2009 IEEE Symposium on Industrial Electronics and Applications (ISIEA 2009), October 4-6, 2009, Kuala Lumpur, Malaysia

Assessing Combinatorial Interaction Strategy for Reverse Engineering of Combinational Circuits

Mohammed I. Younis
School of Electrical and Electronics
Universiti Sains Malaysia
14300 Nibong Tebal, Penang, Malaysia
Email: younismi@gmail.com

Kamal Z. Zamli
School of Electrical and Electronics
Universiti Sains Malaysia
14300 Nibong Tebal, Penang, Malaysia
Email: eekamal@eng.usm.my

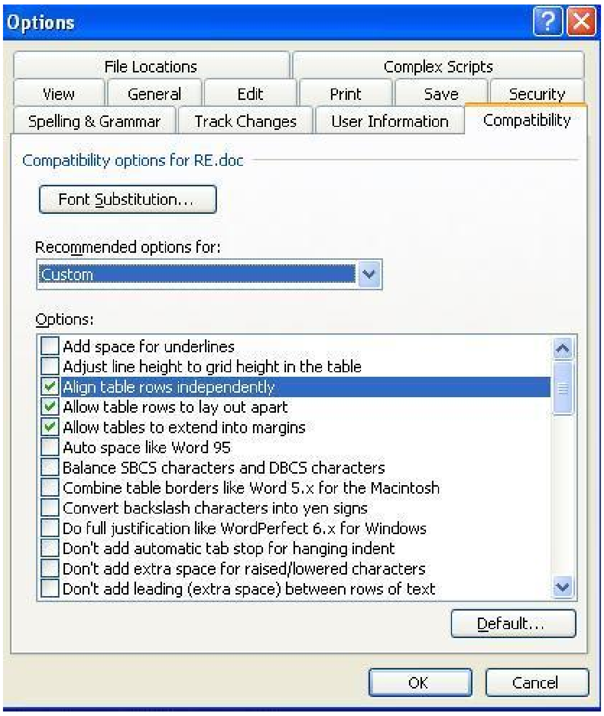
***Abstract*—T-way test data generators play an immensely important role for both hardware and software configuration testing. Earlier work concludes that t-way test data generator can achieve 100% coverage without having to regard for more than 6 way interactions. In this paper, we investigate whether or not such a conclusion can be applicable for reverse engineering of combinational circuits. In this case, we reverse engineer a faulty commercial eight segment display controller using our t-way test data generator in order to redesign the replacement unit. We believe that our application of t-way generators for circuit identification is novel. The results demonstrate the need of more than 6 parameter interactions as well as suggest the effectiveness of cumulative test data for reverse engineering applications.**

***Keywords*—Combinational Circuits, Test Data Generation, T-Way Testing, Combinatorial Interaction, Configuration Testing, Multi-Way Testing**

I. INTRODUCTION

In order to ensure software coverage and reliability, many combinations of possible input parameters, embedded system (i.e. hardware/software) environments, and system configurations need to be tested and verified against for conformance. Although desirable, exhaustive software testing is next to impossible due to resources as well as timing constraints. As illustration, consider the option dialog Microsoft Word software (see Fig. 1). Even if only Compatibility tab option is considered, there are already 62 possible configurations to be tested. With the exception of Font Substitution and Recommended Options which take 5 and 13 possible values respectively, each configuration can take two values (i.e. checked or unchecked). Here, there are 2^{62}x5x13 combinations of test cases to be evaluated. Assuming that it takes only one second for one test case, then it would require nearly 9x10^{13} years for a complete test of the Compatibility tab option.

Similar situation can be observed when testing hardware product. As a simple example, consider a hardware product with 20 on/off switches. To test all possible combination would require 2^{20} = 1,048,576 test cases. If the time required for one test case is 5 minutes, then it would take nearly 10 years for a complete test [1]. Obviously, there is a need for a systematic strategy in order to reduce the test data set into manageable ones.



Compatibility options for RE.doc

Recommended options for: Custom

Options:

☐ Add space for underlines
☐ Adjust line height to grid height in the table
☑ Align table rows independently
☑ Allow table rows to lay out apart
☑ Allow tables to extend into margins
☐ Auto space like Word 95
☐ Balance SBCS characters and DBCS characters
☐ Combine table borders like Word 5.x for the Macintosh
☐ Convert backslash characters into yen signs
☐ Do full justification like WordPerfect 6.x for Windows
☐ Don't add automatic tab stop for hanging indent
☐ Don't add extra space for raised/lowered characters
☐ Don't add leading (extra space) between rows of text

Figure 1. Compatibility Option Dialogue for Microsoft Word

The systematic solution to this problem is based on N-way or (t-way) testing strategy for generating test suite. N-way testing 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 interaction coverage. CIT strategies had been focused on 2-way (pairwise) testing in the last decade. More recently, there are several strategies that can be generated for high degree interaction $(2 \le t \le 6)$ ITCH [2], Jenny [3], TConfig [4], TVG [5] IPOG[6] , IPOD[7], IPOF[8] DDA[9]. Finally, GMIPOG [10] is reported as a strategy that supports very high degree of interaction $(1 \le t \le 12)$.

In this paper we propose a new application of CIT. This new application involves the use of the CIT for reverse engineering of combinational circuit. The remaining 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 a detailed case study with step by step example as a proving of concept. Section 4 gives the lessons learned from our experiment. Finally,

978-1-4244-4683-4/09/$25.00 ©2009 IEEE

section 5 states the conclusion and suggestion for future work.

II. RELATED WORK

Mandl appears to be the first researcher to use pair-wise coverage in the software industry. He uses 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]. Lazi´c and Velaˇsevi´c 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 (GUI) [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 cannot be generalized 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] [19] [20]. In other work it is found that 100% of faults detectable by a relatively low degree of interaction, typically 4-way combinations [21] [22] [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.

Tang et al. [26], Boroday et al. [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 Bshouti [29] give a comprehensive treatment for circuit testing. Finally, Dumer [30] examines the related question of isolating memory faults, and uses binary covering arrays.

III. EXPERIMENT

In this section, we consider the assistance of t-way parameter interaction generator for reverse engineering (i.e. to build an equivalent circuit) of combinational design. Here, we consider eight segment display controller that is used by an entry RFID system in our university. The controller takes eight logical inputs lines (I1...I8) from the circuit board and produces eight logical outputs lines (O1…O8) to the eight segments display (see Fig. 2). Fig. 3 presents the 8-segment display.

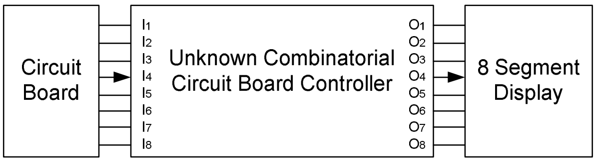


Figure 2. The Interface between Circuit Board and 8-Segments Display

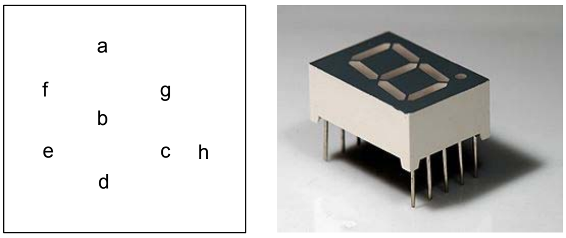


Figure 3. 8-Segments Display (7-segment with dot)

Using the eight segment display controller as a base design, we are to build equivalent controller using t-way test data generators. All we have is the correct outputs from the eight segments display which is summarized in Table 1. It should be noted that the output from table 1 is nonstandard for hexadecimal (e.g., contains both capital and small letters for a, and e hexadecimal digits). The outputs pins (O1, O2, O3, O4, O5, O6, O7, and O8) are connected directly to eight segments display (g, f, e, d, c, b, a, and h) respectively. Further more, any inputs (#) not stated in table 1 produced a constant output Error (or (11100000) corresponding to (gfedcbah) respectively). In this paper we use 0, and 1 to represent logic (0, and 1) or false, and true respectively. So, in order to design the combinational circuit, it is desirable to know (and predict) the inputs (I1…I8) values stated in index # in Table 1. In short, there are 37 distinct inputs caused 37 distinct outputs; all other inputs (256-37=219) caused error outputs.

The trivial approach is to test all possible combinations (equivalent to exhaustive testing), due to resource constraints, we propose partitioning testing in systematic manner. In doing so, we intent to generate test case in one test at a time fashion, that is, by constructing the input space (test case) in a cumulative manner starting from t=1, t=2, etc. The prediction process ends when all inputs are predicted.

The input space and the prediction process are given in Fig. 4. It should be noted that the stopping criteria for the test generator is when all inputs are detected (and corresponding outputs are observed). Here, we adopt GMIPOG [10] as our test generator. The results for t=1, 2, 3 are given in Tables 2, 3, 4 respectively. Table 5 represents the cumulative test set. The complete pseudo algorithm for our approach is given is given in Fig. 4.

By substituting the inputs values in Table 1 to Table 5 yields the desired truth table of the required circuit. Here, the problem of the logic design is solved using the reverse engineering manner. The problem is reduced by running only 196 cumulative test cases instead of running all the exhaustive testing of 256 test cases.

TABLE I. THE UNIQUE OUTPUTS FOR THE CONTROLLER

#	gfedcbah	Meaning	#	gfedcbah	Meaning	#	gfedcbah	Meaning
1	00000000	OFF	14	11111000	b, dot off	27	00001111	seven, dot on
2	01111110	zero, dot off	15	01110010	c, dot off	28	11111111	eight, dot on
3	00001100	one, dot off	16	10111100	d, dot off	29	11001111	nine, dot on
4	10110110	two, dot off	17	11110010	E, dot off	30	11101111	A, dot on
5	10011110	three, dot off	18	11110110	e, dot off	31	10111111	a, dot on
6	11001100	four, dot off	19	11100010	F, dot off	32	11111001	b, dot on
7	11011010	five, dot off	20	01111111	zero, dot on	33	01110011	c, dot on
8	11111010	six, dot off	21	00001101	one, dot on	34	10111101	d, dot on
9	00001110	seven, dot off	22	10110111	two, dot on	35	11110011	E, dot on
10	11111110	eight, dot off	23	10011111	three, dot on	36	11110111	e, dot on
11	11001110	nine, dot off	24	11001101	four, dot on	37	11100011	F, dot on
12	11101110	A, dot off	25	11011011	five, dot on			
13	10111110	a, dot off	26	11111011	six, dot on			

```
1. Starting from empty test set (ts), and 37 desired outputs set given in table 1 (od).
2. Let t=1.
3. Take one generated test case (tg), with t strength of coverage.
4. if tg not in ts , add tg to ts, and observe the output (oo). If (oo) hits one of the desired outputs  in od. delete oo from od. If od  is empty goto 8
5. if there is next  test case (tg), with t strength of coverage, goto
6. t=t+1
7. goto 3.
8. Display the predicted inputs, outputs table.
9. end.
```

Figure 4. The Cumulative Test Case with Prediction Process

TABLE II. TEST SUITE FOR 1-WAY INTERACTION

case	Test Case	case	Test Case
1	00000000	2	11111111

TABLE III. TEST SUITE FOR 2-WAY INTERACTION

case	Test Case	case	Test Case	case	Test Case	case	Test Case
1	00000000	3	10101010	5	01001011	7	00010011
2	01111111	4	11010101	6	10110100	8	11101100

TABLE IV. TEST SUITE FOR 3-WAY INTERACTION

case	Test Case	case	Test Case	case	Test Case	case	Test Case
1	00000000	6	10100110	11	00001111	16	11011110
2	00111111	7	11001100	12	11111000	17	00001100
3	01010101	8	11110011	13	00101101	18	11111111
4	01101010	9	00110000	14	11010000		
5	10011001	10	11000011	15	00010011		

TABLE V. TEST CASE FOR CUMULATIVE TESTING

tg	Test Case	#	tg	Test Case	#	tg	Test Case	#	tg	Test Case	#
1	00000000	24	50	01001001	r	99	11100110	r	148	11001101	34
2	11111111	15	51	11101101	r	100	11001110	36	149	11100011	r
3	01111111	25	52	10010010	r	101	10000110	r	150	11100100	r
4	10101010	r	53	10110111	7	102	10111011	r	151	11101110	35
5	11010101	r	54	11111011	28	103	11010011	r	152	11111010	2
6	01001011	19	55	10110110	6	104	11110010	23	153	00000110	r
7	10110100	r	56	10011111	13	105	11011010	37	154	00100010	r
8	00010011	r	57	11011101	r	106	00011000	r	155	00110110	r
9	11101100	r	58	00001110	3	107	10101100	r	156	01010000	r
10	00111111	r	59	00011011	r	108	01011110	r	157	01001110	r
11	01010101	r	60	01100000	r	109	00001001	r	158	10011010	r
12	01101010	r	61	11011011	29	110	00100011	r	159	10010000	r
13	10011001	r	62	00010101	r	111	10111101	30	160	10101000	r
14	10100110	r	63	00011010	r	112	00110001	r	161	10111110	14
15	11001100	31	64	00100110	r	113	11000101	r	162	11100010	5
16	11110011	16	65	00101001	r	114	11111101	r	163	11110110	21
17	00110000	r	66	00110011	r	115	10100001	r	164	11000110	r
18	11000011	r	67	00111100	r	116	00001010	r	165	11111100	r
19	00001111	20	68	01000111	r	117	00001101	18	166	11011000	r
20	11111000	32	69	01001000	r	118	00010100	r	167	00010010	r
21	00101101	r	70	01010010	r	119	00011001	r	168	00000001	r
22	11010000	r	71	01011101	r	120	00011110	r	169	10000011	r
23	11011110	r	72	01101110	r	121	00100100	r	170	00101111	r
24	00001100	10	73	01110100	r	122	00101110	r	171	00111011	r
25	00011111	r	74	01111011	r	123	00110111	r	172	01010111	r
26	00101010	r	75	10000100	r	124	00111010	r	173	01000011	r
27	00110101	r	76	10001011	r	125	00111101	r	174	11001011	r
28	01001100	r	77	10011110	1	126	01000010	r	175	11001111	33
29	01010011	r	78	10100010	r	127	01000101	r	176	10100111	r
30	01100110	r	79	10101101	r	128	01001111	r	177	10110011	r
31	01111001	r	80	10111000	r	129	01010110	r	178	01110011	12
32	10001110	r	81	11010110	r	130	01011011	r	179	11011111	r
33	10010001	r	82	11011001	r	131	01011100	r	180	10001101	r
34	10100101	r	83	11100101	r	132	01101011	r	181	11110001	r
35	10111010	r	84	11101010	r	133	01101100	r	182	11010001	r
36	11011100	r	85	11111110	4	134	01110010	26	183	11111001	9
37	11101000	r	86	10001000	r	135	01110101	r	184	10010101	r
38	11110111	17	87	10011101	r	136	10000010	r	185	00000101	r
39	10001001	r	88	01010001	r	137	10000101	r	186	00010001	r
40	01011010	r	89	01101101	r	138	10001111	r	187	00010111	r
41	00110010	r	90	00100101	r	139	10010110	r	188	00011101	r
42	01100001	r	91	01111000	r	140	10011011	r	189	00100001	r
43	11110000	r	92	11000000	r	141	10011100	r	190	00100111	r
44	00000111	r	93	11101001	r	142	10101011	r	191	00101000	r
45	11010100	r	94	00000011	r	143	10110010	r	192	00101011	r
46	10111100	22	95	00010110	r	144	10110101	r	193	00111001	r
47	11101111	11	96	01001010	r	145	10111111	8	194	01000001	r
48	00011100	r	97	01110111	r	146	11000111	r	195	01000100	r
49	10111001	r	98	00111110	r	147	11001010	r	196	01001101	27

IV. LESSONS LEARNED FROM OUR EXPERIMENT

In our experiment, we use t-way test data generators to generate cumulative test data in order to partition the exhaustive testing in a systematic fashion. In doing so, we add one test at a time from the generator (see Table 5) to predict the inputs until all desired outputs (as given in Table 1) are observed. Table 6 demonstrates the partitioning process (in terms of number of coverage and the test size) going from 1-w and stopping when the last output is observed (i.e. at tg equal 196 in Table 5). Fig. 5 shows the prediction rate for our case study.

TABLE VI. Cumulative Prediction (Coverage) Rate for t=1 to 7

Strength of Coverage(t)	Coverage /37	Test size	% Coverage
1	2	2	5.4
2	4	9	10.81
3	9	24	24.234
4	18	61	48.65
5	24	115	64.9
6	36	184	97.3
7	37	196	100

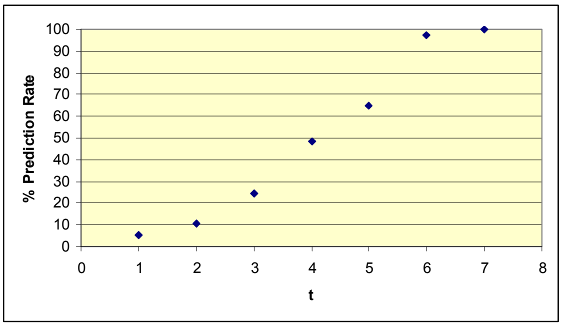


Figure 5. Prediction (Coverage) Rate for the Case Study

Unlike other studies, here we need the degree of interaction higher than 6 in order to have a full 100% coverage. In our case, we need the interaction of 7 even though we adopt a cumulative generation process.

To demonstrates the effectiveness of our cumulative approach, we study the coverage rate, as well as the size, for each individual test case from t=1 to t=8. The result is summarized in Table 7. Here the full coverage (prediction) is not reached until 8-way interaction.

Based on the aforementioned results, we conclude that adopting cumulative approach is more practical than running individual testing. Also, it should be observed that the size of cumulative approach is high due to fact that test suite for lower degree of interaction is not necessarily a subset of the higher degree suite. For this reason, it is a good practice to generate and run small degree interaction suite before generation for the higher one. To illustrate this concept, consider the predicted input (11111000) which is found in test case 12, t=3 (see Table 4, or test case 20 in table5). This test case is not found in the test suite when t=4, 5, 6, and 7 respectively.

TABLE VII. Coverage Rate and Test Size for t=1 to 8

Strength of Coverage(t)	Coverage/37	Test size	% Coverage
1	2	2	5.4
2	3	8	8.1
3	7	18	18.9
4	11	41	29.73
5	12	70	32.43
6	18	117	48.6
7	20	128	54
8	37	256	100

V. CONCLUSION

In this paper, we present a novel approach to use t-way test data generator for reverse engineering. The use of cumulative testing plays important role to make the partitioning exhaustive testing more practical. Our case study demonstrated the requirement of higher degree interaction test suite. Our result demonstrates that the required degree interaction varied from system to system and even cumulative 6-way interaction can not cover 100%. As a part of our future work, we plan to investigate the application of t-way test data generation in different engineering fields.

ACKNOWLEDGMENT

The work described in this paper is funded by the eScience Fund – Development of a Mobile Agent Based Parallel and Automated Java Testing Tool" from the MOSTI.

REFERENCES

[1] 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 Proc. of the International Symposium on Information Technology, ITSim'08, IEEE Press, KLCC, Malaysia, August 2008, pp. 1662-1666.

[2] http://www.alphaworks.ibm.com/tech/whitch. Last accessed on August 6, 2009.

[3] http://www.burtleburtle.net/bob/math. Last accessed on August 6, 2009.

[4] http://www.site.uottawa.ca/~awilliam. Last accessed on August 6, 2009.

[5] http://sourceforge.net/projects/tvg. Last accessed on August 6, 2009.

[6] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A General Strategy for T Way Software Testing", in Proc. of the 14th Annual IEEE Intl. Conf. and Workshops on the Engineering of Computer-Based Systems, Tucson, AZ, March 2007, pp. 549-556.

[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, v 18(3), 2008, pp. 125-148.

[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, Volume 113, Number 5, October 2008. pp. 287-297.

[9] R.C. Bryce and C.J. Colbourn, “A density-based greedy algorithm for higher strength covering arrays”, Software Testing, Verification, and Reliability 19 (2009), pp. 37-53.

[10] M.I. Younis, Kamal Z. Zamli and Nor Ashidi Mat Isa, “A Strategy for Grid Based T-way Test Generation”, in Proc. of the First International Conference on Distributed Framework and Applications, (DFmA 2008), Penang, Malaysia, October 2008, pp 73-78.

[11]R. Mandl, “Orthogonal Latin Squares: an Application of Experiment Design to Compiler Testing”, Communications of the ACM, Vol. 28 (10), 1985, pp. 1054-1058.

[12] T. Berling, and P. Runeson, “Efficient evaluation of multifactor dependent system performance using fractional design”, IEEE Transactions on Software Engineering 2003; 29(9), pp.769–781.

[13] L.J Lazi´c, and D. Velaˇsevi´c, “Applying simulation and design of experiments to the embedded software testing process”, Software Testing, Verification and Reliability 2004, pp. 257–282.

[14] L. White, and H. Almezen, “Generating test cases for GUI responsibilities using complete interaction sequences”, in Proc. of the Intl Sym. on Software Reliability Engineering, San Jose, CA, 2000. IEEE Computer Society Press: Piscataway, NJ, 2000; pp.110–121.

[15] A.M. Memon, and M.L. Soffa, “Regression testing of GUIs”, in Proc. of the 9th European Software Engineering Conf. (ESEC) and 11th ACM SIGSOFT Intl. Sym. on the Foundations of Software Engineering (FSE-11), 1–5 September 2003. ACM Press: New York, 2003, pp. 118–127.

[16] C. Yilmaz, M.B. Cohen, and A.Porter. “Covering arrays for efficient fault characterization in complex configuration spaces”, IEEE Transactions on Software Engineering V31(1), 2006, pp. 20–34.

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

[18] R. Brownlie, J. Prowse, and M.S. Phadke, “Robust Testing of AT&T PMX/StarMail using OATS”, AT&T Technical Journal, 71(3), June 1992, pp. 41-47.

[19] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz, “Model-Based Testing in Practice”, in Proc. of the International Conference on Software Engineering, 1999.

[20] K.C. Tai, and Y. Lei, “A Test Generation Strategy for Pairwise Testing”, IEEE Trans. Software Eng. vol. 28, no. 1, January 2002 ,pp. 109-111.

[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, 2001.

[22] D.R. Kuhn, and M.J. Reilly, “An Investigation of the Applicability of Design of Experiments to Software Testing”, 27th NASA/IEEE Software Engineering Workshop, IEEE Computer Society, 4-6 December, 2002, pp. 91-95.

[23] D.R. Kuhn, D.R. Wallace, and A.M Gallo, “Software Fault Interactions and Implications for Software Testing”, IEEE Transactions on Software Engineering, 30(40):1-4 (2004)

[24] D. R. Kuhn and V. Okun, "Pseudo-exhaustive Testing For Software," in Proc. of the 30th NASA/IEEE Software Engineering Workshop, April, 2006, pp. 25-27.

[25] R. Kuhn, Y. Lei, and R. Kacker, Practical Combinatorial Testing: Beyond Pairwise, IEEE Computer Society, June 2008, pp. 19-23.

[26] D.T. Tang and C.L. Chen. “Iterative exhaustive pattern generation for logic testing”, IBM Journal Research and Development 28 (1984), 212-219.

[27] S.Y. Boroday, “Determining essential arguments of Boolean functions” (Russian), in Proc. Conference on Industrial Mathematics, Taganrog, 1998, pp. 59-61.

[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 20 (1983), pp. 103-111.

[29] G. Seroussi and N. H. Bshouty, “Vector sets for exhaustive testing of logic circuits”, IEEE Transactions on Information Theory 34 (1988), pp. 513-522.

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