Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2010, Article ID 323429, 7 pages
doi:10.1155/2010/323429

Research Article

A Strategy for Automatic Quality Signing and Verification Processes for Hardware and Software Testing

Mohammed I. Younis and Kamal Z. Zamli

School of Electrical and Electronics, Universiti Sains Malaysia, 14300 Nibong Tebal, Malaysia

Correspondence should be addressed to Mohammed I. Younis, younismi@gmail.com

Received 14 June 2009; Revised 4 August 2009; Accepted 20 November 2009

Academic Editor: Phillip Laplante

We propose a novel strategy to optimize the test suite required for testing both hardware and software in a production line. Here, the strategy is based on two processes: Quality Signing Process and Quality Verification Process, respectively. Unlike earlier work, the proposed strategy is based on integration of black box and white box techniques in order to derive an optimum test suite during the Quality Signing Process. In this case, the generated optimal test suite significantly improves the Quality Verification Process. Considering both processes, the novelty of the proposed strategy is the fact that the optimization and reduction of test suite is performed by selecting only mutant killing test cases from cumulating t-way test cases. As such, the proposed strategy can potentially enhance the quality of product with minimal cost in terms of overall resource usage and time execution. As a case study, this paper describes the step-by-step application of the strategy for testing a 4-bit Magnitude Comparator Integrated Circuits in a production line. Comparatively, our result demonstrates that the proposed strategy outperforms the traditional block partitioning strategy with the mutant score of 100% to 90%, respectively, with the same number of test cases.

1. Introduction

In order to ensure acceptable quality and reliability of any embedded engineering products, many inputs parameters as well as software/hardware configurations need to be tested against for conformance. If the input combinations are large, exhaustive testing is next to impossible due to combinatorial explosion problem.

As illustration, consider the following small-scale product, a 4-bit Magnitude Comparator IC. Here, the Magnitude Comparator IC consists of 8 bits for inputs and 3 bits for outputs. It is clear that each IC requires 256 test cases for exhaustive testing. Assuming that each test case takes one second to run and be observed, the testing time for each IC is 256 seconds. If there is a need to test one million chips, the testing process will take more than 8 years using a single line of test.

Now, let us assume that we received an order of delivery for one million qualified (i.e., tested) chips within two weeks. As an option, we can do parallel testing. However, parallel testing can be expensive due to the need for 212 testing lines. Now, what if there are simultaneous multiple orders? Here, as the product demand grows in numbers, parallel testing can also become impossible. Systematic random testing could also be another option. In random testing, test cases are chosen randomly from some input distribution (such as a uniform distribution) without exploiting information from the specification or previously chosen test cases. More recent results have favored partition testing over random testing in many practical cases. In all cases, random testing is found to be less effective than the investigated partition testing methods [1].

A systematic solution to this problem is 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 parameter interaction coverage (t) [2]. To illustrate the CIT approach, consider the web-based system example (see Table 1) [3].

Considering full strength interaction $t = 4$ (i.e., interaction of all parameters) for testing yields exhaustive combinations of $3^4 = 81$ possibilities. Relaxing the interaction strength to $t = 3$ yields 27 test cases, a saving of nearly 67 percent. Here, all the 3-way interaction elements are all covered by at least

Table 1: Web-based system example.

Parameter 1	Parameter 2	Parameter 3	Parameter 4
Netscape	Windows XP	LAN	Sis
IE	Windows VISTA	PPP	Intel
Firefox	Windows 2008	ISDN	VIA

one test. If the interaction is relaxed further to $t = 2$, then the number of combination possibilities is reduced even further to merely 9 test cases, a saving of over 90 percent.

In the last decade, CIT strategies were focused on 2-way (pairwise) testing. More recently, several strategies (e.g., Jenny [4], TVG [5], IPOG [6], IPOD [7], IPOF [8], DDA [9], and GMIPOG [10]) that can generate test suite for high degree interaction ($2 \leq t \leq 6$).

Being predominantly black box, CIT strategy is often criticized for not being efficiently effective for highly interacting parameter coverage. Here, the selected test cases sometimes give poor coverage due to the wrong selection of parameter strength. In order to address this issue, we propose to integrate the CIT strategy with that of fault injection strategy. With such integration, we hope to effectively measure the effectiveness of the test suite with the selection of any particular parameter strength. Here, the optimal test case can be selected as the candidate of the test suite only if it can help detect the occurrence of the injected fault. In this manner, the desired test suite is the most optimum for evaluating the *System Under Test* (SUT).

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 and fault injection tools. Section 3 presents the proposed minimization strategy. Section 4 gives a step-by-step example as prove of concept involving the 4-bit Magnitude Comparator. Section 5 demonstrates the comparison with our proposed strategy and the traditional block partitioning strategy. Finally, Section 6 describes our conclusion and suggestion for future work.

2. Related Work

Mandl was the first researcher who used pairwise coverage in the software industry. In his work, Mandl adopts orthogonal Latin square for testing an Ada compiler [11]. Berling and Runeson use interaction testing to identify real and false targets in target identification system [12]. Lazic and Velasevic employed interaction testing on modeling and simulation for automated target-tracking radar system [13]. White has also applied the technique to test graphical user interfaces (GUIs) [14]. Other applications of interaction testing include regression testing through the graphical user interface [15] and fault localization [16, 17]. While earlier work has indicated that pairwise testing (i.e., based on 2-way interaction of variables) can be effective to detect most faults in a typical software system, a counter argument suggests such conclusion infeasible to generalize to all software system faults. For example, a test set that covers all possible pairs of variable values can typically detect 50% to 75% of the faults in a program [18–20]. In other works it is found that 100% of faults are detectable by a relatively low degree of interaction, typically 4-way combinations [21–23].

More recently, a study by *The National Institute of Standards and Technology* (NIST) for error-detection rates in four application domains included medical devices, a Web browser, an HTTP server, and a NASA-distributed database reported that 95% of the actual faults on the test software involve 4-way interaction [24, 25]. In fact, according to the recommendation from NIST, almost all of the faults detected with 6-way interaction. Thus, as this example illustrates, system faults caused by variable interactions may also span more than two parameters, up to 6-way interaction for moderate systems.

All the aforementioned related work in CIT applications highlighted the potential of adopting the CIT strategies for both software/hardware testing. While the CIT strategies can significantly partition the exhaustive test space into manageable manner, additional reduction can still be possible particularly by systematically examining the effectiveness of each test case in the test suite, that is, by exploiting fault injection techniques.

The use of fault injection techniques for software and hardware testing is not new. Tang and Chen [26], Boroday [27], and Chandra et al. [28] study circuit testing in hardware environment, proposing test coverage that includes each 2^t of the input settings for each subset of t inputs. Seroussi and Bshouty [29] give a comprehensive treatment for circuit testing. Dumer [30] examines the related question of isolating memory faults and uses binary covering arrays. 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 [31]. These studies help in the long-term improvement of the software development process as the recurrence of the same failures can be prevented. Failure modes can be specific to a system or be applicable to systems in general. They can be used in testing for fault tolerance, as realistic faults are needed to perform effective fault injection testing. Additionally, Ghosh and Kelly also describe a technique that injects faults in Java software by manipulating the bytecode level for third party software components used by the developers.

3. Proposed Strategy

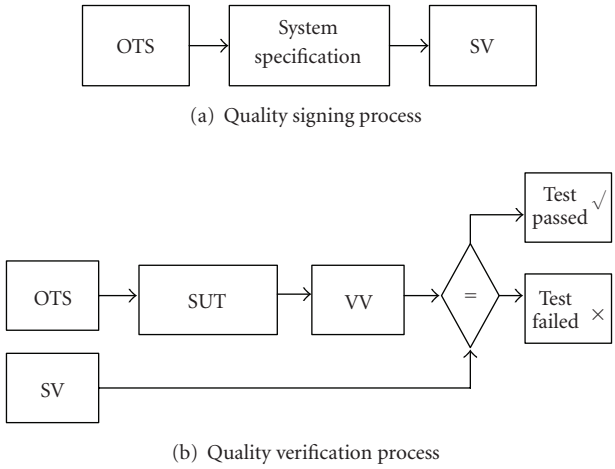
The proposed strategy consists for two processes, namely, *Test Quality Signing* (TQS) process and *Test Verification* process (TV). Briefly, the TQS process deals with optimizing the selection of test suite for fault injection as well as performs the actual injection whilst the TV process analyzes for conformance (see Figure 1).

As implied earlier, the TQS process aims to derive an effective and optimum test suite and works as follows.

(1) Start with an empty *Optimized Test Suite* (OTS), and empty *Signing Vector* (SV).

(2) Select the desired software class (for software testing). Alternatively, build an equivalent software class for the *Circuit Under Test* (CUT) (for hardware testing).

(3) Store these faults in *fault list* (FL).



(a) Quality signing process

(b) Quality verification process

Figure 1: The quality signing and verification processes.

(4) Inject the class with all possible faults.

(5) Let N be maximum number of parameters.

(6) Initialize CIT strategy with strength of coverage (t) equal one (i.e., $t = 1$).

(7) Let CIT strategy partition the exhaustive test space. The portioning involves generating one test case at a time for t coverage. If t coverage criteria are satisfied, then $t = t + 1$.

(8) CIT strategy generates one *Test Case* (TC).

(9) Execute TC.

(10) If TC detects any fault in FL, remove the detected fault(s) from FL, and add TC and its specification output(s) to OTS and SV, respectively.

(11) If FL is not empty or $t <= N$, go to 7.

(12) The desired optimized test suite and its corresponding output(s) are stored in OTS and SV, respectively.

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

(1) for $i = 1..$Size(OTS) each TC in OTS do:

  (a) Subject the SUT to TC[i], store the output in *Verification Vector* VV[i].

  (b) If VV[i] = SV [i], 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.

As noted in the second step of the TQS process, the rationale for taking equivalent software class for the CUT is to ensure that the cost and control of the fault injection be more practical and manageable as opposed to performing it directly to a real hardware circuit. Furthermore, the derivation of OTS is faster in software than in hardware. Despite using equivalent class for the CUT, this verification process should work for both software and hardware systems. In fact, it should be noted that the proposed strategy could also be applicable in the context of N-version programming (e.g., the assessment of student programs for the same assignment) and not just hardware production lines. The concept of N-version programming was introduced by Chen and Avizienis with the central conjecture that the "independence of programming efforts will greatly reduce the probability of identical software faults occurring in two or more versions of the program" [32, 33].

4. Case Study

As proof of concept, we have adopted GMIPOG [10] as our CIT strategy implementation, and MuJava version 3 (described in [34, 35]) as our fault injection strategy implementation.

Briefly, GMIPOG is a combinatorial test generator based on specified inputs and parameter interaction. Running on a Grid environment, GMIPOG adopts both the horizontal and vertical extension mechanism (i.e., similar to that of IPOG [6]) in order to derive the required test suite for a given interaction strength. While there are many useful combinatorial test generators in the literature (e.g., Jenny [3], TConfig [4], TVG [5], IPOG [6], IPOD [7], IPOF [8], DDA [9]), the rationale for choosing GMIPOG is the fact that it supports high degree of interaction and can be run in cumulative mode (i.e., support one-test-at-a-time approach with the capability to vary t automatically until the exhaustive testing is reached).

Complementary to GMIPOG, 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 [35].

Using both tools (i.e., GMIPOG 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 a0 and b0 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 Figure 2. Here, it should be noted that this version of the circuit is a variant realization (implementation) of the Magnitude Comparator found in [36]. The equivalent class of the Magnitude Comparator is given in Figure 3 (using the Java-programming language).

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 process; as illustrated in Section 3. Here, there are 80 faults injected in the system. To assist our work, we use GMIPOG [10] to produce the TC

Table 2: Derivation of OTS 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

Table 3: OTS and SV for the 4-bit Magnitude Comparator.

#TC	OTS TC (a0...a3, b0...b3)	SV 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

Table 4: Cumulative faults detected when $x = 7$.

#TC	TC (a0...a3, b0...b3)	Cumulative faults detected /80
1	FFFFFFFF	53
2	FFFFFTTT	54
3	FFFFTTTT	54
4	FTTTFFFF	59
5	FTTTFTTT	67
6	FTTTTTTT	70
7	TTTTFFFF	71
8	TTTTFTTT	71
9	TTTTTTTT	72

Table 5: Cumulative faults detected when x is randomly selective.

#TC	TC (a0...a3, b0...b3)	Cumulative faults detected /80
1	FFFFFFFF	53
2	FFFFFTTF	55
3	FFFFTTTT	55
4	TFTTFFFF	59
5	TFFTFTTT	61
6	TFTFTTTT	61
7	TTTTFFFF	61
8	TTTTTFFF	64
9	TTTTTTTT	72

in a cumulative mode. Following the steps in TQS process, Table 2 demonstrates the derivation of OTS. Here, it should be noted that the first 36 test cases can remove all the faults. Furthermore, only the first 12 test cases when $t = 4$ are needed to catch that last two live mutants. The efficiency of integration GMIPOG with MuJava can be observed (by taken only the effective TC) in the last column in Table 2.

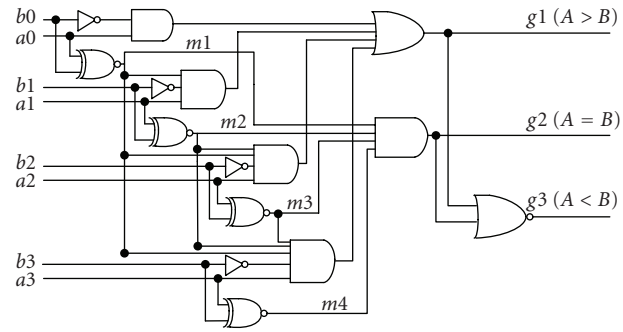


Figure 2: Schematic diagram for the 4-bit magnitude comparator.

Table 3 gives the desired OTS and SV, where 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 Table 2). 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).

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

```
public class Comparator {
//Comparator takes two four bits numbers (A&B), where A = a0a1a2a3
//B=b0b1b2b3. Here, a0 and b0 are the most significant bits.
//The function returns an output string that s
//g1, g2, and g3 represent the logical outputs of A > B, A = B, and A < B respectively.
//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; // just to return output strings for MuJava compatibility
              return s;
    }
}
```

Figure 3: Equivalent class Java program for the 4-bit magnitude comparator.

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 $256 - 9/256 = 96.48\%$.

5. Comparison

In this section, we demonstrate the possible test reduction using block partitioning approach [1, 37] for comparison purposes. Here, the partitions could be two 4-bit numbers, with block values =0, $0 < x < 15$, =15 and 9 test cases would give all combination coverage. In this case, we have chosen $x = 7$ as a representative value. Additionally, we have also run a series of 9 tests where x is chosen at random between 0 and 15. The results of the generated test cases and their corresponding cumulative faults detected are tabulated in Tables 4 and 5, respectively.

Referring to Tables 4 and 5, we observe that block partitioning techniques have achieved the mutant score of 90%. For comparative purposes, it should be noted that our proposed strategy achieved a mutant score of 100% with the same number of test cases.

6. Conclusion

In this paper, we present a novel strategy for automatic quality signing and verification technique for both hardware and software testing. Our case study in hardware production line demonstrated that the proposed strategy could improve the saving efforts factor significantly. In fact, we also demonstrate that our proposed strategy outperforms the traditional block partitioning strategy in terms of achieving better mutant score with the same number of test cases. As such, we can also potentially predict benefits in terms of the time and cost saving if the strategy is applied as part of software testing endeavor.

Despite giving a good result (i.e., as demonstrated in earlier sections), we foresee a number of difficulties as far as adopting mutation testing is concerned. In general, mutation testing does not scale well. Applying mutation testing in large programs can result in very large numbers of mutations making it difficult to find a good test suite to kill all the mutants. We are addressing this issue as part of our future work by dealing with variable strength interaction testing.

Finally, we also plan to investigate the application of our proposed strategy for computer-aided software application and hardware design tool.

Acknowledgments

The authors acknowledge the help of Jeff Offutt, Jeff Lei, Raghu Kacker, Rick Kuhn, Myra B. Cohen, and Sudipto Ghosh for providing them 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, USM GRID—The Development and Integration of Grid Services & Applications, and the fundamental research grants—"Investigating Heuristic Algorithm to Address Combinatorial Explosion Problem" from the Ministry of Higher Education (MOHE). The first author, Mohammed I. Younis, is the USM fellowship recipient.

References

[1] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," Tech. Rep. ISETR-04-05, GMU, July 2004.

[2] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "Algebraic strategy to generate pairwise test set for prime number parameters and variables," in *Proceedings of the International Symposium on Information Technology (ITSim '08)*, vol. 4, pp. 1662–1666, IEEE Press, Kuala Lumpur, Malaysia, August 2008.

[3] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "IRPS: an efficient test data generation strategy for pairwise testing," in *Proceedings of the 12th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES '08)*, vol. 5177 of *Lecture Notes in Computer Science*, pp. 493–500, 2008.

[4] Jenny tool, June 2009, http://www.burtleburtle.net/bob/math/.

[5] TVG tool, June 2009, http://sourceforge.net/projects/tvg/.

[6] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: a general strategy for T-way software testing," in *Proceedings of the International Symposium and Workshop on Engineering of Computer Based Systems*, pp. 549–556, Tucson, Ariz, USA, March 2007.

[7] 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, no. 3, pp. 125–148, 2008.

[8] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," *Journal of Research of the National Institute of Standards and Technology*, vol. 113, no. 5, pp. 287–297, 2008.

[9] R. C. Bryce and C. J. Colbourn, "A density-based greedy algorithm for higher strength covering arrays," *Software Testing Verification and Reliability*, vol. 19, no. 1, pp. 37–53, 2009.

[10] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "A strategy for grid based T-Way test data generation," in *Proceedings the 1st IEEE International Conference on Distributed Frameworks and Application (DFmA '08)*, pp. 73–78, Penang, Malaysia, October 2008.

[11] R. Mandl, "Orthogonal latin squares: an application of experiment design to compiler testing," *Communications of the ACM*, vol. 28, no. 10, pp. 1054–1058, 1985.

[12] T. Berling and P. Runeson, "Efficient evaluation of multifactor dependent system performance using fractional factorial design," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 769–781, 2003.

[13] L. Lazic and D. Velasevic, "Applying simulation and design of experiments to the embedded software testing process," *Software Testing Verification and Reliability*, vol. 14, no. 4, pp. 257–282, 2004.

[14] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '00)*, pp. 110–121, IEEE Computer Society, San Jose, Calif, USA, 2000.

[15] A. M. Memon and M. L. Soffa, "Regression testing of GUIs," in *Proceedings of the 9th Joint European Software Engineering Conference (ESEC) and the 11th SIGSOFT Symposium on the Foundations of Software Engineering (FSE-11)*, pp. 118–127, ACM, September 2003.

[16] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 20–34, 2006.

[17] M. S. Reorda, Z. Peng, and M. Violanate, Eds., *System-Level Test and Validation of Hardware/Software Systems*, Advanced Microelectronics Series, Springer, London, UK, 2005.

[18] R. Brownlie, J. Prowse, and M. S. Phadke, "Robust testing of AT&T PMX/StarMail using OATS," *AT&T Technical Journal*, vol. 71, no. 3, pp. 41–47, 1992.

[19] S. R. Dalal, A. Jain, N. Karunanithi, et al., "Model-based testing in practice," in *Proceedings of the International Conference on Software Engineering*, pp. 285–294, 1999.

[20] K.-C. Tai and Y. Lei, "A test generation strategy for pairwise testing," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 109–111, 2002.

[21] D. R. Wallace and D. R. Kuhn, "Failure modes in medical device software: an analysis of 15 years of recall data," *International Journal of Reliability, Quality, and Safety Engineering*, vol. 8, no. 4, pp. 351–371, 2001.

[22] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Proceedings of the 27th NASA/IEEE Software Engineering Workshop*, pp. 91–95, IEEE Computer Society, December 2002.

[23] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr., "Software fault interactions and implications for software testing," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[24] D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop (SEW '06)*, pp. 153–158, April 2006.

[25] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: beyond pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, 2008.

[26] D. T. Tang and C. L. Chen, "Iterative exhaustive pattern generation for logic testing," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 212–219, 1984.

[27] S. Y. Boroday, "Determining essential arguments of Boolean functions," in *Proceedings of the International Conference on Industrial Mathematics (ICIM '98)*, pp. 59–61, Taganrog, Russia, 1998.

[28] A. K. Chandra, L. T. Kou, G. Markowsky, and S. Zaks, "On sets of Boolean n-vectors with all k-projections surjective," *Acta Informatica*, vol. 20, no. 1, pp. 103–111, 1983.

[29] G. Seroussi and N. H. Bshouty, "Vector sets for exhaustive testing of logic circuits," *IEEE Transactions on Information Theory*, vol. 34, no. 3, pp. 513–522, 1988.

[30] I. I. Dumer, "Asymptotically optimal codes correcting memory defects of fixed multiplicity," *Problemy Peredachi Informatskii*, vol. 25, pp. 3–20, 1989.

[31] S. Ghosh and J. L. Kelly, "Bytecode fault injection for Java software," *Journal of Systems and Software*, vol. 81, no. 11, pp. 2034–2043, 2008.

[32] A. A. Avizienis, *The Methodology of N-Version Programming*, Software Fault Tolerance, John Wiley & Sons, New York, NY, USA, 1995.

[33] L. Chen and A. Avizienis, "N-version programming: a fault-tolerance approach to reliability of software operation," in *Proceedings of the 18th IEEE International Symposium on Fault-Tolerant Computing*, pp. 3–9, 1995.

[34] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: an automated class mutation system," *Software Testing Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[35] MuJava Version 3, June 2009, http://cs.gmu.edu/~offutt/mujava/.

[36] M. M. Mano, *Digital Design*, Prentice Hall, Upper Saddle River, NJ, USA, 3rd edition, 2002.
[37] L. Copeland, *A Practitioner's Guide to Software Test Design*, STQE Publishing, Norwood, Mass, USA, 2004.