATPG in Step 2 can be large because the algorithm checks exhaustively every pair of faults in each class. It follows that the less faults per class for a circuit after PVS, the less time ATPG is expected to consume. This circuit-dependent property is depicted for four benchmarks in Fig. 4. In that figure, there are two graphs for each circuit. The graph on the left depicts the percentage of fault pairs versus the time allocated by ATPG to formally check their equivalence (Step 2). In this graph, fault pairs are classified in five categories (A . . . E) according to the time ATPG needs to prove their equivalence via the multiplexer construction. The graph on the right contains statistics on the number of groups returned at the end of the method versus the size of these groups (not including groups of size 1). We observe that for most circuits fault equivalence favors small groups of size two. We also observe that in most cases, ATPG is very efficient in proving fault equivalence as the majority of faults need times that are less than 0.05 seconds (categories A, B and C).

# 4 Conclusions

Fault equivalence is an important problem in digital VLSI circuits. In this paper, we presented a formal fault equivalence method which uses simulation-based and ATPG-based techniques to return the exact and complete fault equivalence classes. Experiments demonstrate its robustness and effectiveness. In the future, we intend to refine this method with more advanced structural equivalent fault collapsing methods and improve performance. We also plan to apply ideas presented here to diagnostic ATPG.

# References


[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, “Digital Systems Testing and Design for Testability,” *IEEE Press,* 1990.

[2] M. L. Bushnell and V. D. Agrawal, “Essentials of Electronic Testing,” *Kluwer Academic Publishers,* 2001.

[3] M. E.Amyeen, W. K. Fucks, I. Pomeranz and V. Boppana, "Fault Equivalence identification using redundancy information and static and dynamic extraction," in *Proc. of IEEE VLSI Test Symposium, pp. 124-130,* 2001.

[4] S. C. Chang and M. Marek-Sadowska, "Perturb and Simplify: Multi-Level Boolean Network Optimizer," in *Proc. Int'l Conference on Computer-Aided Design, pp. 2-5,* 1994.

[5] J. A. Espejo, L. Entrena, E. San Millàn, and E. Olìas, "Functional extension of structural logic optimization techniques," in *Proc. of Asian-South-Pacific Design Automation Conference, pp. 467-472,* 2001.

[6] T. Gruning, U. Mahlstedt, and H. Koopmeiners, "DIATEST: A fast diagnostic test pattern generator for combinational circuits," in *Proc. Int'l Conf. on Computer-Aided Design, pp. 194-197,* 1991.

[7] I. Hamzaoglu and J. H. Patel, "New Techniques for Deterministic Test Pattern Generation," in *Proc. of VLSI Test Symposium, pp. 446-452,* 1998.

[8] W. Kunz and D. K. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems–Test, Verification, and Optimization," in *IEEE Trans. on Computer-Aided Design, vol. 13, no. 9, pp. 1143-1158* September 1994.

[9] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," in *IEEE Trans. on Computers, vol. C-32, no. 12,* December 1983.

[10] C. R. Kime, "Fault-Tolerant Computing: Theory and Techniques," *vol. II, pp. 577-632, Prentice Hall,* 1986.

[11] A. Lioy, "Advanced Fault Collapsing," in *IEEE Design & Test of Computers, vol. 9, no. 1, pp. 64-71,* March 1999.

[12] M. Nadjarbashi, Z. Navabi and M. R. Movahedin, "Line Oriented Structural Equivalence Fault Collapsing," in *IEEE Workshop on Model and Test,* 2000.

[13] A. V. S. S. Prasad, V. D. Agrawal, and M. V. Atre, "A New Algorithm for Global Fault Collapsing into Equivalence and Dominance Sets," in *Proc. IEEE Int'l Test Conf., pp. 391-397,* 2002.

[14] S. M. Reddy, "Complete Test Sets for Logic Functions," in *IEEE Trans. on Computers, vol. C-22, no. 11, pp. 1016-1020,* November 1973.

[15] Y. Savaria, M. Youussef, B. Kaminska, and M. Koudil, "Automatic Test Point Insertion for Pseudo-random Testing," in *Proc. Int'l Symposium on Circuits and Systems, pp. 1960-1963,* 1991.

[16] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential Circuit Design Using Synthesis and Optimization," in *Proc. of Int'l Conference on Computer Design, pp. 328-333,* 1992.

[17] N. A. Touba, and E. J. McCluskey, "RP-SYN: Synthesis of Random-Pattern Testable Circuits with Test Point Insertion," in *IEEE Trans. on Computer-Aided Design, vol. 18, no. 8, pp. 1202-1213,* August 1999.

[18] A. Veneris, M. S. Abadir and M. Amiri, "Design Rewiring Using ATPG," in *Proc. IEEE Int'l Test Conf., pp. 223-232,* 2002.

[19] A. Veneris, J. Liu, M. Amiri and M. S. Abadir, "Incremental Diagnosis and Debugging of Multiple Faults and Errors," in *Proc. of Design and Test in Europe, pp. 716-721,* 2002.

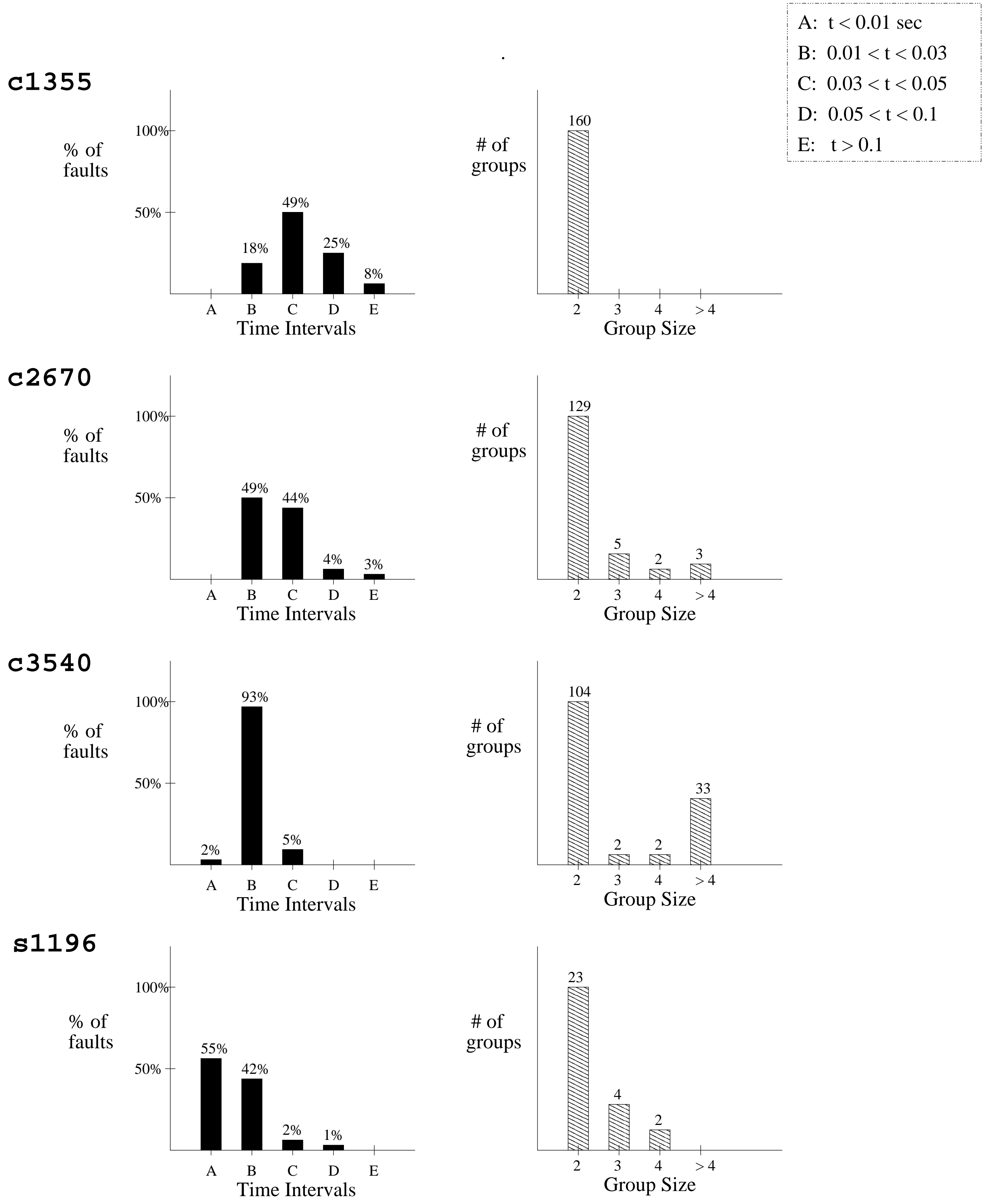


Figure 4: Distribution of Fault Classes and ATPG Time