# Grid_MIPOG: A Loosely Coupled T_way Strategy for Combinatorial Testing

**Mohammed I Younis[1*], Kamal Z Zamli[2]**

**Abstracts:** This paper discusses variant implementations of an MIPOG strategy based on our earlier work. Here, we adopt more scalable design and implementation for the MIPOG strategy. In doing so, a grid-based implementation is proposed; namely Grid_MIPOG. Unlike the previous implementation of MIPOG and MC_MIPOG, GRID_MIPOG utilizes the computing power from loosely coupled machines. Here, Grid_MIPOG provides more speedup and better memory distribution as far as the numbers of parameters, number of variables, and the strength of coverage are concerned.

**Key Words:** Interaction testing, combinatorial testing, parallel processing, distributed computing, parallel algorithms.

## INTRODUCTION

*T-way* testing has been actively investigated for both software and hardware testing due to its ability to efficiently minimize the test size systematically. However, one challenge in this area is dealing with the combinatorial explosion problem, which typically requires a very expensive computational process to find a good test set that covers all the combinations for a given interaction strength (t).

As an illustration of combinatorial explosion problem, consider the option dialog Microsoft Internet Explorer software (see Fig. 1). Even if only "Advanced" tab option is considered, there are already 54 possible configurations to be tested. With the exception of searching and under line links which take 4 and 3 possible values respectively, each configuration can take two values (i.e. checked or unchecked). Here, there are $2^{54}$x4x3 combinations of test cases to evaluate. Assuming that it takes only one second for one test case, then it would require nearly 68x$10^7$ years for a complete test of the "Advanced" tab option.

Similar situation can be observed when testing a typical hardware product. As a simple example, consider a hardware product with 30 on/off switches. To test all possible combination would require $2^{30}$ test cases. If the time required for one test case is one second, then it would take nearly 34 years for a complete test. Furthermore, in hardware testing it is required to test each instance of hardware unit independently. For instance, suppose a hardware production line requires testing one million switches. It is clear that testing every possible combination is impossible due to resources as well as time to market constraints. Thus, there is a need of a sampling strategy for software and hardware testing.

The simplest approach tests all values at least once. The most thorough approach exhaustively tests all parameter-value combinations. While testing only individual values may not be enough, exhaustive testing of all possible combinations is not always feasible. *t*-way strategies (i.e. as sampling strategies) are a reasonable alternative that falls in between these two extremes [1]. Here, the t-way strategy (also termed Combinatorial Interaction Testing) can systematically reduce the number of test data by selecting a subset from exhaustive testing combination based on the strength of interaction coverage (t) such that each t-wise combination of variables is covered at least once.

Consider a cross platform testing to be performed for an RFID-based attendance system. The system has four configuration parameters of interest as shown in Table 1. There are three CPU types; three types of OS; three types of browsers; and three Internet connectivity options. To exhaustively test all combinations of the four configuration parameters that have 3 options each from Table 1 would require $3^4$ = 81 tests. The specification of the cross platform testing is stated that all pairs of combinations should combine together at least once during the testing process. As such, instead of testing every combination, all individual pairs of interactions are tested. The resulting test suite is shown in Table 2. For instance, the first test from Table 2 covers the following pairs: (Intel, Windows), (Intel, Safari), (Intel, LAN), (Windows, Safari), (Windows, LAN), and (Safari, LAN). The entire test suite covers every possible 2-way combination between components. In this example, exhaustive testing requires 81 test cases, but 2-way testing requires merely 9 test cases. Even though, for a small system configuration, reduction from 81 to merely 9 test cases is not that impressive, but consider a larger example: a manufacturing automation system that has 20 controls, each with 10 possible settings, a total of $10^{20}$ combinations, which is far more than a software tester would be able to test in a lifetime. Surprisingly, all pairs of these values can be checked with only 180 tests if they are carefully constructed [2].

Earlier studies (e.g. in [3, 4] have suggested that pairwise testing (i.e. based on two-way interaction of variables) are effective in detecting most faults in a typical software system. While such conclusion may be true for some systems, it cannot be generalized to all faults found in a software system, especially when there are significant interactions between variables. The National Institute of Standards and Technology (NIST) investigated the application of interaction testing for four 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 [5, 6]. In fact, according to the recommendation from NIST, almost all of the faults detected with 6-way interaction [7]. More recently, Younis and Zamli presented an approach to use interaction testing for t-way testing in reverse engineering of combinational circuit [8]. Unlike the NIST study, Younis and Zamli demonstrate the requirement of higher degree interaction test suite (i.e., t=7).

Considering more than two parameter interaction is not without difficulties. When the parameter interaction coverage t increases (i.e. as t > 2), the number of t-way tuples, and hence the number of test cases, also increases exponentially (see Table 3).

Although useful works [6, 9-12] have been done in the recent years, these works are mainly sequential in nature. Parallelization can be an effective approach to manage the aforementioned computational cost, that is, by taking advantage of the recent advancement of multi-core architectures in both single machine (tightly coupled CPUs) and loosely coupled machines (loosely coupled CPUs).

In our previous work, we have proposed and evaluated MIPOG [13] and its multi-threaded (concurrent) version (i.e., target to multi-core CPUs lay in a single machine); namely: MC_MIPOG [12]. Complementing and building from the earlier work, this paper proposes an alternative parallelization of the MIPOG; namely: Grid _MIPOG. Unlike MC_MIPOG, Grid_MIPOG is targeted toward loosely coupled machines (e.g., Grid). In this paper, we report the speedup gain of MC_MIPOG and GRID_MIPOG against the sequential MIPOG with the increase of CPUs as computational nodes. The remaining of this paper is organized as follows. Section 2 highlights some related works in t-way testing strategies. Section 3 gives an overview of MIPOG and MC_MIPOG. Section 4 gives the design criteria for Grid_MIPOG and the details of its implementation. Section 5 contains the evaluation section of the speedup gains for the proposed strategy. Finally, Section 6 states our conclusion and suggestion for future work.

## Related Works

The problem of generation test suite that covers t-way tuples at least is considered as NP-complete problem [14, 15]. That is, there is no unique solution to the problem. For this reason, many strategies do exist in the literature. These strategies adopt either algebraic or computational approaches [4, 16].

[1]Department of Computer Engineering, College of Engineering, University of Baghdad, Baghdad, Iraq.
E-mail: younismi@gmail.com
*Corresponding author

[2]School of Electrical and Electronics Engineering, Universiti Sains Malaysia, Penang, Malaysia.

**Table 1: Four configuration parameters that have three possible settings each for an RFID-based Attendance system**

| CPU | OS | Browser | Internet connectivity |
|---|---|---|---|
| Intel | Windows | Safari | LAN |
| AMD | Linux | Firefox | Wireless-LAN |
| Cyrix | Mac | Opera | Modem |

**Table 2: A pair-wise combinatorial test configuration for an RFID-based Attendance system**

| Test No. | CPU | OS | Browser | Internet connectivity |
|---|---|---|---|---|
| 1 | Intel | Windows | Safari | LAN |
| 2 | Intel | Linux | Firefox | Wireless-LAN |
| 3 | Intel | Mac | Opera | Modem |
| 4 | AMD | Windows | Opera | Wireless-LAN |
| 5 | AMD | Linux | Safari | Modem |
| 6 | AMD | Mac | Firefox | LAN |
| 7 | Cyrix | Windows | Firefox | Modem |
| 8 | Cyrix | Linux | Opera | LAN |
| 9 | Cyrix | Mac | Safari | Wireless-LAN |

**Table 3: Number of Tuples and test cases for varying t for a system with 10-5 parameter-valued**

| t | # Tuples | # Test cases |
|---|---|---|
| 2 | 1125 | 45 |
| 3 | 15000 | 281 |
| 4 | 131250 | 1643 |
| 5 | 787500 | 8169 |
| 6 | 3281250 | 45168 |
| 7 | 9375000 | 186664 |

**Table 4: $\pi$ set data structure for i=4**

| Parameter-Control | Tuples-set | P4-value |
|---|---|---|
| 110 | 00<br>01<br>10<br>11 | 0 |
| 101 | 00<br>01<br>10<br>11 | |
| 011 | 00<br>01<br>10<br>11 | |
| 110 | 00<br>01<br>10<br>11 | 1 |
| 101 | 00<br>01<br>10<br>11 | |
| 011 | 00<br>01<br>10<br>11 | |
| 110 | 00<br>01<br>10<br>11 | 2 |
| 101 | 00<br>01<br>10<br>11 | |
| 011 | 00<br>01<br>10<br>11 | |

Table 5: $\pi$ set tuples for i=4 , after deleting tuples covered by the first test case

| Parameter-Control | Tuples-set | P4-value |
|---|---|---|
| 110 | 01<br>10<br>11 | 0 |
| 101 | 01<br>10<br>11 | |
| 011 | 01<br>10<br>11 | |
| 110 | 00<br>01<br>10<br>11 | 1 |
| 101 | 00<br>01<br>10<br>11 | |
| 011 | 00<br>01<br>10<br>11 | |
| 110 | 00<br>01<br>10<br>11 | 2 |
| 101 | 00<br>01<br>10<br>11 | |
| 011 | 00<br>01<br>10<br>11 | |

Table 6: $\pi$ set tuples for i=4 , after horizontal extension

| Parameter-Control | Tuples-set | P4-value |
|---|---|---|
| | | 0 |
| | | 1 |
| 110 | 00<br>01<br>10<br>11 | 2 |
| 101 | 00<br>01<br>10<br>11 | |
| 011 | 00<br>01<br>10<br>11 | |

Table 7: Speedup Gain Assessment for Group 1: CA (Size, t: 2..7, 10, 5)

| t | Size | MIPOG | MC_MIPOG | | Grid_MIPOG | | |
|---|---|---|---|---|---|---|---|
| | | Time | Time | S_up | Time | W/M | S_up |
| 2 | 45 | 0.074 | 0.09 | 0.822 | 0.098 | 5/2 | 0.755 |
| 3 | 281 | 0.327 | 0.281 | 1.163 | 0.317 | 5/2 | 1.03 |
| 4 | 1643 | 6.9 | 3.818 | 1.807 | 3.575 | 5/2 | 1.93 |
| 5 | 8169 | 44.3 | 21.146 | 2.095 | 13.758 | 5/2 | 3.22 |
| 6 | 45168 | 4025.442 | 1311.209 | 3.07 | 1056.546 | 5/2 | 3.81 |
| 7 | 186664 | 82668.19 | 23512.7 | 3.516 | 18049.823 | 5/2 | 4.58 |

Most algebraic approaches compute test sets directly by a mathematical function [17]. The construction of test suite by means of pure algebraic approaches (i.e. without searching the tuples space) can be achieved either by applying successive transformations to well known array or by using a product of construction [18, 19]. For this reason, algebraic approaches often impose restrictions on the system configurations to which they can be applied [20]. This significantly limits the applicability of algebraic approaches for software testing [21].

Unlike algebraic approaches, computational approaches often rely on the

**Table 8: Speedup Gain Assessment for Group 2: CA (Size, 4, P: 5..15, 5)**

| P | Size | MIPOG | MC_MIPOG | | Grid_MIPOG | | |
|---|---|---|---|---|---|---|---|
| | | Time | Time | S_up | Time | W/M | S_up |
| 5 | 625 | 0.128 | 0.15 | 0.8533 | 0.162 | 5/2 | 0.79 |
| 6 | 625 | 0.31 | 0.269 | 1.152 | 0.276 | 5/2 | 1.12 |
| 7 | 1125 | 0.778 | 0.57 | 1.365 | 0.563 | 5/2 | 1.38 |
| 8 | 1348 | 1.981 | 1.272 | 1.557 | 1.294 | 5/2 | 1.53 |
| 9 | 1543 | 3.735 | 2.275 | 1.642 | 2.197 | 5/2 | 1.7 |
| 10 | 1643 | 6.9 | 3.818 | 1.807 | 3.575 | 5/2 | 1.93 |
| 11 | 1722 | 10.642 | 5.803 | 1.833 | 5.216 | 5/2 | 2.04 |
| 12 | 1837 | 19.39 | 10.298 | 1.883 | 8.394 | 5/2 | 2.31 |
| 13 | 1956 | 44.169 | 21.171 | 2.086 | 16.605 | 5/2 | 2.66 |
| 14 | 2051 | 71.104 | 33.213 | 2.14 | 23.78 | 5/2 | 2.99 |
| 15 | 2150 | 143.29 | 60.931 | 2.352 | 43.553 | 5/2 | 3.29 |

**Table 9: Speedup Gain Assessment for Group 3: CA (Size, 4, 10, V: 2..10)**

| V | Size | MIPOG | MC_MIPOG | | Grid_MIPOG | | |
|---|---|---|---|---|---|---|---|
| | | Time | Time | S_up | Time | W/M | S_up |
| 2 | 43 | 0.148 | 0.141 | 1.05 | 0.164 | 2/1 | 0.9 |
| 3 | 217 | 0.408 | 0.383 | 1.065 | 0.4 | 3/1 | 1.02 |
| 4 | 637 | 1.39 | 0.983 | 1.414 | 1.061 | 4/1 | 1.31 |
| 5 | 1643 | 6.9 | 3.818 | 1.807 | 3.575 | 5/2 | 1.93 |
| 6 | 3657 | 68.32 | 31.484 | 2.17 | 26.076 | 6/2 | 2.62 |
| 7 | 5927 | 70.495 | 31.755 | 2.22 | 17.846 | 7/2 | 3.95 |
| 8 | 11355 | 4767.778 | 1538.719 | 3.099 | 703.212 | 8/2 | 6.78 |
| 9 | 18036 | 5203.01 | 1605.372 | 3.241 | 657.693 | 9/3 | 7.911 |
| 10 | 27306 | 56786.346 | 16220.036 | 3.501 | 6750.635 | 10/3 | 8.412 |

generation of all tuples and search the tuple space to generate the required test suite until all tuples have been covered. In the case where the number of tuples to be considered is significantly large, adopting computational approaches can be expensive especially in terms of the space required to store the tuples and the time required to explicit enumeration. Unlike algebraic approaches, the computational approaches can be applied to arbitrary system configurations. Furthermore, computational approaches are more adaptable for constraint handling [22, 23] and test prioritization [24].

In line with the aim of this research, this paper is focusing on a general strategy for t-way test generation. Thus, what follows is a survey on existing tools that supports both pairwise and higher order t (i.e. t≥2).

Hartman et al. developed the IBM's Intelligent Test Case Handler (ITCH) as an Eclipse Java plug-in tool [25]. ITCH uses a sophisticated combinatorial algorithm to construct the test suites for t-way testing. Due to its exhaustive search algorithm, ITCH execution typically takes a long time. ITCH supports t-way test generation for 2≤t≤4.

Jenkins developed a deterministic t-way generation strategy, called Jenny [26]. Jenny adopts a greedy algorithm to produce a test suite in one-test-at-a time fashion. In Jenny, each feature has its own list of tuples. It starts out with 1-tuple (just the feature itself). When there are no tuple left to cover, Jenny goes to 2-tuples and so on. Hence, during generation instances, it is possible to have one feature still covering 2-tuples while another feature is already working on 3-tuples. This process goes on until all tuples are covered. Jenny has been implemented as an MSDOS tool using C programming language. Jenny supports t-way test generation for 2≤t≤6.

Complementary to the aforementioned work, significant efforts also involve extending existing pairwise strategies (e.g. in the case of AETG and IPO) to support *t*-way testing. AETG builds a test set "one-test-at-a-time" until all tuples are covered [27, 28]. In contrast, IPO covers "one-parameter-at-a-time" (i.e. through horizontal and vertical extension mechanisms), achieving a lower order of complexity than that of AETG [14]. AETG is a commercialized tool that supports t-way test generation for 2≤t≤6.

Arshem developed a freeware Java-based t-way testing tool called Test Vector Generator (TVG) based on extension of AETG to support t-way testing [29]. Similar efforts are also undertaken by Bryce et al. to enhance AETG for t-way testing [11, 30]. TVG supports t-way test generation for 2≤t≤6.

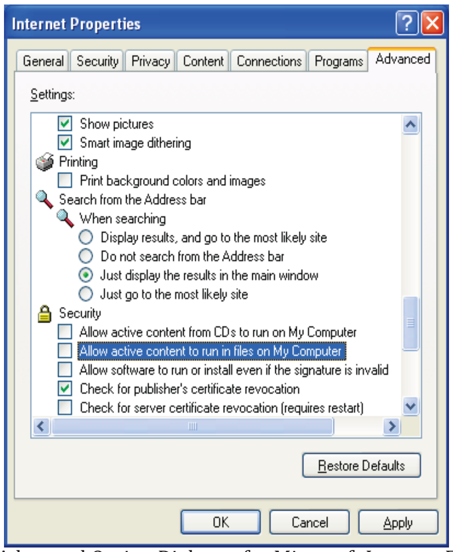
Williams implemented a Java-based t-way test tool called TConfig (Test Configuration) based on IPO [31, 32]. Later, IPO is generalized into IPOG for supporting t-way testing [9]. A number of variants have also been developed to improve the IPOG's performance. These variants including: IPOG-D [21], and IPOG-F [33]. Currently, IPOG, IPOG-D, IPOG-F are integrated into a Java-based tool called ACTS (Advanced Combinatorial Testing Suite) [34]. ACTS supports t-way test generation for 2≤t≤6.

More recently, Younis and Zamli proposed another variant to IPOG called Modified IPOG (MIPOG) [12]. Unlike earlier works, MIPOG adopts variants extension algorithms during test suite generation. The net effect of the variant extension algorithms in MIPOG is twofold. First, MIPOG can always get less test cases which would be at least the same size or even smaller than that of IPOG. Secondly, there are no dependencies between subsequently generated test values, thus, permitting the possibility of parallelization. MIPOG supports t-way test generation for 2≤t≤7 and reported to support even higher t for small system of configuration (up to t=12) [12].

Seroussi and Bshouty suggest that the problem of finding the minimum size of test suite for an arbitrary set of tuples is at least as hard as this problem [35] (i.e. NP-hard problem).Therefore, it is often unlikely that an efficient strategy exists that can always generate optimal test set (i.e. each t-way interaction is covered by minimum number of test cases). As a benchmarking exercise, Colbourn web site has collected the current best known upper bounds ( termed CAN (t, p, v) where 2 ≤ t ≤ 6) [36], regardless of the strategies used. Referring to Colbourn's web site, it should be noted that all the upper bounds are found in one-by-one basis, that is, no single strategy could be applied and yield the best test size in every configuration (i.e. due to NP-complete and NP-hard problems). As such, even a small contribution over existing tools is a daunting task to accomplish.

## MIPOG and MC_MIPOG Overview

The MIPOG strategy constructs a t-way test set configuration for the first t parameters. Then, it extends the test set to construct a t-way test set for the t+1 parameters, after which it continues to extend the test set until all t-way test set has been constructed for all the parameters of the system. For a system



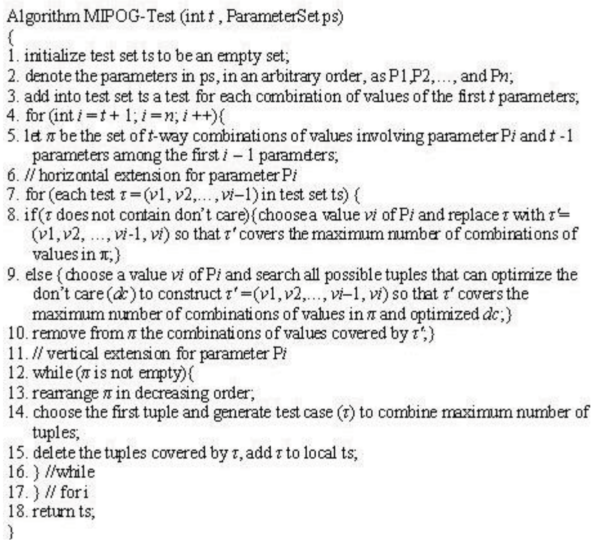
Internet Properties

General | Security | Privacy | Content | Connections | Programs | Advanced

Settings:

- [✓] Show pictures
- [✓] Smart image dithering
- Printing
  - [ ] Print background colors and images
- Search from the Address bar
  - When searching
    - ○ Display results, and go to the most likely site
    - ○ Do not search from the Address bar
    - ● Just display the results in the main window
    - ○ Just go to the most likely site
- Security
  - [ ] Allow active content from CDs to run on My Computer
  - [ ] Allow active content to run in files on My Computer
  - [ ] Allow software to run or install even if the signature is invalid
  - [✓] Check for publisher's certificate revocation
  - [ ] Check for server certificate revocation (requires restart)

**Figure 1:** Advanced Option Dialogue for Microsoft Internet Explorer



```
Algorithm MIPOG-Test (int t , ParameterSet ps)
{
1. initialize test set ts to be an empty set;
2. denote the parameters in ps, in an arbitrary order, as P1,P2,…, and Pn;
3. add into test set ts a test for each combination of values of the first t parameters;
4. for (int i = t + 1; i = n; i ++){
5. let π be the set of t-way combinations of values involving parameter Pi and t -1
     parameters among the first i – 1 parameters;
6. // horizontal extension for parameter Pi
7. for (each test τ = (v1, v2,…, vi–1) in test set ts) {
8. if(τ does not contain don't care){choose a value vi of Pi and replace τ with τ'=
     (v1, v2, …, vi-1, vi) so that τ' covers the maximum number of combinations of
     values in π;}
9. else { choose a value vi of Pi and search all possible tuples that can optimize the
     don't care (dc) to construct τ' =(v1, v2,…, vi–1, vi) so that τ' covers the
     maximum number of combinations of values in π and optimized dc;}
10. remove from π the combinations of values covered by τ';}
11. // vertical extension for parameter Pi
12. while (π is not empty){
13. rearrange π in decreasing order;
14. choose the first tuple and generate test case (τ) to combine maximum number of
     tuples;
15. delete the tuples covered by τ, add τ to local ts;
16. } //while
17. } // for i
18. return ts;
}
```

**Figure 2:** MIPOG strategy

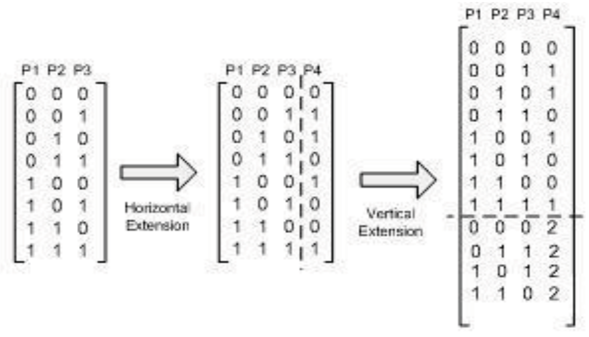
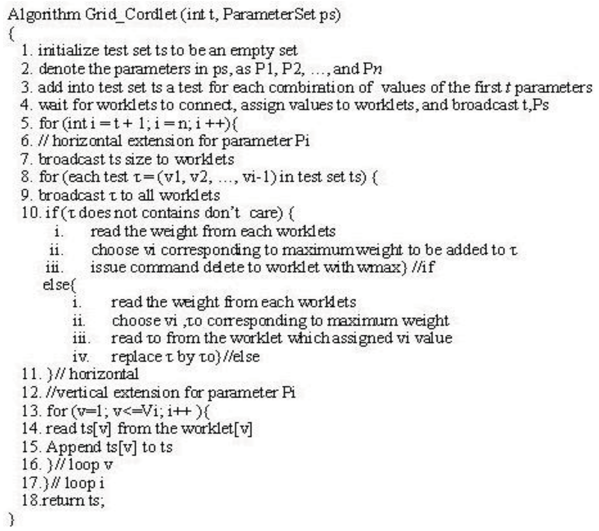


**Figure 3:** Generation of Test Set Using MIPOG



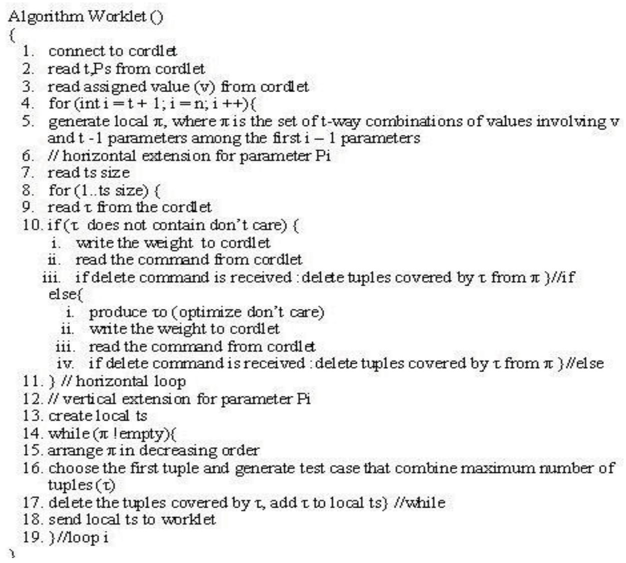
```
Algorithm Grid_Cordlet (int t, ParameterSet ps)
{
  1. initialize test set ts to be an empty set
  2. denote the parameters in ps, as P1, P2, …, and Pn
  3. add into test set ts a test for each combination of values of the first t parameters
  4. wait for worklets to connect, assign values to worklets, and broadcast t,Ps
  5. for (int i = t + 1; i = n; i ++){
  6. // horizontal extension for parameter Pi
  7. broadcast ts size to worklets
  8. for (each test τ = (v1, v2, …, vi-1) in test set ts) {
  9. broadcast τ to all worklets
  10. if (τ does not contains don't care) {
       i.    read the weight from each worklets
       ii.   choose vi corresponding to maximum weight to be added to τ
       iii.  issue command delete to worklet with wmax) //if
     else{
         i.    read the weight from each worklets
         ii.   choose vi ,τo corresponding to maximum weight
         iii.  read τo from the worklet which assigned vi value
         iv.   replace τ by τo} //else
  11. }// horizontal
  12. //vertical extension for parameter Pi
  13. for (v=1; v<=Vi; i++ ){
  14. read ts[v] from the worklet[v]
  15. Append ts[v] to ts
  16. }// loop v
  17.}// loop i
  18.return ts;
}
```

**Figure 4:** Algorithm for Grid Cordlet

with p parameters, v values, and t degree interaction (t≤p), the MIPOG strategy is illustrated in Fig. 2.

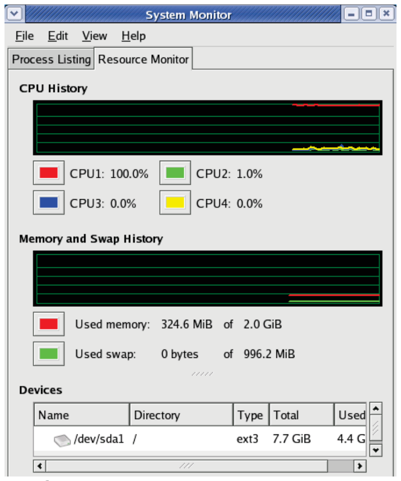
To illustrate how the MIPOG works and how it can be parallelized, a system with 4 parameters (three 2-valued and one 3-valued parameter) is considered. Fig. 3 shows the process of generating a 3-way test set. The test set (ts) starts with the eight combinations of P1P2P3 (line 3 in Fig. 2). Next (i=4), the set of *t*-way combinations ($\pi$) of values involving parameter P4 and two parameters among the first three parameters (P1P2P3) is constructed as tabulated in Table 4 with parallelism in mind. Referring to Table 4, the first column ("Parameter-Control") presents the parameter control which is used to select two parameters from P1P2P3 to be interacted with P4. The second column ("Tuple-set") presents the parameter-values for the selected two parameters among the first three parameters. Finally, the last column ("P4-value") presents the shared value for each tuple-set.

Now, return back to out illustrative example in Fig. 3, in horizontal extension, the first test case in ts, this test case starts with P1P2P3=000. According to Table 4, there are three possible values for P4 (i.e., 0,1 and 2). The MIPOG determines the weight (i.e., the number of covered tuples). In the case of P4=0 (i.e., P1P2P3P4=0000); P4=0, there are three parameter control in the first column (110, 101, and 011) that presents the parameters (P1P2, P1P3, and P2P3) respectively, and thus, the test case is decomposed into the following tuples {00, 00, and 00} respectively. Each of the decomposed tuples is contained in tuple-set column. Therefore, the weight of this candidate test case is 3. Similarly, the next possible values of P4 are 1 and 2. In the case of P4=1 (i.e., P1P2P3P4=0001); P4=1, there are three parameter control in the first column (110, 101, and 011) that presents the parameters (P1P2, P1P3, and P2P3) respectively , and thus, the test case is decomposed into the following tuples {00, 01, and 01} respectively. Each of the decomposed tuples is contained in tuple-set column. Therefore, the weight of this candidate test case is also 3. Similarly, the next possible value of P4 is 2. (i.e., P1P2P3P4=0002); P4=1, there are three parameter control in the first column (110, 101, and 011) that presents the parameters (P1P2, P1P3, and P2P3) respectively, and thus, the test case is decomposed into the following tuples {00, 02, and 02} respectively. Each of the decomposed tuples is contained in tuple-set column. Therefore, the weight of this candidate test case is also 3. Thus, MIPOG engine selects parameter value witch constructs a test case that has the first maximum weight (i.e., 0000). After adding the selected parameter value, the MIPOG engine deletes the covered tuples (i.e., {00, 00, and 00} for P4=0, and



```
Algorithm Worklet ()
{
  1.  connect to cordlet
  2.  read t,Ps from cordlet
  3.  read assigned value (v) from cordlet
  4.  for (int i = t + 1; i = n; i ++){
  5.  generate local π, where π is the set of t-way combinations of values involving v
      and t -1 parameters among the first i – 1 parameters
  6.  // horizontal extension for parameter Pi
  7.  read ts size
  8.  for (1..ts size) {
  9.  read τ from the cordlet
  10. if(τ does not contain don't care) {
        i.   write the weight to cordlet
        ii.  read the command from cordlet
        iii. if delete command is received : delete tuples covered by τ from π }//if
      else{
         i.   produce το (optimize don't care)
         ii.  write the weight to cordlet
         iii. read the command from cordlet
         iv.  if delete command is received : delete tuples covered by τ from π }//else
  11. } // horizontal loop
  12. // vertical extension for parameter Pi
  13. create local ts
  14. while (π !empty){
  15. arrange π in decreasing order
  16. choose the first tuple and generate test case that combine maximum number of
      tuples (τ)
  17. delete the tuples covered by τ, add τ to local ts} //while
  18. send local ts to worklet
  19. }//loop i
}
```

**Figure 5:** Algorithm for Worklet

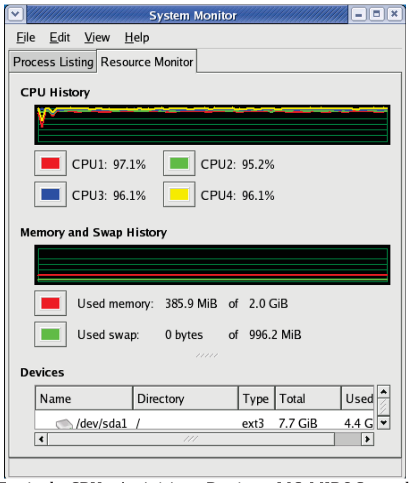


CPU1: 100.0% CPU2: 1.0%
CPU3: 0.0% CPU4: 0.0%

Used memory: 324.6 MiB of 2.0 GiB
Used swap: 0 bytes of 996.2 MiB

| Name | Directory | Type | Total | Used |
|---|---|---|---|---|
| /dev/sda1 | / | ext3 | 7.7 GiB | 4.4 G |

Typical CPU Activities During MIPOG Execution



CPU1: 97.1% CPU2: 95.2%
CPU3: 96.1% CPU4: 96.1%

Used memory: 385.9 MiB of 2.0 GiB
Used swap: 0 bytes of 996.2 MiB

| Name | Directory | Type | Total | Used |
|---|---|---|---|---|
| /dev/sda1 | / | ext3 | 7.7 GiB | 4.4 G |

Typical CPU Activities During MC_MIPOG and G_MIPOG Execution

**Figure 6:** The CPU Activities

parameter-control (110, 101, and 011) respectively). The remaining $\pi$ set tuples is given in Table 5.

For the second test case in ts, this test case starts with P1P2P3=001. According to Table 5, there are three possible values for P4 (i.e., 0,1 and 2). Again, the corresponding weights of these values are (2, 3, and 3) respectively. Thus, the constructed test case is 0011, the MIPOG engine deletes the covered tuples (i.e., {00, 01, and 01} for P4=1, and parameter-control (110, 101, and 011) respectively). The process of horizontal extension is continued for the third till eighth test cases in ts; yields the selection the following values for P4 (1,0,1,0,0,and 1)respectively. The remaining $\pi$ set tuples is given in Table 6.

In vertical extension for P4=0 and 1, the remaining $\pi$ set tuples are empty. As such, the first and second partial tests set are empty. Now, the third partial test set (i.e., for P4=2) is constructed as follows. Referring to Table 6, the $\pi$ set tuples already arranged in descend order (according to the number of tuples for each parameter control). The maximum weight (number of parameter control is 3). The first tuple is 00, with the parameter-control 110. Here, the MIPOG search engine expands the tuple to be 00*2 (i.e., by adding * value for corresponding 0-vlaue in parameter control, and append the parameter value (2) at the end, * can be further optimized by assigning value to it). Next, parameter-control is 101. Thus, the expanded tuple (00*2) can be combined with the first tuple in tuple set (i.e., 00) yields 0002. This candidate test case has a weight=3. Therefore, it is added to test suite. Then, the MIPOG engine deletes the tuples covered by the generated test case (i.e., 0002). This process is continued until π set is empty (line 11-16 in Fig. 2).

Note that the proposed data structure for π set in the MIPOG permits the parallelization as follows. Each parameter values can be generated and laid in a separate processing element. Each processing element can handle the computation of the horizontal and vertical extension for particular value. In our illustrative example, there are three possible values for P4. Thus, the π set and its corresponding computation can be distributed to three processing elements. These distribution doses not affect the process of generation, and hence, the generated test suite is identical to sequential implementation.

Building from MIPOG, the MC_MIPOG has been proposed in [12]. Here, the processing elements are distributed in tightly-coupled (Multicore) CPUs. The evaluation, speedup gain for adopting MC_MIPOG, and comparisons with other existing tools are well elaborated in [12].

In MC_MIPOG strategy the computation and memory lay in logically shared memory and CPUs. However, the physical memory is bounded to the main memory (RAM) of the tightly-coupled system. In addition, the trend in speedup is also bounded by the number of CPUs in the system. To overcome these obstacles, we propose an alternative distribution of MIPOG aimed for loosely coupled system.

## Grid_MIPOG

Like MC_MIPOG, GRID_MIPOG strategy distributes the computational processes and memory into pieces that are distributed into loosely coupled machines. The complete implementation of GRID_MIPOG strategy is actually based on the following design criteria:

- Memory storages for the tuples and their related computation (i.e., horizontal and vertical extension) are distributed to relatively independent cells, called worklets.
- The worklets can be sited in a single machine or multiple machines (i.e. for scalability and distribution purposes).
- The selected test set is stored into a shared memory controlled by test generation server, called cordlet.
- The uses of a cordlet are two-fold: as a coordinator and a server.

For clarity, the complete algorithms for cordlet and worklet are given in Fig. 4 and Fig. 5 respectively.

For our illustrative example (discussed in Section II), The Cordlet waits for three Worklets to be connected. After the Worklets are connected, the Cordlet assigns values to the Woklet (Worklet0, Worklet1, and Worklet2). For instance the Cordlet broadcast P1P2P3P4 and t=3 to the Worklets. In horizontal extension, the Cordlet broad cast the test size (8). Then for each test case in ts, the Cordlet broadcast the test case to all Worklets. For the first test case, the Cordlet broadcasts test case 000 to all Worklets. After that, the Cordlet reads the weights from Worklet0, Worklet1, and Worklet2, these weights values are 3. Then the Cordlet add 0 (i.e., construct test case 0000). Next, the Cordlet issues delete command to Worklet0, and cancel commands to Worklet1 and Worklet2. This process is continued for the remaining test cases in horizontal extension. In vertical extension, The Cordlet receives an empty partial test set from Worklet0 and Worklet1. Finally, The Cordlet receives 4 test cases from Worklet2, and then adds these test cases to ts. Thus, Grid_MIPOG produces the same test suite as in MIPOG.

**Evaluation and Discussion**

In order to assess the speedup gain, we apply MIPOG, MC_MIPOG, and GRID_MIPOG to the same 3 experimental groups adopted from [9, 12] as follows.

- **Group 1:** The number of parameters (P) and the values (V) are constant at10 and 5, but the coverage strength (t) varies from 2 to 7.
- **Group 2:** The coverage strength (t) and the values (V) are constant to 4 and 5, but the number of parameter (P) is varied from 5 to 15.
- **Group 3:** The number of parameter (P) and the coverage strength (t) are constant from t to 10 and 4 respectively, but the values (V) are varied from 2 to 10.

Here, speedup is defined as ratio of the time taken by the sequential MIPOG algorithm to the time taken by the parallel algorithm. In our evaluation, we adopt two system configuration and running environments. The first environment is a standalone system for MIPOG and MC_MIPOG execution consisting of Linux Centos OS with 2.4 GHz Core 2 Quad CPU [37], 2 GB RAM, with JDK 1.5 installed. The execution within this standalone environment is necessary as the base execution time result for MIPOG and MC_MIPOG to be compared with that of GRID_MIPOG. As discussed earlier, unlike MIPOG which has sequential algorithm, the MC_MIPOG algorithm adopts tightly coupled approach (i.e. concurrent algorithm) over the quad CPUs. The second environment is intended for Grid_MIPOG execution. The environment consists of a loosely coupled GRID environment, that is, four machines act as a cluster (one for Cordlet as master, and others are slaves for Worklets). Here, each machine has identical specification for the standalone machine. As such, the experimental comparison amongst MIPOG, MC_MIPOG, and Grid _MIPOG is considered fair. Apart from executing within the same system configuration and running environment, the data structure and implementation language of the strategies are identical. The only differences are in terms of whether or not the employed algorithms are sequential, concurrent, or distributed. Tables 7 through 9 highlight the comparative results in terms of execution time and speedup gain amongst MIPOG, MC_MIPOG, and Grid_MIPOG. The columns “S_up” and “W/M” refer to speedup and workers per machines respectively. Note that the execution time is in seconds, and MIPOG, MC_MIPOG, and Grid_MIPOG produce the same test set in all cases.

The CPU activities during MIPOG, MC_MIPOG, and G_MIPOG must also exhibit the efficient use of the multi-cores CPUs. As depicted in Fig. 6, the typical CPU activities for both MIPOG and MC_MIPOG during all the experimental groups. Here, due to its sequential nature MIPOG strategy utilizes only 25 % from the total number of CPU (or 1/N, where N=Number of CPUs) during execution, whilst the MC_MIPOG and G_MIPOG utilizes almost all the processing power of each cores (i.e. all core utilization is > 90%).

Overall, MC_MIPOG appears to perform slightly better than Grid_MIPOG for small systems due to heavy networking and inter-process communications overhead as compared to simple thread synchronization in a single system. However, Grid_MIPOG performs better than MC_MIPOG for higher degree interactions (typically t>3), and higher number of parameters (typically >6) and high number of values (typically >4). Extrapolating and performing curve fitting of the results from Tables 7 to 9, the trends in both MC_MIPG and Grid_MIPOG are to achieve maximum theoretical speedup. The maximum theoretical speedup is equal to the number of variables in the case of MC_MIPOG and Grid_MIPOG (assume the number of CPUs available>=number of variables); otherwise, the maximum theoretical speedup equals to the number of CPUs. The maximum theoretical speedup is the trend of parallelism as far as the number of variables (V) is concerned. Also, the speedup grows linearly towards the maximum theoretical speed up, as far as the number of parameters (P) is concerned. Finally, the speedup grows logarithmically towards the maximum theoretical speedup, as far as the strength of coverage (t) is concerned. Specifically, Grid_MIPOG is scale better than MC_MIPOG as far as the number of parameter values is concerned. For instance, the last column in Table 9, the maximum theoretical speedup is 4 and 10 for MC_MIPOG and Grid_MIPOG respectively, whilst the practical speedup is 3.501 and 8.412 respectively. This increasing in speedup as far as the number of parameter values is concerned can lead to produce less execution time for large values as compared with small values. For instance, consider the fifth and sixth rows in Table 9. In the case of the G_MIPOG strategy, the speedup for V=6, and 7 are 2.62, and 3.95 respectively. This is the reason why the execution time for V=7 is less than the execution time for V=6 (i.e. 17.846 versus 26.076 respectively, see Table 9).

**CONCLUSION**

In this paper, we investigated and evaluated a parallel strategy called Grid_MIPOG for t-way test data generation on loosely coupled architecture. Our results indicate that the distributed implementation scales well against concurrent implementation (MC_MIPOG) and sequential predecessor (MIPOG). As computer manufactures make multi-core CPUs pervasively available within reasonable costs, harnessing this technology is no longer a luxury but a viable and useful option.

The current implementation of the MIPOG and its family takes on parameter at a time. In fact, as the computing powers duplicated every 18 months according to Moore's law, it is evidence that taking more than one parameter at a time (e.g. 2 parameters) is also feasible. This research avenue is considered as a part of our future work that definitely can lead to more speedup and perhaps more optimal test size.

Finally, much research work has been done in this field in the last decade; the adoption of these strategies for studying and testing real life systems (e.g. software, hardware, medical, genes) has not been widespread. For these reasons, more research into the algorithms, techniques, and methodologies are required to facilitate its adoption in the main stream of software engineering.

**REFERENCES AND NOTES**

1. R.C. Bryce, Y. Lei, D.R. Kuhn, and R. Kacker, “Combinatorial Testing,” Handbook of Research on Software Engineering and Productivity Technologies: Implications of Globalization, M. Ramachandran, and R.A.d. Carvalho eds., IGI Global, pp. 196-208, 2010.
2. R. Kuhn, R. Kacker, Y. Lei, and J. Hunter, “Combinatorial Software Testing,” IEEE Transaction on Computer, vol. 42, no. 8, pp. 94-96, 2009.
3. D.M. Cohen, S.R. Dalal, A. Kajla, and G.C. Patton, “The Automatic Efficient Test Generator (AETG) System,” Proceedings of the 5th International Symposium on Software Reliability Engineering, IEEE Computer Society, pp. 303 - 309, 1994.
4. M.B. Cohen, C.J. Colbourn, P.B. Gibbons, and W.B. Mugridge, “Constructing Test Suites for Interaction Testing,” Proceedings of the 25th IEEE International Conference on Software Engineering, IEEE Computer Society, pp. 38-48, 2003.
5. D.R. Kuhn, and V. Okun, “Pseudo Exhaustive Testing For Software,” Proceedings of the 30th NASA/IEEE Software Engineering Workshop, IEEE Computer Society, pp. 153-158, 2006.
6. R. Kuhn, Y. Lei, and R. Kacker, “Practical Combinatorial Testing: Beyond Pairwise,” IEEE IT Professional, vol. 10, no. 3, pp. 19-23, 2008.
7. M. Ellims, D. Ince, and M. Petre, “The Effectiveness of T-Way Test Data Generation,” Springer LNCS 5219, vol. SAFECOMP 2008, pp. 16–29, 2008.
8. M.I. Younis, and K.Z. Zamli, “Assessing Combinatorial Interaction Strategy for Reverse Engineering of Combinational Circuits,” IEEE Symposium on Industrial Electronics and Applications (ISIEA 2009) , pp. 473-478, 2009.
9. Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence, “IPOG: A General Strategy for T-Way Software Testing,” Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS2007), IEEE Computer Society, pp. 549-556, 2007.
10. C.J. Colbourn, G. Keri, P.P.R. Soriano, and J.-C. Schlage-Puchta, “Covering and radius-covering

arrays: Constructions and classification," Discrete Applied Mathematics, vol. 158, no. 11, pp. 1158-1180, 2010.

11. 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.
12. M.I. Younis, and K.Z. Zamli, "MC-MIPOG: A Parallel t-Way Test Generation Strategy for Multicore Systems," ETRI Journal, vol. 32, no. 1, pp. 73-82, 2010.
13. M.I. Younis, and K.Z. Zamli, "MIPOG - An Efficient t-Way Minimization Strategy for Combinatorial Testing," IJCTE, vol. 3, no. 3, pp. 388-397, 2011.
14. 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.
15. T. Shiba, T. Tsuchiya, and T. Kikuno, "Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing," Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04), IEEE Computer Society, pp. 72-77, 2004.
16. A. Hartman, and L. Raskin, "Problems and Algorithms for Covering Arrays," Discrete Mathematics, vol. 284, no. 1-3, pp. 149-156, 2004.
17. M. Grindal, J. Offutt, and S. Andler, "Combination Testing Strategies: a Survey," Software Testing, Verification, and Reliability, vol. 15, no. 3, pp. 167-199, 2005.
18. C.J. Colbourn, S.S. Martirosyan, G.L. Mullen, D. Shasha, G.B. Sherwood, and J.L. Yucas, "Products of Mixed Covering Arrays of Strength Two," Journal of Combinatorial Designs, vol. 14, no. 2, pp. 124–138, 2006.
19. G.B. Sherwood, "Pairwise Testing Comes of Age," Testcover Inc., 2008.
20. C. Yilmaz, M.B. Cohen, and A. Porter, "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces," IEEE Transactions on Software Engineering, vol. 31, no. 1, pp. 20–34, 2006.
21. 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.
22. M. Grindal, J. Offutt, and J. Mellin, "Conflict Management when Using Combination Strategies for Software Testing," Proceedings of the 18th Australian Software Engineering Conference (ASWEC 2007), pp. 1-10, 2007.
23. M.B. Cohen, M.B. Dwyer, and J. Shi, "Interaction Testing of Highly-Configurable Systems in the Presence of Constraints," Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007), ACM Press, pp. 129–139, 2007.
24. R.C. Bryce, and C.J. Colbourn, "Prioritized Interaction Testing for Pairwise Coverage with Seeding and Avoids," Information and Software Technology Journal, vol. 48, no. 10, pp. 960-970, 2006.
25. A. Hartman, T. Klinger, and L. Raskin, "WHITCH: IBM Intelligent Test Configuration Handler," IBM Haifa and Watson Research Laboratories, pp. 1-47, 2005.
26. B. Jenkins, "Jenny Test Tool," available from http://www.burtleburtle.net./bob/math/jenny.html, last accessed on April, 2011.
27. D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," IEEE Software, vol. 13, no. 5, pp. 83-88, 1996.
28. D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton, "The AETG System: An Approach to Testing based on Combinatorial Design," IEEE Transactions on Software Engineering, vol. 23, no. 7, pp. 437–443, 1997.
29. J. Arshem, "Test Vector Generator Tool (TVG)," available from http://sourceforge.net/projects/tvg, last accessed on October, 2011.
30. R.C. Bryce, C.J. Colbourn, and M.B. Cohen, "A Framework of Greedy Methods for Constructing Interaction Test Suites," Proceedings of the 27th IEEE International Conference on Software Engineering, pp. 146-155, 2005.
31. A.W. Williams, "Software Component Interaction Testing: Coverage Measurment and Generation of the Configurations (PhD Thesis)," School of Information Technology and Engineering, University of Ottawa, Ottawa, Canada, 2002.
32. A.W. Williams, J.H. Ho, and A. Lareau, "TConfig Test Tool Version 2.1," 2003, available from http://www.site.uottawa.ca/~awilliam, last access on December 2011.
33. 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.
34. NIST, "Automated Combinatorial Testing for Software," available from http://csrc.nist.gov/groups /SNS/acts, last accessed on September, 2011.
35. G. Seroussi, and N.H. Bshouty, "Vector Sets for Exhaustive Testing of Logic Circuits," IEEE Transactions on Information Theory, vol. 34, pp. 513-522, 1988.
36. C.J. Colbourn, "Covering Array Tables," available from http://www.public.asu.edu /ccolbou/src/tabby, last access on January 2012.
37. Intel, "Intel Core 2 Quad Processors," available at http://www.intel.com/products /processor/core2quad / index.htm, last accessed on December, 2011.