ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/261336209

Effectiveness of the cumulative vs. normal mode of operation for combinatorial testing

Conference Paper · October 2010

DOI: 10.1109/ISIEA.2010.5679441

CITATIONS
5

READS
64

3 authors:

Mohammed I. Younis
University of Baghdad
48 PUBLICATIONS **351** CITATIONS

SEE PROFILE

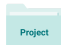
Kamal Z Zamli
Universiti Malaysia Pahang
167 PUBLICATIONS **1,250** CITATIONS

SEE PROFILE

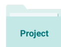
Rozmie Razif Othman
Universiti Malaysia Perlis
32 PUBLICATIONS **111** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Environmental medicine: social and medical aspects View project



Current Projects View project

All content following this page was uploaded by Rozmie Razif Othman on 19 February 2014.

The user has requested enhancement of the downloaded file.

Effectiveness of the Cumulative vs. Normal Mode of Operation for Combinatorial Testing

Mohammed I. Younis, Kamal Z. Zamli, and Rozmie R. Othman
School of Electrical and Electronic Engineering,
Universiti Sains Malaysia,
Penang, Malaysia
younismi@gmail.com, eekamal@eng.usm.my, rozmie.razif.othman@gmail.com

***Abstract*—This paper discusses the state of the art of applying combinatorial interaction testing (CIT) in conjunction with mutation testing for hardware testing. In addition, the paper discusses the art of the practice of applying CIT in normal and cumulative mode in order to derive an optimal test suite that can be used for hardware testing in a production line. Our previous study based on applying CIT in cumulative mode; described the systematic application of the strategy for testing 4-bit Magnitude Comparator Integrated Circuits in a production line. Complementing our previous work, this paper compares the effectiveness of cumulative mode versus normal mode of operation. Our result demonstrates that the use of CIT in cumulative mode is more practical than normal mode of operation as far as detecting faults introduced by mutation.**

***Keywords*— t-way testing; combinatorial interaction testing; multi way testing; mutation testing; hardware testing; software testing**

I. INTRODUCTION

Engineering can be viewed as a discipline that aims to facilitate the production of fault free product within time with acceptable budget. In order to ensure software coverage and reliability, many interactions of possible input parameters, embedded system (i.e. hardware/middleware /software) environments, and system configurations needed to be tested and verified against for conformance. Although desirable, exhaustive testing is next to impossible due to resources as well as timing constraints [1] [2].

As illustration, consider the Customize Toolbars option menu for Adobe Acrobat Professional (see Fig. 1). Even if only Customize Toolbars option is considered, there are already 123 possible configurations to be tested. Next, each configuration can take two values (i.e. checked or unchecked). Here, there are 2^{123} combinations of test cases to evaluate. Assuming that it takes only one second for one test case, then it would require nearly 34×10^{28} years for a complete test of the Customize Toolbars option.

Similar observed situation when testing a typical hardware product. As a simple example, consider a hardware product with 30 tri state (true, false, high impedance) buffers. To test all possible combination would require 3^{30} test cases. If the time requires executing and observing for one test case is one second, then it would take nearly 6.6×10^{6} years for a complete test.

As an example of a small-scale product, consider 4-bits Magnitude Comparator IC. The Magnitude Comparator IC consists of 8-bits for inputs and 3-bits for output. It is clear that each IC requires 256 test cases for exhaustive testing. Assuming that each test case takes one second to run and observed, the testing time for each IC is 256 seconds. Here, if it is required to test one million chips, the testing process will take more than 8 years if we use single line of test [2]. Now, let us assume that we received an order of delivery for one million qualified (i.e., tested) chips within two weeks. Here, we can do parallel testing. However, parallel testing can be expensive due to the need for 212 testing lines [2].

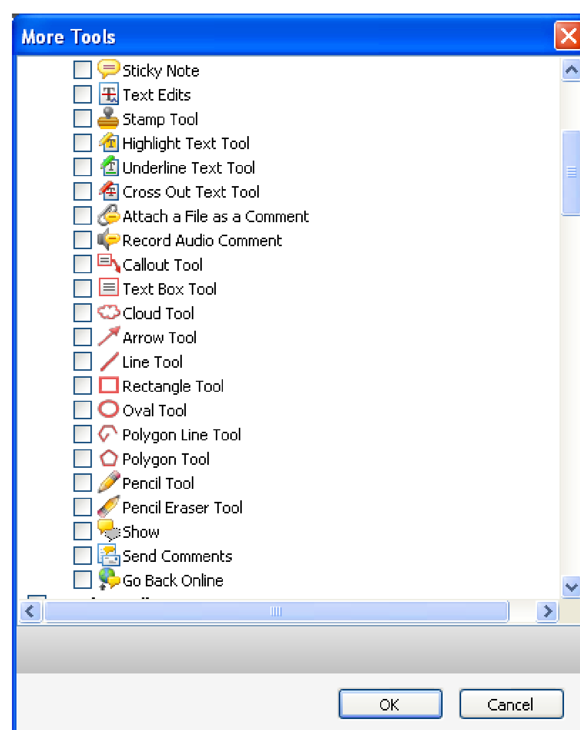


Figure 1. Customize toolbars dialogue for Adobe Acrobat Professional

Obviously, there is a need for a systematic strategy in order to reduce the test data set into manageable ones for both software and hardware testing. The systematic solution to this problem based on *Combinatorial Interaction Testing* (CIT) strategy. The CIT approach can systematically reduce the number of test cases by selecting a subset from exhaustive testing combination based on the strength of interaction coverage. CIT strategies focused on 2-way (pairwise) testing in the last decade. More recently, there are several strategies that can be generated for high degree interaction ($2 \leq t \leq 6$): Jenny [3], TConfig [4], TVG [5] ACTL [6]. Finally,

MC_MIPOG [7] reported as a strategy that supports very high degree of interaction ($1 \leq t \leq 12$).

In this paper, we propose a novel comparison on the application of CIT in normal mode and cumulative mode. Here, the application of interest involves the integration of the CIT with a fault injection tool to produce minimal test suite that is capable to detect all faults injected into the system. In this case, we are also interested to study the effectiveness of this minimization in the production line for IC manufacturer. The rest of this paper is organized as follows. Section 2 presents related work on the state of the art of the applications of t-way testing. Section 3 presents the proposed minimization strategy. Section 4 gives a systematic example as a proving of concept. Finally, section 5 states the conclusion and suggestion for future work.

II. RELATED WORKS

Concerning its usage, interaction testing has a wide range of applications. A significant efforts in the literature put focus on pairwise testing. Mandl adopts pairwise coverage using Orthogonal Latin Square (OLS) to testing an Ada compiler [8]. Berling and Runeson use interaction testing to identify real and false targets in target identification system [9]. White has also applied the technique to test Graphical User Interfaces (GUI) [10]. The use of fault injection techniques for software and hardware testing is not new. Other applications of interaction testing include regression testing through the GUI [11] and fault localization [12]. Tang and Chen, Boroday, and Chandra et al. studied circuit testing in hardware environment, proposing test coverage that includes each 2t of the input settings for each subset of t inputs [13-15]. Seroussi and Bshouti give a comprehensive treatment for circuit testing [16]. In addition, Dumer examines the related question of isolating memory faults, and uses binary covering arrays [17].

Concerning the general interaction testing, much work has also been undertaken in the literature. Dunietz et al. demonstrate the need for higher order strength. In this case, Dunietz et al. demonstrates that significant block coverage is obtained when testing with two-way interactions, but higher strength is needed for good path coverage [18]. In other work, it is found that 100% of faults detectable by a relatively low degree of interaction, typically 4-way combinations [19-22].

The National Institute of Standards and Technology (NIST) investigated the application of interaction testing for 4 application domains: medical devices, a Web browser, a HTTP server, and a NASA distributed database. It was reported that in the NASA study, 95% of the actual faults on the test software involve 4-way interaction [23, 24]. In fact, according to the recommendation from NIST, almost all of the faults detected with 6-way interaction. Younis and Zamli presented a novel approach to use interaction testing for test data generator for reverse engineering of combinational circuit [25]. Unlike the NIST study, Younis and Zamli demonstrated the requirement of higher degree interaction test suite (i.e., t>6). Finally, Ghosh and Kelly give a survey to include a number of studies and tools that have been reported in the area of failure mode identification [26].

III. PROPOSED STRATEGY

The proposed strategy consists for two parts, namely: *Test Quality Signing* (TQS) process and *Test Verification* process (TV).

The TQS process aims to derive an effective and optimum test suite. Based on CIT mode of operation, there are two TQS; namely: TQS_Normal and TQS_Cumulative [2] processes. The TQS_Normal works as follows.

1. Start with an empty *optimized test suite* (OTS).
2. Build an equivalent software class for the C*ircuit Under Test* (CUT).
3. Inject the equivalent class with all possible faults; store these faults in *fault list* (FL).
4. Initialize CIT strategy with the desired strength of coverage (t).
5. Let CIT strategy partitioning the exhaustive test space. The portioning involves generating one test case at a time for t coverage.
6. CIT strategy generates one *test case* (TC).
7. Execute TC.
8. If TC detects any fault in FL, remove the detected fault(s) from FL, and add TC and its specification output(s) to OTS.
9. If (FL is not empty or t tuples are not covered) go to 6.
10. The desired optimized test suite and its corresponding output(s) are stored in OTS.

The TQS_Cumulative [2] works as follows.

1. Start with an empty *optimized test suite* (OTS).
2. Build an equivalent software class for the *circuit under test* (cut).
3. Inject the equivalent class with all possible faults; store these faults in *fault list* (FL).
4. Let N be maximum number of parameters.
5. Initialize CIT strategy with strength of coverage (t) equal one (i.e., t=1).
6. Let CIT strategy partitioning the exhaustive test space. The portioning involves generating one test case at a time for t coverage. If t coverage criteria is satisfied, then t=t+1.
7. CIT strategy generates one *test case* (TC).
8. Execute TC.
9. If TC detects any fault in FL, remove the detected fault(s) from FL, and add TC and its specification output(s) to OTS.
10. If (FL is not empty or t<=N) go to 7.
11. The desired optimized test suite and its corresponding output(s) are stored in OTS.

The TV process involves the verification of fault free for each unit. TV process for a single unit works as follows.

1. For each TC in OTS do:
    a. Subject the unit under test to TC.
    b. Compare the output(s) from the unit to output(s) already stored in OTS.
    c. If the outputs are equal, continue. Else, go to 3.
2. Report that the cut has been passing in the test. Go to 4.

3. Report that the cut has failed the test.
4. The verification process ends.

The reason to undertake equivalent software class for the cut is to ensure that the cost and control of the fault injection be more practical than performing it directly to a real hardware circuit. Furthermore, the derivation of OTS is faster in software than in hardware. In the next section, we will compare the effectiveness of the proposed strategies in a production line.

IV. EVALUATION AND DISCUSSION

As proof of concept, we have adopted MC_MIPOG [7] as our CIT strategy implementation, and MuJava version 3 (described in [27] [28]) as our fault injection strategy implementation.

Briefly, MC_MIPOG is a combinatorial test generator based on specified inputs and parameter interaction. Running on a Grid environment, MC_MIPOG adopts both the horizontal and vertical extension mechanism (i.e. similar to that of IPOG [22]) in order to derive the required test suite for a given interaction strength. While there are many useful combinatorial test generators in the literature, the rationale for choosing MC_MIPOG is the fact that it supports high degree of interaction and can be run in normal and cumulative modes (i.e. support one-test-at-a-time approach with the capability to vary t automatically until the exhaustive testing is reached).

Complementary to MC_MIPOG, MuJava is a fault injection tool that permits mutated Java code (i.e. based on some defined operators) to be injected into the running Java program. Here, the reason for choosing MuJava stemmed from the fact that it is a public domain Java tool freely accessible for download in the internet [28].

Using both tools (i.e. MC_MIPOG and MuJava), a case study problem involving a 4-bit Magnitude Comparator IC will be discussed here in order to evaluate the proposed strategy. A 4-bit Magnitude Comparator consists of 8-inputs (two four bits inputs namely: a0..a3, and b0..b3. where ao, bo are the most significant bits), 4 xnor gates (or equivalent to 4xor with 4 not gates), five not gates, five and gates, three or gates, and three outputs. The actual circuit realization of the Magnitude Comparator is given in Fig. 2. Here, it should be noted that this version of the circuit is a variant realization (implementation) of the Magnitude Comparator found in [29]. The equivalent class of the Magnitude Comparator is given in Fig. 3 (using the Java-programming language) [2].

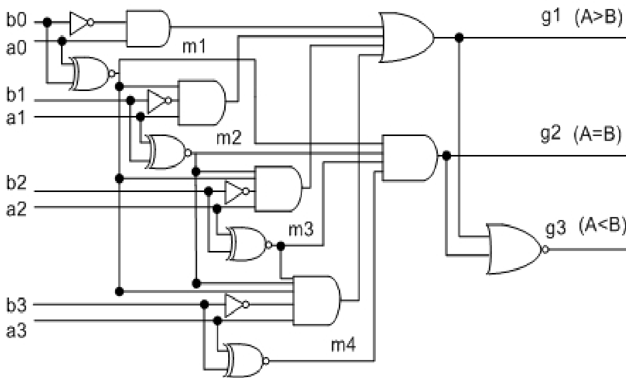


Figure 2. Schematic diagram for the 4-bit Magnitude Comparator

Here, it is important to ensure that the software implementation obeys the hardware implementation strictly. By doing so, we can undertake the fault injection and produce the OTS in the software domain without affecting the logical of relation and parameter interactions of the hardware implementation.

Now, we apply the TQS processes (i.e, TQS_Normal, TQS_Cumulative), as illustrated in section 3. Here, there are 80 faults injected in the system. To assist our work, we use MC_MIPOG [7] [25] to produce the TC in a normal mode followed by a cumulative mode. Following the steps in TQS process, Tables 1, and 2 demonstrate the derivation of OTS_Normal and QTS_Cumulative respectively.

```
public class Comparator
{

 /* Comparator class  takes two four bits numbers (A&B),
    where i.e. A=a0a1a2a3, B=b0b1b2b3. Here, a0 and b0
    are the most significant bits. The function returns an
    output string g1, g2, an g3.
    g1 - represent the output  A>B
    g2 – represents the output A=B,
    g3 – represents the output A<B
   The code symbols (!, ^, |, and  &)represent the logical
   operator for Not,Xor,Or, and And respectively.  */

 public static String compare (
                          boolean a0, boolean a1,
                          boolean a2, boolean a3,
                          boolean b0, boolean b1,
                          boolean b2, boolean b3  )
      {
         boolean    g1,g2,g3 ;
         boolean    m1,m2,m3,m4;
         String s=null;
         m1=!(a0^b0);
         m2=!(a1^b1);
         m3=!(a2^b2);
         m4=!(a3^b3);
         g1=(a0 &!b0)| (m1&a1 &!b1)
              |(m1&m2&a2 &!b2)|
                (m1&m2 &m3&a3 &!b3) ;
         g2= (m1&m2 &m3&m4)   ;
         g3=!(g1|g2);
         s=g1+""+g2+""+g3
         return s;
      }

}
```

Figure 3. Equivalent class Java program for the 4-bit Magnitude Comparator

TABLE I.
DERIVATION OF OTS_NORMAL FOR THE 4-BIT MAGNITUDE COMPARATOR

t=	Test Size	Live Mutant	Killed Mutant	%Mutant Score	Effective test size
1	2	15	65	81.25	2
2	8	8	72	90.00	6
3	18	2	78	97.50	11
4	39	0	80	100.00	10
5	69	2	78	97.50	13
6	116	2	78	97.50	16
7	128	0	80	100.00	13
8	256	0	80	100.00	13

Referring to Table I from a different perspective, it can be observed that the efforts for generation and execution the test suite increases exponentially as far as the strength of coverage (t) is concerned. Moreover, it can also be observed that for this system, the test suites for t=4, 7, and 8 have successfully removed all faults (i.e.mutant score of 100%). It is worth noting that for t=5, and 6, there are some mutants still in existence. The reason behind this is the fact that the test suite for t=4 is unnecessarily a subset for t=5 and 6 respectively. From the other perspective, the efficiency of integration MC_MIPOG in normal mode with MuJava can be observed (by taken only the effective TC) in the last column in Table I. Here, it is clear that the optimal test suite can be determined when t=4. In order to come out with the conclusion (i.e. at t=4), there is a need to run each individual test suite individually (i.e. for each t from 1 to maximum strength interaction coverage). This is impractical as it is hard to predict the strength of coverage that satisfies the fault removing requirement.

Considering the same 4 bit magnitude comparator, Table II highlights the result for cumulative mode of operation.

TABLE II.
DERIVATION OF OTS_CUMULATIVE FOR THE 4-BIT MAGNITUDE COMPARATOR

t=	Cumulative Test Size	Live Mutant	Killed Mutant	%Mutant Score	Effective test size
1	2	15	65	81.25	2
2	9	5	75	93.75	6
3	24	2	78	97.50	8
4	36	0	80	100.00	9

Referring to Table II, it should be noted that the first 36 test cases can removes all the faults. Furthermore, only the first 12 test cases when t=4 are needed to catch that last two live mutants. Here, there is no need to go to t=5 through 8 as compared to earlier result in Table I, resulting into less number of executed test cases. The efficiency of integration MC_MIPOG with MuJava can be observed (by taken only the effective TC) in the last column in Table II. It is clear that the cumulative mode has successfully predicted all faults in the system using only 36 trails which is less than 39 trails when t=4 in Table I. Finally, the overall test suites derived from the cumulative mode is 9 which is also less than 10 in normal mode when t=4. As such, adopting cumulative mode is more systematic than the normal mode of operation with better utilization of efforts. For all above mentioned reasons, we will take the test suite derived from TQS_Cumulative as the desired OTS. Table III gives the desired OTS, outputs, and TV process. Here T and F represent true and false respectively. In this case, TQS process reduces the test size to nine test cases only, which significantly improves the TV process.

To illustrate how the verification process is done (see Figure 2), assume that the second output (i.e., A=B) is out-of-order (i.e. malfunction). Suppose that A=B output is always on (i.e., short circuit to 'VCC'). This fault cannot be detected as either TC1 or TC2 (according to Table2). Nevertheless, when TC3 the output vector ('VV') of faulty IC is FTT, and the SV is FFT, the TV process can straightforwardly detects that the IC is malfunctioning (i.e. cut fails).

TABLE III.
OTS , OUTPUTS AND TV PROCESS FOR THE 4-BIT MAGNITUDE COMPARATOR

#TC	OTS TC (a0...a3, b0..b3)	Outputs(A>B, A=B, A<B)	Accumulative faults detected /80
1	FFFFFFFF	F T F	53
2	TTTTTTTT	F T F	65
3	FTTTTTTT	F F T	68
4	TTFTFTFT	T F F	71
5	TTFFTFTT	T F F	72
6	TTTFTTFF	T F F	75
7	TTFTTTTF	F F T	77
8	FFTTTTTF	F F T	78
9	TFTTTFTF	T F F	80

To consider the effectiveness of the proposed strategy in the production line, we return to our illustrative example given in section 1. Here, the reduction of exhaustive test from 256 test cases to merely nine test cases is significantly important. In this case, the TV process requires only 9 seconds instead of 256 seconds for considering all tests. Now, using one testing line and adopting our strategy for two weeks can test (14X24X60X60/9=134400) chips. Hence, to deliver one millions tested ICs' during these two weeks; our strategy requires eight parallel testing lines instead of 212 testing lines (if the test depends on exhaustive testing strategy). Now, if we consider the saving efforts factor as the size of exhaustive test suite minus optimized test suite to the size of exhaustive test suite, we would obtain the saving efforts factor of 256-9/256=96.48%.

V. CONCLUSION

This paper has discussed the validation of using combination of t-way testing with fault injection strategies for systematic IC testing. In addition, the paper concludes and stresses that putting the t-way in cumulative mode is more practical than normal mode of operation. Our case study in hardware production line demonstrated the proposed strategy could improve the saving efforts factor significantly. As a part of our future work, we plan to integrate the proposed strategy for fault localization in a hardware design tool.

ACKNOWLEDGMENT

We acknowledge the help of Jeff Offutt, Jeff Lei, Raghu Kacker, Rick Kuhn, Myra B. Cohen, and Sudipto

Ghosh for providing us with useful comments and the background materials. This research is partially funded by the USM: Post Graduate Research Grant –T-Way Test Data Generation Strategy Utilizing Multicore System, and the Fundamental research grants – “Development of Interaction Testing Tool for Pairwise Coverage with Seeding and Constraint” from Ministry of Higher Education (MOHE). The first author, Mohammed I. Younis, is the USM Fellowship recipient

REFERENCES

[1] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "YZ Strategy for IC Testing," in t*he 7th International Conference on Robotics, Vision, Signal Processing, & Power Applications (RoViSP 2009)* Langkawi, Kedah, Malyasia 2009, pp. 1-5.

[2] M. I. Younis and K. Z. Zamli, "A Strategy for Automatic Quality Signing and Verification Processes for Hardware and Software Testing," Advances in Software Engineering, pp. 1-7, 2010.

[3] B. Jenkins, "Jenny Test tool", http://www.burtleburtle.net./bob/math/jenny.html.

[4] A. Williams, "TConfig Test Tool. ," School of Information Technology and Eng., University of Ottawa. http://www.site.uottawa.ca/~awilliam/.

[5] J. Arshem, "Test Vector Generator Tool (TVG)", http://sourceforge.net/projects/tvg/.

[6] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter, "Combinatorial Software Testing," *IEEE Trans. on Computer,* vol. 42, pp. 94-96, Aug. 2009.

[7] M. I. Younis and K. Z. Zamli, "MC-MIPOG: A Parallel t-Way Test Generation Strategy for Multicore Systems," *ETRI Journal,* vol. 32, pp. 73-82, February 2010.

[8] R. Mandl, "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing," *Comm. of the ACM,* vol. 28, pp. 1054-1058, 1985.

[9] T. Berling and P. Runeson, "Efficient Evaluation of Multifactor Dependent System Performance Using Fractional Design," *IEEE Trans. on Software Eng.,* vol. 29, pp. 769–781, 2003.

[10] L. White and H. Almezen, "Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences," in *the Intl. Sym. on Software Reliability Eng.*, Piscataway, NJ, 2000, pp. 110–121.

[11] A. M. Memon and M. L. Soffa, "Regression Testing of GUIs," in the 9th European Software Eng. Conf. (ESEC) and 11th ACM SIGSOFT Intl. Sym. on the Foundations of Software Eng. (FSE-11), Helsinki, Finland, 2003, pp. 118–127.

[12] M. S. Reorda, Z. Peng, and M. Violanate, "System-Level Test and Validation of Hardware/Software Systems," in Advanced Microelectronics Series London: Springer-Verlag, 2005.

[13] D. T. Tang and C. L. Chen, "Iterative Exhaustive Pattern Generation for Logic Testing," *IBM Jrnl. Research and Development,* vol. 28, pp. 212-219, 1984.

[14] S. Y. Boroday, "Determining Essential Arguments of Boolean Functions " in *the Conf. on Industrial Mathematics*, Taganrog, 1998, pp. 59-61.

[15] A. K. Chandra, L. T. Kou, G. Markowsky, and S. Zaks, "On Sets of Boolean n-Vectors with All k-Projections Surjective," *Acta Informatica 20,* vol. 20, pp. 103-111, October 1983.

[16] G. Seroussi and N. H. Bshouty, "Vector Sets for Exhaustive Testing of Logic Circuits," *IEEE Trans. on Information Theory,* vol. 34, pp. 513-522, 1988.

[17] Dumer, "Asymptotically Optimal Codes Correcting Memory Defects of Fixed Multiplicity," *Problemy Peredachi Informatskii,* vol. 25, pp. 3–20, 1989.

[18] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, "Applying Design of Experiments to Software Testing," in *the Intl. Conf. on Software Eng. (ICSE '97)*, Boston, MA, 1997, pp. 205–215.

[19] D. R. Wallace and D. R. Kuhn, "Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data," *Intl. Jrnl. of Reliability, Quality, and Safety Eng.,* vol. 8, 2001.

[20] D. R. Kuhn and M. J. Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing," in *the 27th Annual NASA Goddard Software Eng. Workshop (SEW-27'02)*, 2002, pp. 91-95.

[21] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Trans. on Software Eng.,* vol. 30, pp. 418-421, 2004.

[22] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: Efficient Test Generation for Multi-way Combinatorial Testing," *Software Testing, Verification and Reliability,* vol. 18, pp. 125-148, 2008.

[23] D. R. Kuhn and V. Okun, "Pseudo Exhaustive Testing For Software," in *the 30th NASA/IEEE Software Eng. Workshop*, 2006, pp. 25-27.

[24] R. Kuhn, Y. Lei, and R. Kacker, "Practical Combinatorial Testing: Beyond Pairwise," *IEEE IT Professional,* vol. 10, pp. 19-23, June 2008.

[25] M. I. Younis and K. Z. Zamli, "Assessing Combinatorial Interaction Strategy for Reverse Engineering of Combinational Circuits," in the *IEEE Symp. on Industrial Electronics and Applications (ISIEA 2009)*, Kuala Lumpur, Malaysia, 2009.

[26] G. Sudipto and L. K. John, “Bytecode Fault Injection for Java Software”, Journal of Systems and Software, November 2008, pp. 2034-2043.

[27] Y. Ma, J. Offutt, and Y. Kwon, “MuJava: An Automated Class Mutation System”, Journal of Software Testing, Verification and Reliability, 15(2), June 2005, p.p. 97-133.

[28] MuJava Version 3. Available at http://cs.gmu.edu/~offutt/mujava/. Last accessed on March 2010.

[29] M. M. Mano, “Digital Design”, Third Edition, Prentice Hall Inc., 2002.

View publication stats