ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/224385741

A strategy for Grid based t-way test data generation

Conference Paper · November 2008

DOI: 10.1109/ICDFMA.2008.4784416 · Source: IEEE Xplore

CITATIONS
19

READS
70

3 authors:

Mohammed I. Younis
University of Baghdad
48 PUBLICATIONS 351 CITATIONS

SEE PROFILE

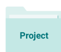
Kamal Z Zamli
Universiti Malaysia Pahang
167 PUBLICATIONS 1,250 CITATIONS

SEE PROFILE

Nor Ashidi Mat Isa
Universiti Sains Malaysia
225 PUBLICATIONS 2,398 CITATIONS

SEE PROFILE

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



Enhancment of Face Recognition by adopting pre-processing Techniques View project

Improving some existing Optimizations algorithms by the hybridizations approach to solve the combinatorial optimization problems. View project

All content following this page was uploaded by Kamal Z Zamli on 21 May 2014.

The user has requested enhancement of the downloaded file.

A Strategy for Grid Based T-Way Test Data Generation

Mohammed I. Younis, Kamal Z. Zamli, and Nor Ashidi Mat Isa
School of Electrical and Electronic Engineering
Universiti Sains Malaysia Engineering Campus
14300 Nibong Tebal, Penang, Malaysia
Email: {younismi@gmail.com, eekamal@eng.usm.my, ashidi@eng.usm.my}

Abstract

Although desirable as an important activity for ensuring quality assurances and enhancing reliability, complete and exhaustive software testing is next to impossible due to resources as well as timing constraints. 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. In some system, faults may also be caused by more than two parameters.

As the number of parameter interaction coverage (i.e. the strength) increases, the number of t-way test set also increases exponentially. As such, for large system with many parameters, considering higher order t-way test set can lead toward combinatorial explosion problem (i.e. too many data set to consider). We consider this problem for t-way generation of test set using the Grid strategy. Building and complementing from earlier work in IPOG and MIPOG, we present the Grid MIPOG strategy (G_MIPOG). Experimental results demonstrate that G_MIPOG scales well against the sequential strategies IPOG and MIPOG with the increase of the computers as computational nodes.

1. Introduction

As an activity for ensuring quality assurances and improving reliability, software testing is an important part of the software engineering lifecycle. Lack of testing often leads to disastrous consequences including loss of data, fortunes and even lives. For these reasons, many inputs parameters and system conditions need to be tested against the system's specification for conformance. Although desirable, exhaustive software testing is next to impossible due to resources as well as timing constraints.

While earlier work (e.g. in [3][10]) 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, the study by The National Institute of Standards and Technology (NIST) [10] reported that 95% of the actual faults on the test software involve 4-way interaction. In fact, almost all of the faults are detected with 6-way interaction. Thus, as this example illustrates, system faults caused by variable interactions may also span more than two parameters.

Considering more than two parameter interaction is not without difficulties. To highlight the difficulties, consider the TCAS is an aircraft collision avoidance system from the Federal Aviation Administration which has been used as case study in other related works [2][9][10]. Here, TCAS module has twelve parameters: seven parameters have 2 values, two parameters have three values, one parameter has four values, and two parameters have 10 values. Running exhaustive test requires 460800 (i.e., 10x10x4x3x3x2x2x2x2x2x2x2), or 12 way testing for this system (i.e. running such test may be impossible). Alternatively, 11-way testing requires 230400. 10-way requires 201601. 9-way requires 120361. 8-way requires 56742. 7-way requires 26061. 6-way requires 10851. 5-way requires 4196. 4-way requires 1265. 3-way requires 400. Finally, 2-way requires 100 test cases.

As demonstrated above, when the number of parameter coverage increases, the number of t-way test set also increases exponentially. As such, for large system with many parameters, considering higher order t-way test set can lead toward combinatorial explosion problem. We consider this problem for t-way generation of test set using the Grid strategy. Building and complementing from earlier work in IPOG and MIPOG, we present the Grid MIPOG strategy

(G_MIPOG). Experimental results demonstrate that G_MIPOG scales well against the sequential strategies IPOG and MIPOG with the increase of the computers as computational nodes.

The remainder of the paper is organized as follows. Section 2 briefly reviews related work on t-way testing. Section 3 gives analysis the IPOG strategy, and gives the reasons for variant IPOG (MIPOG). Section 4 reports the design of Grid based Test generator (GMIPOG). Section 5 gives the results of the experiments. Section 6 provides concluding remarks and our plan for further work

2. Overview and Related Work

Lei *et al.* proposes a useful strategy, called In-Parameter-Order-General (or IPOG) [2] to support t-way test generation. In a nut shell, IPOG generalizes an existing strategy, called In-Parameter-Order (or IPO [3]), from pairwise testing to support general t-way testing. Here, as the name suggests, IPOG is based on IPO. IPO is a test generation strategy used for Pairwise (2-way) testing. For a system with two or more input parameters, the IPO strategy first generates a pairwise test set for the first two parameters. It then continues to extend the test set to generate a pairwise test set for the first three parameters and continues to do so for each additional parameter until all the parameters of the system are covered.

IPO follows two steps to extend the test when additional parameters are added:

i. Horizontal Growth, which extends each additional test by adding one value of the new parameter.
ii. Vertical Growth, which adds new tests if required after the completion of Horizontal growth.

In their work, Lei *et al.* demonstrates the effectiveness of IPOG for t-way testing and describe its implementation tool called FireEye. Additionally, Lei *et al*. have also identified the following existing tools that support t-way testing:

- Intelligent Test Case Handler (or ITCH) [4]
- Jenny [5]
- TConfig [6]
- Test Vector Generator (or TVG) [7].

Comparative results from Lei *et al.* demonstrated that IPOG performed better than all the abovementioned tools, both in terms of the sizes of the test sets and the execution times. For this reason, we have adopted the IPOG strategy as our benchmark.

As part of an effort to enhance and improve IPOG, we have implemented the modified IPOG (MIPOG) in our earlier work (described in [8]). The key feature of MIPOG is the fact that it generates more optimum test set (i.e. each t-way interaction is covered by only one test) and hence lesser combinations than that of IPOG itself. Based on MIPOG, we are to redesign and distribute the test generation process (called G_MIPOG) on the GRID.

In order to understand G_MIPOG, it is necessary to understand IPOG and MIPOG first. From Lei *et al.,* the IPOG strategy for t-way test generation is given in Figure 1.

Algorithm *IPOG (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. choose a value vi of Pi and replace τ with τ’ = (v1, v2,…, vi-1, vi) so that τ’ covers the
most number of combinations of values in π
9. remove from π the combinations of values covered by τ’
10. }
11. // vertical extension for parameter Pi
12. ***for*** *(each combination σ in set π){*
13. ***if*** *(there exists a test that already covers σ) {*
14. remove σ from π
15. } ***else*** *{*
16. change an existing test, if possible, or otherwise add a new test to cover σ and remove it from π
17. }
18. }
19.}
20. ***return*** *ts;*
}

Figure 1. IPOG Strategy

Referring to Figure 1, we note two aspects of the current IPOG strategy that can further be improved. Firstly, the generation of test set (ts) is clearly unstable, due to the possibility for changing the test case during the vertical extension (especially for test cases that includes don’t care). This raises the issue of dependency between previous generated test and new

one. Secondly, the tuples for combinations in the ith (i>t) parameter are stored in single π set. This means that for large numbers of variables or t, the memory requirement increases substantially. Such memory requirement can lead to huge heap size and potentially cause out of memory exception during runtime. In this way, the system performance can be seriously affected.

To improve the first aspect, we have considered variant algorithms for both horizontal and vertical extensions to remove dependencies (see Figure 2). Here, the generated test case is independent of each other and the size of the generated set would also be optimum. Interest readers are referred to our earlier paper in [8] for detail explanations.

***Algorithm** MIPOG (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 not contains don't care){
choose a value vi of Pi and replace τ with τ' = (v1, v2, dc,…, 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 be optimized 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 not empty){*
13. rearranges π in decreasing order.
14. Choose the first tuple and generate test case (τ) that combine maximum number of tuples
15. delete the tuples covered by τ, add τ to local ts
16. } //while
*17. **return** ts;*
}

Figure 2. MIPOG Strategy

To improve the second aspect, we have opted to enhance and modify the MIPOG strategy to run on the GRID. As discussed earlier, apart from tackling out of memory problems when dealing with high order parameter interaction, we also aim to improve execution time as well as the scalability of the strategy to support higher t. The details modification of MIPOG (or G_MIPOG) will be discussed next.

4. The Proposed Strategy (G_MIPOG)

The G_MIPOG strategy distributes the computational processes and memory into pieces. The complete implementation of G_MIPOG strategy is actually based on the following design criteria.

i. Memory needs to be distributed in order to hold Pi into relatively independent cells, called *worklet*. Here, each worklet need to have its own memory to hold the t-way combinations for a particular value.
ii. The worklets can be in a single machine or multiple machines (i.e. for scalability purposes).
iii. The selected test set is stored into a shared memory controlled by TestGeneration server, called *cordlet*.

4.1. Cordlet

As implied earlier, the cordlet roles are two folds: as coordinator and server. Briefly, the cordlet works as follows:

1. The cordlet assigns parameters values for each worklet in advance. In this way, each worklet knows in advance its role in the generation process. The cordlet waits for max (maximum number of variables) to join it to go to step2; otherwise remain in step 1.
2. The cordlet starts with an empty test set (ts), and generates all tuples for the first t-parameters.
3. In Horizontal extension
   a. For each test case τ in test set ts, the cordlet broadcasts τ to all worklets.
   b. If τ do not contain don't care, the cordlet reads the weight (i.e. number of covered tuples after adding the assigned value) from each worklets. Then, the cordlet chooses the value corresponding to maximum weight to be added to ts.
   c. If τ contains don't care, the cordlet reads the weight (number of covered tuples after adding the assigned value) from each worklets. The cordlet chooses the value corresponding to maximum weight.
   d. In both cases b and c, the cordlet issues command to the selected worklet to delete tuples from their own π set (πv). In case c, the cordlet reads the optimized test case

(τo). Then, replace τ by τo. It should be noted that the optimization process is done by the worklets themselves which may issue command to others worklets.

4. In Vertical extension.
   a. The cordlet waits for the worklets to finish their partial test set (tsvth). Then, the cordlet collects tsvths from the worklets. Finally, the cordlet adds each tsvth to ts.

For clarity, the complete algorithm for cordlet is given in Figure 3.

***Algorithm** Grid-Cordlet (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. Waits for (max values) worklets to connect and assign values to worklets. broadcasts t,Ps
*5. **for** (int i = t + 1; i ≤ n; i ++){*
6. // horizontal extension for parameter Pi
Send ts size
*7. **for** (each test τ = (v1, v2, …, vi-1) in test set ts) {*
8. broadcast τ to all worklets.
9.If (τ not contains don't care) {
reads the weight from each worklets
chooses vi corresponding to maximum weight to be added to τ, issue command delete to worklet with wmax}//if
else{ reads the weight from each worklets
chooses vi ,τo corresponding to maximum weight worklet
read τo from worklet max
replace τ by τo}//else
10. }// horizontal
11. // vertical extension for parameter Pi
*12. **for** (v=1; v<=N; i++){// N connected worklets*
13. read ts[v] from the worklet[v]
14. adds each ts[v] to ts
15. }// loop v
16.}// loop i
*17.**return** ts;*
}// algorithm

Figure 3. Algorithm for Grid Cordlet

4.2 Worklet

In this section, we now describe how the worklet works.

1. The worklet first connects to the cordlet. Then, it reads the assigned value (v) from the cordlet, and the input vector. For example, for 10 5-valued parameters the input vector will be { 5,5,5,5,5,5,5,5,5,5}.
2. The worklet generates its own partial tuples set (πv).
3. In Horizontal extension
   a. The worklet reads test case τ in test set ts from the cordlet.
   b. If τ not contains don't care. The worklet determines the weight of τ. sends the weight to the cordlet.
   c. If τ contains don't care, the worklet optimizes the don't care to have as much weight as possible and sends the weight to the cordlet.
   d. The worklet reads the command from cordlet, if it contains delete the worklet deletes tuples covered by τ (in case b) or τo (in case c) from (πv).
4. In Vertical extension
   a. The worklet arranges πv in decreasing order and choose the first tuple and generates test case with maximum weight. Step 4 is repeated untill (πv) is empty. Then, the worklet sends local test set (tsvth) to the cordlet.

The complete algorithm for the worklet is given in Figure 4.

***Algorithm** Worklet ()*
{
1. Connect to cordlet
2. read t,Ps from cordlet
3. read assigned value (v) from cordlet
*3. **for** (int i = t + 1; i ≤ n; i ++){*
5. Generates 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
read ts size
*7. **for** (1..ts size) {*
8. read τ from the cordlet
9.If (τ not contains don't care) {
writes the weight to cordlet
read the command from cordlet
if it is delete command delete tuples covered by τ from π }//if
else{ produce τo (optimize don't care)
writes the weight to cordlet
read the command from cordlet
if it is delete command delete tuples covered

by τo from π }//else
10. // vertical extension for parameter Pi
11. create local ts
*12.**while** (π!empty){*
13. arranges π in decreasing order.
14. choose the first tuple and generate test case that combine maximum number of tuples (τ)
15. delete the tuples covered by τ, add τ to local ts } //while
16. send local ts to worklet
} //algorithm

Figure 4. Algorithm for Worklet

5. Evaluation

Here, we are interested to investigate whether or not there is speedup gain from distributing MIPOG in G_MIPOG. Three experiments are applied to both MIPOG and G_MIPOG in order to gauge the speedup. Here, the speedup is defined as ratio of the time taken by single computer to the time taken by multiple computers. The experimental goals are: to investigate the speedup as the number of parameters increases; to investigate the speedup as the number of computers increases; and to investigate on whether or not there are significant increase in speedup as parameter coverage increases.

To achieve the first goal we apply MIPOG, and G_MIPOG (consists of 6 computers) to 5-valued parameters, and changing the number of parameters from 7 to 10, and fixed t=6, as given in Table 1. To achieve the second goal, we have fixed t=4, with 10 10-valued parameters. Then, we determine the speedup using 2 to 11 computers. The results are shown in Table 2. Finally, we apply G_MIPOG to Traffic Collision Avoidance System (TCAS) module. As discussed earlier, TCAS is an aircraft collision avoidance system from the Federal Aviation Administration, and has been used in other studies of software testing [2] [9] [10]. TCAS module has twelve parameters: seven parameters have 2 values, two

Table 1. Results For 7 to 10 5-Valued Parameters in 6-Way Testing

#Parameters	7	8	9	10
Test Case Size	15625	28125	40146	45168
Time(MIPOG) (Single Computer)	118.703	485.047	1637.097	4657.457
Time(G_MIPOG) (6-Computers)	55.234	183.184	626.797	1552.125
Speedup	2.149	2.648	2.838	3.203

Table 2. Results Using 1 to 11 Computers for 10 10-Valued Parameters in 4-Way Testing

# Computers	Time	Speedup	Test Case Size
1 (MIPOG)	77882.16	1	27306
2 (G_MIPOG)	43680.404	1.783	
3 (G_MIPOG)	31140.4	2.501	
4 (G_MIPOG)	24089.749	3.233	
5 (G_MIPOG)	17150.883	4.541	
6 (G_MIPOG)	15193.554	5.126	
7 (G_MIPOG)	14928.537	5.217	
8 (G_MIPOG)	14789.624	5.266	
9 (G_MIPOG)	12928.645	6.024	
10 (G_MIPOG)	9570.184	8.138	
11 (G_MIPOG)	9352.967	8.327	

Table 3. Results Using t=2 to 11 for TCAS Module

T-Way	Time(MIPOG)	Time(G_MIPOG)	Speedup	Test Case Size
2	0.156	0.292	0.534	100
3	0.609	0.547	1.113	400
4	3.234	2.578	1.2544	1265
5	36.797	29.125	1.263	4196
6	301.128	214.091	1.406	10851
7	1772.407	1086.7	1.631	26061
8	10242.09	5839.276	1.754	56742
9	36284.7	19124.78	1.897	120361
10	41481.672	21832.46	1.9	201601
11	14939.891	7821.932	1.91	230400

parameters have three values, one parameter has four values, and two parameters have 10 values. We use 2 computers, and vary t from 2 to 11 to determine the speedup, as given in Table 3.

Figures 5, 6 and 7 demonstrate the speedup obtained from Tables 2, 3, and 4 respectively. Figure 5 demonstrates the speedup increases linearly as the number of parameters increases. Figure 6 demonstrates the speedup also increases as the number of computers increases. The maximum speedup is obtained when the number of computers equal to the maximum number of variables plus one. Here, each worklet is assigned for each parameter value to a single computer as well as one other computer as cordlet.

Referring to Figure 7, we also observe that the speedup increases logarithmically as the strength of coverage increases. It should be noted that there is no speedup gained for this strategy when t=2, due to the network overhead required for coordination as well as for inter-process communications.

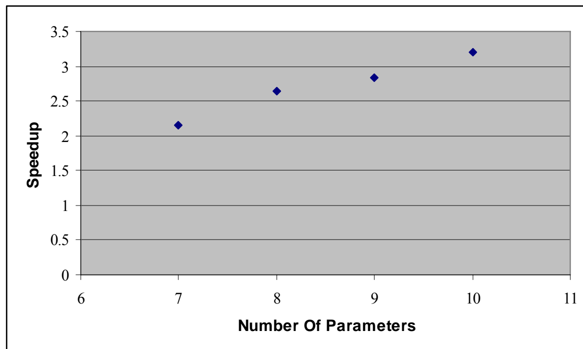


Figure 5. Speedup vs number of parameters from Table 1

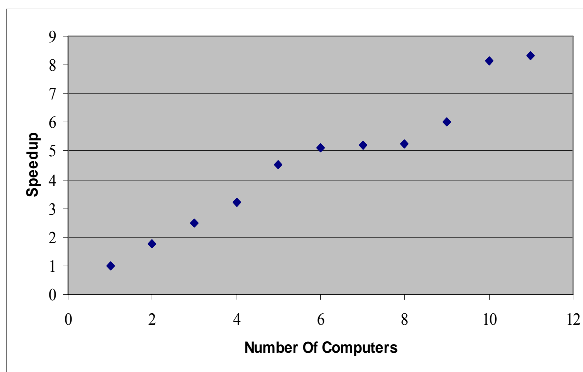


Figure 6. Speedup vs the number of computers from Table 2

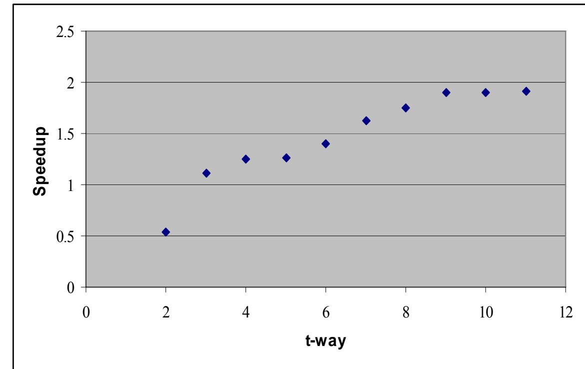


Figure 7. Speedup vs the strength of coverage (T) from Table 3

6. Conclusion

In this paper, we investigate the Grid based strategy for generating t-way test set. Our practical results are encouraging particularly in terms execution time, whilst keeping the optimized test set size. As part as our future work, we are currently study further optimization for the test size and further reduction in execution time. As part of our future work, we will be integrating G_MIPOG for Grid Based Automated Testing Environment under our USM-GRID Grants.

References

[1] P. Niemeyer and J. Peck, Exploring Java, 2nd Edition ed.: O'Reilly, September 1997.

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

[3] Y. Lei and K. C. Tai, "In-Parameter-Order: A Test Generating Strategy for Pairwise Testing," IEEE Transaction on Software Engineering vol. 28 (1), pp. 1-3, 2002.

[4] http://www.alphaworks.ibm.com/tech/whitch.

[5] www.burtleburtle.net.

[6] http://www.site.uottawa.ca/~awilliam.

[7] http://sourceforge.net/projects/tvg.

[8] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "MIPOG - Modification of the IPOG Strategy for T-Way Software Testing", submitted for publication.

[9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In Proc. of the 16th Intl. Conf. on Software Engineering, pp. 191–200, May 1994.

[10] D. R. Kuhn, V. Okun, “Pseudo-exhaustive Testing For Software,” In the Proc of the 30th NASA/IEEE Software Engineering Workshop, April 25-27, 2006.

View publication stats