Information Sciences 181 (2011) 1741–1758

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Design and implementation of a t-way test data generation strategy with automated execution tool support

Kamal Z. Zamli [a,*], Mohammad F.J. Klaib [a], Mohammed I. Younis [a], Nor Ashidi Mat Isa [a], Rusli Abdullah [b]

[a] School of Electrical and Electronic Engineering, Universiti Sains Malaysia, 14300 Nibong Tebal, Penang, Malaysia
[b] Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400 Serdang, Selangor, Malaysia

ARTICLE INFO

Article history:
Received 22 January 2009
Received in revised form 25 October 2010
Accepted 1 January 2011
Available online 12 January 2011

Keywords:
GTWay
Software testing
t-way testing
Combinatorial testing

ABSTRACT

To ensure an acceptable level of quality and reliability of a typical software product, it is desirable to test every possible combination of input data under various configurations. However, due to the combinatorial explosion problem, exhaustive testing is practically impossible. Resource constraints, cost factors, and strict time-to-market deadlines are some of the main factors that inhibit such a consideration. Earlier research has suggested that a sampling strategy (i.e., one that is based on a t-way parameter interaction) can be effective. As a result, many helpful t-way sampling strategies have been developed and can be found in the literature.

Several advances have been achieved in the last 15 years, which have, in particular, served to facilitate the test planning process by systematically minimizing the test size required (based on certain t-way parameter interactions). Despite this significant progress, the integration and automation of strategies (from planning process to execution) are still lacking. Additionally, strategizing to sample (and construct) a minimum test set from the exhaustive test space is an NP-complete problem; that is, it is often unlikely that an efficient strategy exists that could regularly generate an optimal test set. Motivated by these challenges, this paper discusses the design, implementation, and validation of an efficient strategy for t-way testing, the GTWay strategy. The main contribution of GTWay is the integration of t-way test data generation with automated (concurrent) execution as part of its tool implementation. Unlike most previous methods, GTWay addresses the generation of test data for a high coverage strength ($t > 6$).

1. Introduction

To ensure quality assurance and improve reliability, software testing [8–10] is an important phase in any software engineering life cycle. Lack of testing often leads to disastrous consequences, including the loss of data, fortunes, and even lives. For these reasons, many input parameters and system conditions need to be tested against the system's specification for conformance. Although desirable, exhaustive testing [22] is next to impossible due to resource and time constraints.

Earlier studies [12,19] suggested that pairwise testing (i.e., based on the two-way interaction of variables) is effective in detecting most faults in a typical software system. While this conclusion may apply to some systems, it cannot be generalized to all of the faults found in a software system, especially when there are significant interactions between the

* Corresponding author. Tel.: +60 4 5996003; fax: +60 4 5941023.
E-mail addresses: eekamal@eng.usm.my (K.Z. Zamli), momklaib@yahoo.com (M.F.J. Klaib), younismi@gmail.com (M.I. Younis), ashidi@eng.usm.my (N.A.M. Isa), rusli@fsktm.upm.edu.my (R. Abdullah).

0020-0255/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2011.01.002

variables. Recently, empirical evidence has suggested the need for high levels of interaction (t > 2). For instance, Dunietz et al. [23] demonstrated that, while two-way interaction provides good block coverage, the path coverage is rather poor. Based on empirical evidence in four application domains (involving medical devices, a web browser, an HTTP server, and a NASA-distributed database application [32,33]), Kuhn et al. concluded that all faults in any typical software system can be detected at t = 6. Using a different application domain, Younis and Zamli demonstrated that only after t = 7 can the behavior of the combinatorial circuits of interest be predicted for reverse engineering applications [59].

Given the potentially diverse current (and future) applications of t-way strategies, there is a clear need to aim for a high interaction strength (i.e., t > 6) to allow for the possibility of a new intertwined dependency between the involved parameters. As indicated by the number of newly developed strategies surveyed in Section 2, a great deal of progress has already been made. However, despite this progress, the integration and automation of the strategies (from the generation to execution) appear to be lacking. Currently, sampled t-way test data need to be manually extracted and converted into an acceptable format before they can be executed (e.g., by a human tester, by a code driver or by a third party execution tool). This lack of integration and automation between test generation and execution can potentially burden test engineers, especially if the application module to be tested is significantly large.

Apart from the integration and automation issues, constructing the minimum test set for t-way testing is also an NP-complete problem [45,49]; that is, it is unlikely that an efficient algorithm exists that could regularly generate an optimal test set [45]. Motivated by these challenges, this paper discusses the design, implementation, and validation of an efficient strategy, called GTWay. The main contribution of GTWay is the integration of t-way test data generation with automated (concurrent) execution as part of its tool implementation. Furthermore, unlike most previous studies, GTWay addresses the generation of test data for a high strength of coverage (i.e., t > 6).

The structure of this paper is as follows: Section 2 outlines the related work. Section 3 describes the GTWay strategy. Section 4 discusses our evaluation of the GTWay strategy. Finally, Section 5 gives our conclusion and highlights some future work.

2. Related work

There has been a significant effort to develop new t-way strategies. In general, these strategies adopt either algebraic or computational approaches [14,35].

As the name suggests, algebraic approaches are often based on the extensions of mathematical functions to construct orthogonal arrays (OA) [7,39]. Although useful, most OA-based approaches require the parameter values to be uniform, hence, restricting their application. While mutual orthogonal arrays (MOA) allow non-uniform values, not all OA and MOA solutions exist [46]. In tackling this issue, some variations in OA- and MOA-based approaches [48,53] have been proposed as flexible alternatives that support test generation even in the absence of an OA solution. Nonetheless, most OA- and MOA-based approaches merely support pairwise strategies (or two-way interactions).

Frequently, strategies that are based on algebraic approaches are extremely fast [35] because the computations are typically lightweight. Nonetheless, algebraic approaches often impose restrictions on the system configurations to which they can be applied [61] (with some fixed generalizations/assumptions). These restrictions significantly limit the applicability of algebraic approaches for software testing.

Unlike algebraic approaches, computational approaches, sometimes referred to as t-way strategies, often rely on the generation of all interaction possibilities. There is a significant searching effort required in the combinatorial space to generate the required test suite until all interactions are covered. In cases where the number of interactions to be considered is large, adopting computational approaches can be expensive, in terms of both the storage space needed for interactions and the time required to make an explicit enumeration. On a positive note, computational approaches can be applied to arbitrary system configurations. Furthermore, computational approaches are more adaptable for constraint handling [18,25] and test prioritization [4].

A number of useful strategies have been developed for t-way testing over the last decade. A significant number of studies have focused on pairwise (t = 2) strategies, such as Automatic Efficient Test Generator (AETG) [12,13], Orthogonal Array Test System (OATS) [39], Intersection Residual Pair Set Strategy (IRPS) [60], AllPairs [2], In Parameter Order (IPO) [37], Test Case Generator (TCG) [50], Orthogonal Array Test Set Generator (OATSGen) [30], ReduceArray2 [21], Deterministic Density Algorithm (DDA) [20], CTE-XL [34], and SmartTest [47]. As interaction is limited to t = 2, pairwise strategies yield the minimum test set, unlike higher orders of t. Although useful in some classes of systems, pairwise testing is known to be ineffective for systems with highly interacting variables [31,36]. For this reason, we shall focus our general strategy on t-way test generation. Thus, what follows is our survey on the existing strategies that support both pairwise interactions and higher orders of t ($t \geqslant 2$).

Hartman et al. developed IBM's Intelligent Test Case Handler (WHITCH) as a Java plug-in integrated into Eclipse tool [26]. WHITCH uses sophisticated combinatorial algorithms (based on exhaustive searches) to construct test suites for t-way testing. Although useful as part of IBM's automated test plan generation, WHITCH results are not optimized with regard to the number of generated test cases. Furthermore, due to its exhaustive search algorithm, WHITCH typically takes a long time to execute.

Jenkins developed a deterministic t-way generation strategy called "Jenny" [28]. Jenny adopts a greedy algorithm to produce a test suite in a one-test-at-a-time fashion. In Jenny, each feature has its own list of t-way interactions. Jenny begins with one-way interactions (just the feature itself). When there are no further one-way interactions left to cover, Jenny shifts to two-way interactions (this feature with another feature), etc. Thus, during generation, one feature continues to cover a two-way interaction while another feature begins working on three-way interactions. This process continues until all interactions have been covered.

Complementary to the aforementioned works, significant efforts have been made to extend the existing pairwise strategies (e.g., AETG and IPO) to support t-way testing. AETG starts with the generation of all possible parameter interactions. Based on all of the possible parameter interactions, AETG chooses a combination of values to maximize interaction coverage, building an efficient test set for the system. This selection process is performed "one-test-at-a-time" until all parameter interactions have been covered [11,13]. To enhance capability (i.e., to improve the test size), variants of AETG implementations are been developed, such as mAETG [14–19] and TCG [14,50].

Unlike AETG, IPO covers "one-parameter-at-a-time" through horizontal and vertical extension mechanisms, achieving a lower order of complexity than AETG [49]. As a t-way strategy, IPO has been extended into IPOG. Briefly, IPOG extends the "one-parameter-at-a-time" approach of IPO to support a higher t. The interaction parameters are first generated as a partial test suite that is based on the number of parameters and the interaction values. The test suite is then extended with the values of the next parameters using horizontal and vertical extension mechanisms. Horizontal extension extends the partial test suite with values of the next parameter to cover the maximum number of interactions. Upon completion of horizontal extension, vertical extension may be summoned to generate additional test cases that cover all of the uncovered interactions. More recently, a number of variants have been developed to improve IPOG performance (IPOG-D [35]; IPOF and IPOF2 [24]).

Arshem developed a freeware Java-based t-way testing tool called Test Vector Generator (TVG) [1], which extends the AETG strategy to support t-way testing [42–44]. Similar efforts have also been undertaken by Bryce and Colbourn to enhance AETG for t-way testing [5,6]. Nie et al. proposed a generalization of IPO with a genetic algorithm (GA), referred to as IPO_N and GA_N for t = 3. IPO_N performed better than GA_N in terms of test size and execution time [41].

Williams implemented a deterministic Java-based t-way test tool called Test Configuration (TConfig) [53]. TConfig consists of two strategies: recursive algorithm (RE) for t = 2 [55,56] and IPO for $2 \leqslant t \leqslant 6$ [52]. Williams reported that the recursive algorithm failed to cover all interactions for t > 2 [54]. For this reason, TConfig uses a minor version of IPO to cover the uncovered interactions in a greedy manner [54].

Finally, based on the analysis of the available algebraic and computational strategies, it is evident that, while most strategies strive to obtain the optimal test set, little attention has been given to support high interaction strengths (t > 6) or to automate the execution process in the tool implementation. GTWay serves as our research vehicle to investigate these issues.

3. Overview of the GTWay strategy

Within the context of software testing, GTWay is summarized in Fig. 1. GTWay is intended to assist test generation and execution.

Based on our earlier work on a pairwise strategy [29], the main algorithms forming the GTWay strategy are the parser algorithm, the t-way pair generation algorithm, the backtracking algorithm, and the executor algorithm (Fig. 2).

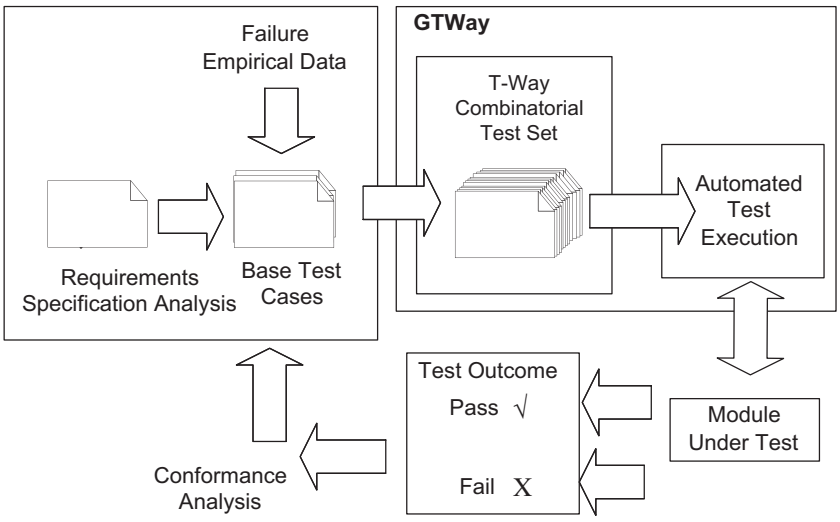

Fig. 1. Overview of GTWay.

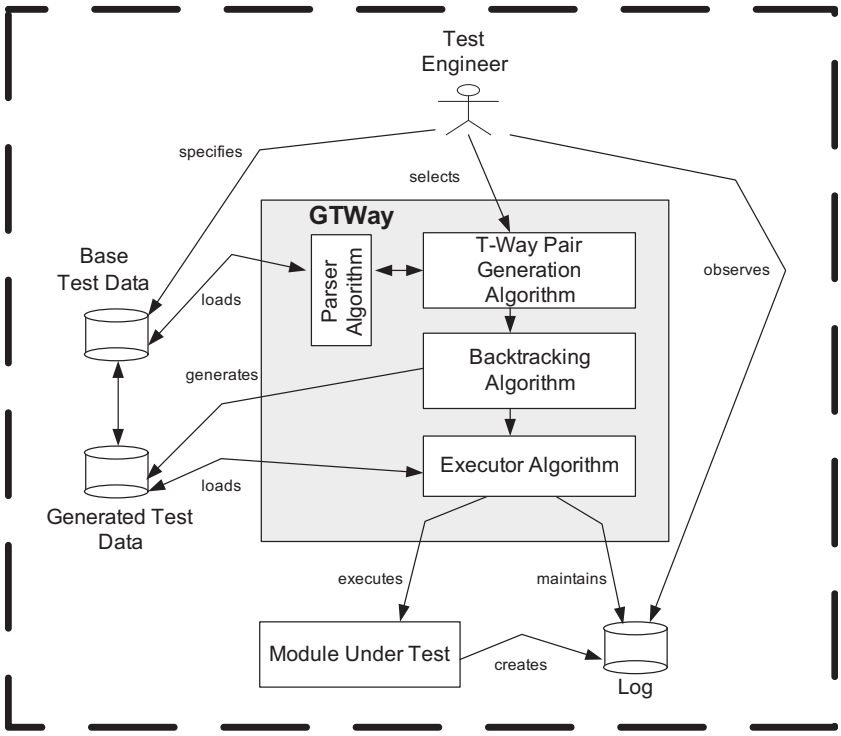

Fig. 2. Overview of the GTWay strategy and its implementation.

To initiate GTWay, the base test data must be manually specified using a special markup language, which is based on our earlier work [62]. Fig. 3 illustrates a snapshot of a specification of the base input test data expressed using the markup language (the keywords are shown in bold). Apart from capturing the input test data, the markup language also defines the values, data types, and access scopes, as well as describes the methods/functions that need to be tested. As discussed below, this information will be used by the executor algorithm to execute the test data by automatically generating a code driver to automate the actual testing process.

Upon execution of the GTWay strategy, the parser algorithm loads the parameters and values to be used by the t-way pair generation algorithm (discussed below). Then the parser algorithm represents the parameters and values by a symbolic representation for manipulation. Using this symbolic representation, the t-way pair generation algorithm can then generate the required t-way pairs.

Iterating through the t-way pairs, the backtracking algorithm generates the complete t-way test set. Upon completion, the backtracking algorithm forwards the results to the executor algorithm, which remaps the symbolic representation to actual values and then executes the test set.

3.1. The parser algorithm

The parser algorithm (Fig. 4) parses the module under the test information (specified in the fault file) to capture the necessary keywords and values to be used in the t-way generation and execution (e.g., className, methodName, paramNo, paramTypes, and returntype). In addition, the parser algorithm maps the actual values into symbolic representations and loads them into the parameter and value set. The rationale for mapping the actual values into symbolic representations is to ease the computation of merging and matching the t-way pairs in the backtracking algorithm.

3.2. The t-way pair generation algorithm

To illustrate how the GTWay strategy exploits the t-way pair generation algorithm, a running example (with four parameters and two values) is considered, assuming an interaction strength of t = 3.

The pair generation algorithm first identifies all of the possible three-way interactions. By referring to Table 1, the three-way interaction possibilities include the parameters ABC, ABD, ACD, and BCD. Based on these interactions, the t-way pair generation algorithm generates the following sets:

```
@FaultFile
////////////////////////////////////////
       Common Header Definition
////////////////////////////////////////
className : CollAccept
methodName : testAcceptance
specifier: public
paramTypes : 5
returnType: void
parameter : partypes[0]=Double.TYPE
parameter : partypes[1]=Double.TYPE
parameter : partypes[2]=Double.TYPE
parameter : partypes[3]=Double.TYPE
parameter : partypes[4]=Double.TYPE
////////////////////////////////////////
       Body - Test case 0
////////////////////////////////////////
arglist:arglist[0]=new Double(49)
arglist:arglist[1]=new Double(49)
arglist:arglist[2]=new Double(49)
arglist:arglist[3]=new Double(49)
arglist:arglist[4]=new Double(49)
////////////////////////////////////////
       Body - Test case 1
////////////////////////////////////////
arglist:arglist[0]=new Double(74)
arglist:arglist[1]=new Double(74)
.............
```

Fig. 3. Sample base test case specification.

$$\begin{aligned}
\mathrm{ABC} = \{&(a1,b1,c1),(a1,b1,c2),(a1,b2,c1),(a1,b2,c2),\\
&(a2,b1,c1),(a2,b1,c2),(a2,b2,c1),(a2,b2,c2)\}\\
\mathrm{ABD} = \{&(a1,b1,d1),(a1,b1,d2),(a1,b2,d1),(a1,b2,d2),\\
&(a2,b1,d1),(a2,b1,d2),(a2,b2,d1),(a2,b2,d2)\}\\
\mathrm{ACD} = \{&(a1,c1,d1),(a1,c1,d2),(a1,c2,d1),(a1,c2,d2),\\
&(a2,c1,d1),(a2,c1,d2),(a2,c2,d1),(a2,c2,d2)\}\\
\mathrm{BCD} = \{&(b1,c1,d1),(b1,c1,d2),(b1,c2,d1),(b1,c2,d2),\\
&(b2,c1,d1),(b2,c1,d2),(b2,c2,d1),(b2,c2,d2)\}
\end{aligned}$$

The generated sets serve two purposes. First, any one of these sets can be merged together with another set to form a complete test suite (e.g., ABC and ABD). Second, all of the elements in the sets can be used to countercheck that all of the pairs are indeed covered.

The t-way pair generation algorithm initially finds the loop edge of the t-way combinations (based on the number of defined parameters, P). The algorithm then performs index searches through a loop from 0 to $2^p - 1$. For each index, the algorithm converts the number to binary format. If the number of binary 1's in the index is equal to the value of t (i.e., the index represents a t-way interaction), the specific index is included in the index set.

Using the same example, Table 1 indicates that the loop edge is 15 (i.e., $2^4 - 1$). In this case, the index search loop finds four indexes. Each index has three one's (for 3-way combinations). These indexes are 7, 11, 13, and 14 (Table 2).

In the t-way pair generation algorithm, each index contains a number of t-way combinations equal to the multiplication of values defined in each shared parameter. In our example, each of the first, second, third, and the fourth indexes separately contains $2 \times 2 \times 2$ combinations. Thus, there are a total of 32 combinations.

To minimize the access time and space requirements, an efficient data structure (structure of bits) was designed. Row indexes are used to store the indexes of the t-way combinations. Using our example, row index 0 corresponds to the (A,B,C) combinations and stores eight combinations, which are indicated as bits b0 to b7. Similarly, Row Index 1, Row Index 2, and Row Index 3 each separately stores eight combinations (Table 3).

Having described its implementation, the t-way pair generation algorithm can then be summarized as shown by Fig. 5.

```
Algorithm Parser (fault_file:File)
1: begin
2:  load fault_file
3:  let p = {} the empty set, where p represents the defined parameters
4:  let n∑ = {} the empty set, where n∑ represents the parameter values
5:  while not EOF (fault_file)
6:   begin
7:    read value
8:    if keyword in value = {className or methodName or paramNo or
      paramType or returnType or parameter}
9:      begin
9:        parse value
11:       assign corresponding value to className or methodName or
          paramNo or paramType or returnType or parameter
12:     end
10:   if keyword in value = {arglist[i]...arglist[paramNo]}
      where i<=0<=paramNo
11:    begin
12:      parse value
13:      assign i to the p set as symbolic representation
14:      assign value to n∑ set as symbolic representation
15:    end
16: end
```

Fig. 4. The parser algorithm.

Table 1
Base test values.

Base value	Input variables			
	A	B	C	D
	a1	b1	c1	d1
	a2	b2	c2	d2

Table 2
Index search for a four-parameter system.

Index	0	1	2	3	4	5	6	7
Binary	0000	0001	0010	0011	0100	0101	0110	0111
Index	8	9	10	11	12	13	14	15
Binary	1000	1001	1010	1011	1100	1101	1110	1111

Table 3
Row index for a four-parameter system.

Row index	Index		Bits							
			b7	b6	b5	b4	b3	b2	b1	b0
0	7	→	1	1	1	1	1	1	1	1
1	11	→	1	1	1	1	1	1	1	1
2	13	→	1	1	1	1	1	1	1	1
3	14	→	1	1	1	1	1	1	1	1

```
Algorithm Pair_Generation (t : t-way value)
1: begin
2:  let Sp ={} the empty set, where Sp represents the pair set
3:  let n∑ = {n0.....nm} where n∑ represents the values defined for
    each parameter, m = max no of parameters
4:  let p = {p0 ..pj}, where p represents the sorted set of sets of
    values defined for each parameter
5:  for index=0  to 2 m - 1
6:   begin
7:     let b = binary number
8:     b = convert index to binary
8:     if (the no of  '1's in b = t)
9:      begin
10:      calculate number of possible combinations (PCi )between the
         partial sets of values
11:      for the shared parameters
12:         begin
13:          multiply {nx x ny} values from n∑
14:          set the bits group (equal to PCi) in the index row to 1
15:         end
16:     end
17:   end
18: end
```

Fig. 5. The t-way pair generation algorithm.

3.3. The backtracking algorithm

Backtracking algorithms have been discussed by Yan and Zhang [57,58] and Hnich et al. [27]. Here, the backtracking algorithms employed an exhaustive search method to search for the optimal suite. As such, these algorithms require a long execution time and may be subjected to a small number of configurations. Unlike the aforementioned algorithms, GTWay's backtracking algorithm is designed to be a flexible heuristic algorithm that does not rely on exhaustive search methods.

Superficially, the backtracking algorithm implemented in GTWay appears to be similar to IPOG's horizontal extension. However, a closer look reveals one fundamental difference in terms of the construction of the minimal test suite. In GTWay, the test suite is constructed in a one-test-at-a-time fashion (similar to AETG). GTWay relies on a backtracking search procedure with a defined merger rule, which goes through the uncovered t-way pairs via recombination as a way to minimize the test cases. Although it would not necessarily be chosen as the best fit value (i.e., one that covers the most uncovered t-way pairs), GTWay always produces a complete test case (i.e., with all of the defined parameter values) after each backtracking iteration. In contrast, IPOG's horizontal extension searches the parameter values in a one-parameter-at-a-time fashion. In this manner, IPOG will always incrementally produce a partial test case until the completion of all of the horizontal (and possibly vertical) extensions. Because of this characteristic, IPOG produces and commits a complete (and optimal) test case at a later stage compared to GTWay.

For details and a step-by-step description of the backtracking algorithm, refer to the earlier example in Table 1. To further illustrate, assume that the backtracking algorithm chose sets ABC and ABD as mergers. The merger rule functions as follows:

- Two elements for each t-way pair sets can only be merged when they are combinable (i.e., each pair element complements the other's missing value).
- A test case is selected as a result of the merger only if it covers most of the uncovered t-way pairs. This is carried out to ensure that the optimal test suite is generated at the end.
- In cases when some pairs cannot be merged (due to values that are not uniform), the backtracking algorithm falls back to the first defined values.

By applying the merger rule while traversing the three-way pair sets of ABC and ABD, the first combinable elements shall be the first elements for both sets. When the first element in ABC (a1,b1,c1) is merged with the first element in ABD

(a1,b1,d1), the resulting test case is (a1,b1,c1,d1). Because the test case covers all of the new three-way pairs: (a1,b1,c1), (a1,b1,d1), (a1,c1,d1), and (b1,c1,d1), the resulting test case is selected for the final test suite. Upon selection, the covered pairs are deleted from their respective sets. The next combinable element within ABC is (a1,b1,c2) and the second element in ABD is (a1,b1,d2), of which the resulting test case is (a1,b1,c2,d2). Because the test case covers all of the new three-way pairs: (a1,b1,c2), (a1,b1,d2), (a1,c2,d2), and (b1,c2,d2), the resulting test case is selected for in the final test suite, and the covered pairs are deleted from their respective sets. Now, the next combinable element within ABC is (a1,b2,c1), and the third element in ABD is (a1,b2,d1). The resulting test case is (a1,b2,c1,d1). In this case, the three-way pairs covered are (a1,b2,c1), (a1,b2,d1), (a1,c1,d1), and (b2,c1,d1). Because the covered pair (a1,c1,d1) exists from an earlier merger, the resulting test case, (a1,b2,c1,d1), is not selected in the final test suite (because it does not cover the most uncovered three-way pairs). The traversing and merging process continues until all of the three-way pairs are covered. In this example, the final optimal test suite for $t = 3$ consists of {(a1,b1,c1,d1), (a1,b1,c2,d2), (a1,b2,c1,d2), (a1,b2,c2,d1), (a2,b1,c1,d2),(a2,b1,c2,d1),(a2,b2,c1,d1), and (a2,b2,c2,d2)}. Upon completion of the final test suite, all of the sets (i.e., ABC, ABD, ACD, and BCD) should be empty, indicating that each three-way pair element within them has been covered.

Based on the above discussion, the backtracking algorithm is summarized in Fig. 6.

```
Algorithm Backtracking (Sp: Set;var St: Set)
1: begin
2:  let St ={} the empty set, where St represents the generated test set
3:  for the first two parameters
4:   begin
5:     create partial the test cases by selecting best values for
       higher parameters {P3...Pj}, that covers the maximum number of
       uncovered pairwise combinations in Sp
6:     store generated test cases in St
7:     remove covered pairs from Sp (by setting zero values
       to indicated bits).
8:   end
9:   while still found elements in Sp
10:    begin
11:      add a new element in the St set with empty fields
12:      bring the first uncovered combination, decompose and fills the
         initial value in the element set
13:      for the 2nd uncovered combination
14:        begin
15:          decompose uncovered combination
16:          if (current pair element in Sp can be combined with
             other pair element)
17:            begin
18:              count number of uncovered combination
19:              if (has most uncovered pairs)
20:                fill it in the element set
21:            end
22:          if (the element set does not have matching pair)
23:            select the first element as default values to the
               missing parameters
24:          store it in St and remove the covered pairs from Sp
25:        end
26:    end
27: end
```

Fig. 6. The backtracking algorithm.

3.4. Execution algorithm

As discussed earlier, GTWay can be employed to facilitate t-way test generation, automation, and (concurrent) test execution. To enable execution, the symbolic data representations must be remapped into actual data values in the test data specification (i.e., in the fault file). In GTWay, execution is supported by the executor algorithm. The executor algorithm simply takes the name of the defined class and methods (as well as the associated parameters and values) and automatically generates an independent test driver to drive execution (as taken from the parser algorithm). In this manner, concurrent execution can be carried out through the judicious use of threads. Upon execution, the test results are also captured in a test log for conformance analysis.

A complete description of the executor algorithm is depicted in Fig. 7.

4. Evaluation

The accuracy of the GTWay strategy has been demonstrated in our previous work [29]. In this paper, our evaluation focuses on two main goals: (1) to assess the effectiveness of the GTWay strategy for general t-way test data generation and execution and (2) to compare the performance of GTWay with existing strategies, particularly by examining the size and the time taken to produce these test sets. In the following sub-sections, we will present our complete evaluations based on the aforementioned goals.

4.1. Assessment of GTWay t-way test data generation and execution support

To demonstrate support for high levels of interaction, we opted to use program source codes consisting of highly interacting input variables. We envisioned a hypothetical program called *college_acceptance* that can automatically report student acceptance for college admissions. In this program, it is assumed that the college includes four main departments: the Department of Mathematics, the Department of Physics, the Department of Biology, and the Department of Computing. The acceptance criteria of any of the departments depend on the student's high school grades in eight subjects: English, mathematics, physics, biology, computer science, art, economics, and social science. In this hypothetical problem, a student can only be accepted into one of the departments if the following criteria are met:

- He/she passes all eight subjects (i.e., a score of 50% or better must be achieved in each subject).
- He/she scores 75% or better in the subjects that are related to the department he/she is applying for.
- The acceptance will be conditional if the English subject score is less than 75%.

Designed intentionally in this manner, it can be observed that each acceptance criterion is highly intertwined and interdependent with the others. Thus, it is expected that pairwise interactions (t = 2) may not be sufficient for good coverage.

```
Algorithm Executor (className, methodName, paramNo, paramTypes,
                    returnType, parameter: String; St:Set)
1: begin
2:  for each test case, i, in St set
3:   begin
4:    map the actual values from symbolic representation
5:    create a public driver class specified in className
6:    generate and compile the main method for the driver class
      with specific call to the method under test, methodName,
      by passing the ith test case from St with the correct passing
      parameter (e.g., paramNo, paramTypes and returnType)
7:    instantiate a driver object
8:    if (thread<limit)
9:      spawn and execute the thread for the created driver object
10:   capture the result and errors (if any) in log
11:  end
12: end
```

Fig. 7. The executor algorithm.

To serve as our case study, we implemented the *college_acceptance* program using the Java programming language. The implemented *college_acceptance* program consists of 1 class, 2 methods, 660 blocks, and 64 lines. The two methods in the program are the *main ()* and the *testAcceptance ()* methods. The *testAcceptance* () method covers eight parameters of type double that correspond to the scores in each individual subject (i.e., English, mathematics, physics, biology, computer science, art, economics, and social science). Considering the subject scores based on the earlier set of criteria, a decision is made and printed: incorrect grades, not accepted in any department, conditionally accepted in a specific department, or accepted in a specific department.

Using the equivalence partitioning technique, the grade level can be divided into two main sets: the valid value set and the out-of-range value set. The valid set can be further divided into three intervals: between [0,50), between [50,75), and between [75,100]. Similarly, the out-of-range value set can be further divided into two intervals: negative values and values greater than 100. Table 4 summarizes the selected base values for all of the valid and out-of-range value sets. The first value, 49, belongs to the first interval in the valid set. The second value, 74, belongs to the second interval, while the third value, 76, belongs to the last interval in the valid set. The fourth value, −1, belongs to the first interval in the out-of-range set. Finally, the fifth value, 101, belongs to the second interval in the out-of-range set. An excerpt snapshot of the test data specification for the base test data appears in Fig. 3.

We used GTWay to generate the test cases for t = 2 to t = 8 (where t = 8 includes the exhaustive combinations) for both valid and out-of-range sets. We concurrently executed all of the test cases against the *college_acceptance* program.

In this case, the (concurrent) execution support is a useful automation feature of GTWay. While most t-way strategies are useful test generators (i.e., they assist in the test planning process), they often do not provide any automated execution. Frequently, the generated test data must be manually converted into some format prior to execution. In GTWay, these aspects of the execution are automated. As discussed earlier, GTWay permits the actual data values and its control parameters (e.g., class name, types, and access rights) to be specified in the fault file using a defined markup language. Because of this

Table 4
Base test cases.

	Math	Physics	Biology	English	Computer science	Art	Economics	Social science
Valid values	49	49	49	49	49	49	49	49
	74	74	74	74	74	74	74	74
	76	76	76	76	76	76	76	76
Out-of-range values	−1	−1	−1	−1	−1	−1	−1	−1
	101	101	101	101	101	101	101	101

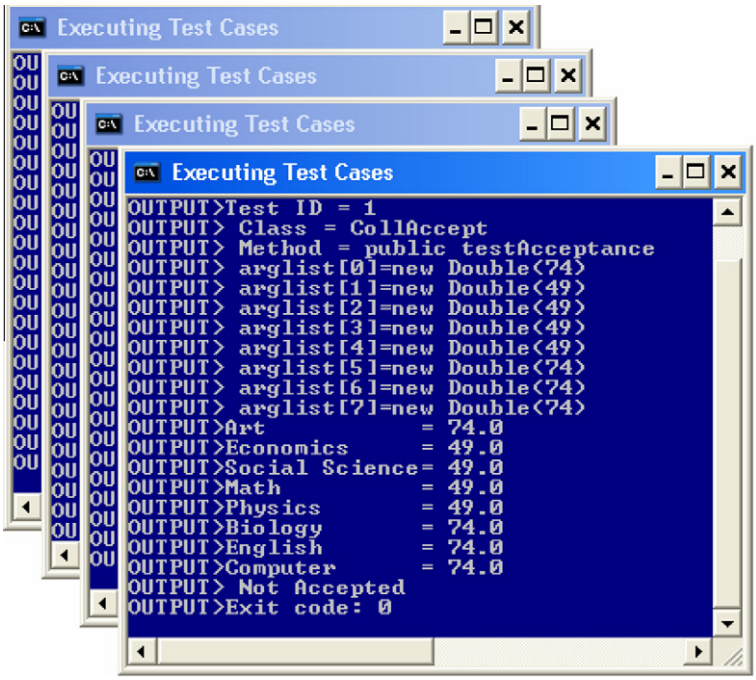

Fig. 8. Concurrent execution snapshot.

feature, the execution of test data in GTWay involves parsing the data values and their control parameters (as defined in the fault file) to automatically drive the execution and capture the execution log for conformance analysis.

The fact that GTWay can assist in execution introduces some timing overhead to parse the parameter values and controls (as defined in the fault file). From an automation perspective, the timing overhead is justified; it alleviates the burden of mapping (and executing) the values and control manually. Our experience indicates that the timing overhead is directly proportional to the number of defined base test cases (approximately 50 μs per defined test).

Referring back to our execution of the *college_acceptance* program, we observed no errors, and the program behaved as expected (Fig. 8). To help measure coverage, we adopted the open source coverage tool EMMA [51] from SourceForge. Using EMMA, a number of coverage metrics were reported. The first coverage metric is the class coverage. In EMMA, the class coverage refers to the ratio of covered classes to the total number of classes. The second metric, the method coverage, refers to the ratio of covered methods to the total number of methods. The third metric, the block coverage, is defined as the ratio of the total number of covered blocks to the total number of blocks. Finally, the last metric, the line coverage, is defined as the ratio of the number of covered lines to the total number of lines.

The coverage results are listed in Table 5 and summarized in Fig. 9. Two conclusions can be derived from this case example. First, GTWay can generate the required set of test data, as seen at t = 6 in Fig. 9, and give 100% coverage in the same manner as exhaustive test data would. The second conclusion confirms our intuition that pairwise testing does not render good coverage for highly interacting systems. Class coverage and method coverage lines are flat at 100% for all of the values of t. Block coverage and line coverage appear to be approximately linear when t is within the interval from 2 to 6; both metrics then saturate at t = 6. At this point, 100% coverage is achieved for all of the metrics (i.e., class coverage, method coverage, block coverage, and line coverage). In this case, GTWay has successfully reduced the number of test cases from 6817 (6561 + 256) to 1532 (1420 + 112), a reduction of over 75%.

4.2. Adoption of GTWay for hardware test design

To further validate GTWay and highlight the need to accomplish a large number of high interactions, we also adopted a case study involving combinatorial hardware logic design. With the current technology in hardware logic design, we were able to take advantage of the reprogrammable feature and implement any combinatorial circuits into a field-programmable gate array (FPGA) board; this implementation allowed us to arrive at a generic and low-cost solution. One of the common problems in FPGA-based designs is the need to accurately map the hardware design using a specific hardware description

Table 5
Number of test cases with coverage for the *college_acceptance* implementation.

t	Number of test cases		Class coverage (%)	Method coverage (%)	Block coverage (%)	Line coverage (%)
	Valid set	Out-of-range set				
2	18	8	100	100	30	23
3	58	16	100	100	50	39
4	192	38	100	100	69	60
5	539	68	100	100	96	93
6	1420	112	100	100	100	100
7	2655	128	100	100	100	100
8	6561	256	100	100	100	100

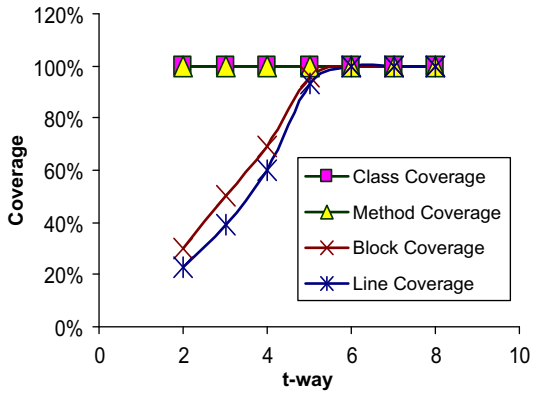

Fig. 9. Percentage coverage chart for *college_acceptance*.

language (HDL) that supports FPGA boards. To evaluate the accuracy of the mapping, one must ensure that the HDL description conforms to the hardware implementation specification.

For our case study, a 4-bit magnitude comparator design (Fig. 10) was used for variant implementation, as suggested by Mano [40]. Using Java as the hardware description language, as in JHDL [3], the 4-bit magnitude comparator design in Fig. 10 can be expressed as the following equivalent Java program (see Fig. 11).

Rather than exhaustively testing all parameter inputs with 2^8 possibilities, we adopted a mutation tool called MuJava [38] to verify the accuracy of our design. Complementary to GTWay, MuJava is a fault injection tool that permits mutated Java code (which is based on some defined operators) to be injected into a running Java program.

In our case study, we introduced 140 mutants (i.e., variations in the logical operators) into our 4-bit magnitude comparator design using MuJava. We then used GTWay to incrementally generate the t-way tests for various values of interactions to kill the mutants. Table 6 highlights our findings.

Table 6 indicates that, while t = 2 provides a good 92% mutant score (e.g., 110 out of 140 mutants killed), it is not until t = 4 that all mutants were removed completely by the generated test set. The findings indicate that an exhaustive combination (i.e., t = 8) was not necessary to kill all of the mutants. Apart from supporting the earlier conclusions in Section 4.1, this finding also illustrates the applicability of GTWay to facilitate the verification of logic designs.

4.3. Comparison of GTWay with others strategies

A number of experiments adopted from Lei et al. [36] were performed to analyze the performance of GTWay in terms of both test size and execution time. However, unlike those utilized by Lei et al., other t-way strategies, such as IPOG, WHITCH, Jenny, TConfig, and TVG, were included in the analysis for comparative purposes. The experiments were divided into four groups:

i. Group 1: The number of parameters (P) and values (V) are constant, but the coverage strength (t) is varied from 2 to 6.
ii. Group 2: The coverage strength (t) and the values (V) are constant (at 4 and 5, respectively), but the number of parameters (P) is varied from 5 to 15.
iii. Group 3: The number of parameters (P) and the coverage strength (t) are constant (at 10 and 4, respectively), but the values (V) are varied from 2 to 10.
iv. Group 4: The Traffic Collision Avoidance System Module (TCAS) with 12 parameters (two 10-valued parameters, one 4-valued parameter, two 3-valued parameters, and seven 2-valued parameters).

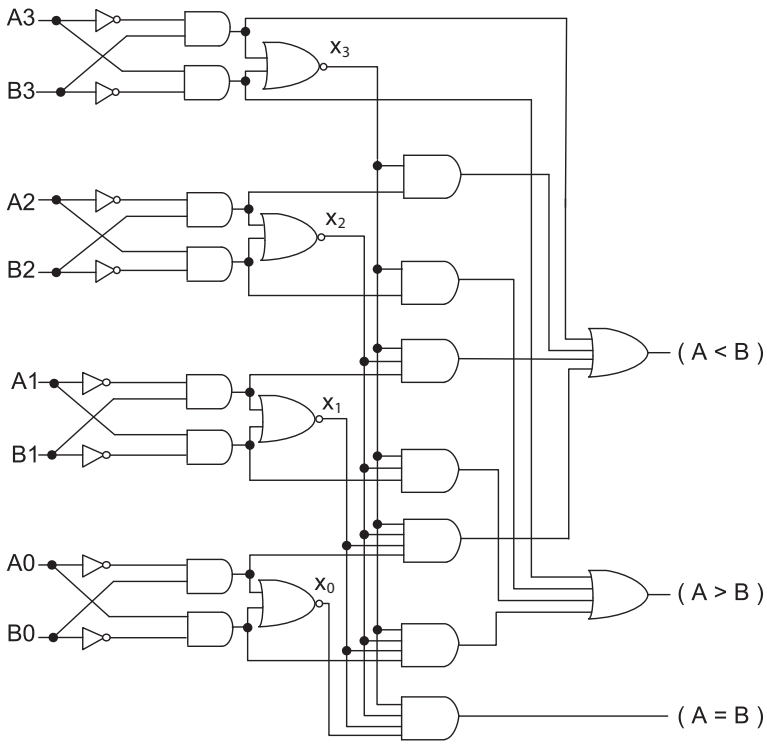

Fig. 10. Schematic diagram for the 4-bit magnitude comparator.

```
public class Comparator {

   public static String Compare
           (boolean A3,  boolean A2, boolean A1, boolean A0,
            boolean B3,Boolean B2,boolean B1, boolean B0 )
   {
    boolean    g1,g2,g3; //outputs gates
    boolean    x3,x2,x1,x0;// intermediate gates
    String s=null;

    x3=!(!A3&A3|A3&!A3);
    x2=!(!A2&B2|B2&!B2);
   x1=!(!A1&B1|A1&!B1);
   x0=!(!A0&B0|A0&!B0);

    g1=(!A3 &B3)|(!A2 &B2)&x3|
        (!A1 &B1)&x2 &x3|(!A0 &A0)&x1&x2 &x3; //A<B

    g2=(A3 &!B3)|(A2 &!B2)&x3|
         (A1 &!B1)&x2 &x3|(A0 &!B0)&x1&x2 &x3; //A>B

    g3=x3&x2&x1&x0;  //A=B

     s=g1+""+g2+""+g3;
   return s;
   }

}
```

Fig. 11. Equivalent class Java program for the 4-bit magnitude comparator.

Table 6
Percentage of mutant scores for the 4-bit magnitude comparator (total: 140).

t-Way	Test size	Live mutants	Killed mutants	% Mutant score
2	8	10	130	92
3	16	7	133	95
4	38	0	140	100

In our experiment, all of the above-mentioned strategies were employed within our environment, which consisted of a desktop PC with Windows XP, 1.6 GHz CPU, 1 gigabyte of RAM, and JDK 1.5 installed.

Tables 7–14 highlight the results obtained for each experimental group. We reported the results for size and time in two separate tables for each experiment. Table 7 shows darkened cell entries for each row to indicate the best performance in terms of test size and execution time. Note, however, that some rows have more than one cell entry with the same test size; thus, more than two cell entries were darkened. Cells marked NA (“not available”) indicate that the results were unavailable after one day even though the (P,V,t) values could be selected. Cells marked as NS (“not supported”) indicate that the tool could not generate the test case for those specific (P,V,t) values.

A number of observations can be summarized based on the results in Tables 7–14. First, no single strategy is absolutely dominant over the other strategies. IPOG is the best strategy in terms of execution time, due to its lightweight and deterministic algorithm implementation. The closest competitor to IPOG is Jenny. GTWay performs considerably well within acceptable values. On average, TConfig appears to have the slowest execution time.

Table 7
Group 1 (size): P and V were constants (10,5), but t varied up to 6.

t-Way	IPOG Size	WHITCH Size	Jenny Size	Tconfig Size	TVG II Size	GTWay Size
2	48	45	45	48	50	46
3	308	225	290	312	342	293
4	1843	1750	1719	1878	1971	1714
5	10,119	NS	9437	NA	NA	9487
6	50,920	NS	NS	NA	NA	44,884

Table 8
Group 1 (time): P and V were constants (10,5), but t varied up to 6.

t-Way	IPOG Time (s)	WHITCH Time (s)	Jenny Time (s)	Tconfig Time (s)	TVG II Time (s)	GTWay Time (s)
2	0.11	1	0.43	1	0.141	0.12
3	0.56	23	0.78	88.62	5.797	1.53
4	6.38	350	17.53	>8 h	276.328	19.85
5	63.8	NS	500.93	>24 h	>24 h	338.84
6	791.35	NS	NS	>24 h	>24 h	5545.2

Table 9
Group 2 (size): t and V were constants (4,5), but P varied (from 5 up to 15).

P	IPOG Size	WHITCH Size	Jenny Size	Tconfig Size	TVG II Size	GTWay Size
5	784	625	837	773	849	731
6	1064	625	1074	1092	1128	1027
7	1290	1750	1248	1320	1384	1216
8	1491	1750	1424	1532	1595	1443
9	1677	1750	1578	1724	1795	1579
10	1843	1750	1719	1878	1971	1714
11	1990	1750	1839	2038	2122	1852
12	2132	1750	1964	NA	2268	2022
13	2254	NA	2072	NA	2398	2116
14	2378	NA	2169	NA	NA	2222
15	2497	NA	2277	NA	NA	2332

Table 10
Group 2 (time): t and V were constants (4,5), but P varied (from 5 up to 15).

P	IPOG Time (s)	WHITCH Time (s)	Jenny Time (s)	Tconfig Time (s)	TVG II Time (s)	GTWay Time (s)
5	0.19	5.26	0.44	31.46	1.468	0.047
6	0.45	14.23	0.71	231.56	5.922	0.467
7	0.92	59.56	1.93	1120	18.766	2.834
8	1.88	115.77	4.37	>1 h	55.172	6.355
9	3.58	210.87	9.41	>3 h	132.766	12.932
10	6.38	350	17.53	>8 h	276.328	19.85
11	10.83	417	30.61	>23 h	548.703	44.54
12	17.52	628.94	50.22	>24 h	921.781	73.68
13	27.3	>24 h	76.41	>24 h	1565.5	125.891
14	41.71	>24 h	115.71	>24 h	>24 h	191.22
15	61.26	>24 h	165.06	>24 h	>24 h	299.29

Despite being the fastest strategy in all of the experimental groups, IPOG does not provide the optimal test size. For experimental Groups 1 and 2, Jenny mostly produced the optimal test size relative to the other strategies. However, for experimental Groups 3 and 4, GTWay outperformed all of other strategies in test size, followed closely by the IPOG.

WHITCH and TConfig appear to cater only to small values of t (a maximum of t = 4 for the TCAS module, as shown in Tables 13 and 14). GTWay is the only strategy that was able to address higher orders of t in all of the experimental groups. Jenny and IPOG performed well with higher orders of t, relative to TConfig, TVG, and WHITCH. For the TCAS module given in

Table 11
Group 3 (size): P and t were constants (10,4), but V varied (from 2 up to 10).

V	IPOG Size	WHITCH Size	Jenny Size	Tconfig Size	TVG II Size	GTWay Size
2	46	58	39	45	40	46
3	229	336	221	235	228	224
4	649	704	703	718	782	621
5	1843	1750	1719	1878	1971	1714
6	3808	NA	3519	NA	4159	3514
7	7061	NA	6482	NA	7854	6459
8	11,993	NA	11,021	NA	NA	10,850
9	19,098	NA	17,527	NA	NA	17,272
10	28,985	NA	26,624	NA	NA	26,121

Table 12
Group 3 (time): P and t were constants (10,4), but V varied (from 2 up to 10).

V	IPOG Time (s)	WHITCH Time (s)	Jenny Time (s)	Tconfig Time (s)	TVG II Time (s)	GTWay Time (s)
2	0.16	1	0.47	14.43	0.297	0.672
3	0.547	120.22	0.51	379.38	3.937	2.357
4	1.8	180	4.41	>1 h	46.094	7.92
5	6.33	350	17.53	>8 h	276.328	28.58
6	16.44	>24 h	134.67	>24 h	1,273.469	74.88
7	38.61	>24 h	485.91	>24 h	4,724	177.6
8	83.96	>24 h	1410.27	>24 h	>24 h	403.01
9	168.37	>24 h	2125.8	>24 h	>24 h	825.95
10	329.36	>24 h	5458	>24 h	>24 h	1650.65

Table 13
Group 4 (size): TCAS module (12 multi-valued parameters, t varied from 2 to 12).

t-Way	IPOG Size	WHITCH Size	Jenny Size	Tconfig Size	TVG II Size	GTWay Size
2	100	120	108	108	101	100
3	400	2388	413	472	434	402
4	1361	1484	1536	1476	1599	1429
5	4219	NS	4580	NA	4773	4286
6	10,919	NS	11,625	NA	NS	11,727
7	NS	NS	27,630	NS	NS	27,119
8	NS	NS	58,865	NS	NS	58,584
9	NS	NS	NA	NS	NS	114,411
10	NS	NS	NA	NS	NS	201,728
11	NS	NS	NA	NS	NS	230,400
12	NS	NS	NA	NS	NS	460,800

Table 14
Group 4 (time): TCAS module (12 multi-valued parameters, t varied from 2 to 12).

t-Way	IPOG Time (s)	WHITCH Time (s)	Jenny Time (s)	Tconfig Time (s)	TVG II Time (s)	GTWay Time (s)
2	0.8	0.73	0.001	>1 h	0.078	0.021
3	0.36	1,020	0.71	>12 h	2.625	0.933
4	3.05	5,400	3.54	>21 h	104.093	13.567
5	18.41	NS	43.54	>24 h	1,975.172	97.33
6	65.03	NS	470	>24 h	NS	391.84
7	NS	NS	2,461.1	NS	NS	965.74
8	NS	NS	11,879.2	NS	NS	2418.81
9	NS	NS	>1 d	NS	NS	3544.18
10	NS	NS	>1 d	NS	NS	2670.68
11	NS	NS	>1 d	NS	NS	94.52
12	NS	NS	>1 d	NS	NS	11.647

Tables 13 and 14, Jenny is the only strategy other than GTWay that deals with values beyond t = 6. Note that the parameters and values for the TCAS module are two 10-valued parameters, two 3-valued parameters, one 4-valued parameter, and seven 2-valued parameters. Based on the findings presented in the previous sub-section (Table 5), supporting values up to t = 6 can be significantly important for testing some classes of a highly interacting system.

A subtle observation can be seen in the execution time for GTWay in the cases of t = 10, t = 11, and t = 12 in Table 14 (which involve the TCAS module), compared to earlier execution times. At a glance, the results appear to be counterintuitive. At t = 10, the execution time is 2670.68 s, but the execution time decreases to 94.52 s at t = 11. At t = 12, the execution time drops significantly to only 11.647 s. In the case of t = 11, the computation becomes lighter compared to the earlier case (i.e., t = 10), because the pair generation and backtracking search algorithms now deal with significantly fewer t-way pair combinations for the merger, which was close to an exhaustive number of combinations. For t = 12 (the exhaustive number of combinations), GTWay relies on a different algorithm (i.e., it does not use the backtracking search algorithm) and degrades to merely a pair generation algorithm. By doing so, GTWay is able to achieve a good execution time.

Concerning the overall performance of GTWay, a number of observations should be discussed. Tables 7, 9, and 11 indicate that the test size grows exponentially, logarithmically, and quadratically in term of the strength of coverage, number of parameters, and number of values. These results are consistent with the theoretical expectation $O(v^t \log p)$ [13]. The same observation can be derived for the execution times given in Tables 8, 10, and 12.

Finally, we conclude that obtaining an optimal size and a fast execution time are two sides of the same coin. On one hand, a t-way strategy may be fast but generate a non-optimal solution when applied to a large test size. On the other hand, a t-way strategy may provide an optimal size at the expense of a slower execution time (possibly due to the need for some optimization process). Our present work has added a new dimension to the above-mentioned criteria by examining whether the t-way strategy can be fully maximized for both the test generation and the (automated) test execution.

5. Conclusion

In this paper, we proposed and implemented GTWay, an efficient strategy for t-way testing. Experimental results demonstrate that GTWay scales well against other strategies (e.g., WHITCH, Jenny, TConfig, TVG, and IPOG) in terms of the generated test size. Furthermore, GTWay appears to be the only strategy that addresses higher orders of t and integrates test generation with execution as part of the tool implementation. Although it does not provide the best execution time, GTWay is considered acceptable (due to the overhead incurred in the input–output processing and parsing of the external file for base inputs) to permit support for automated execution. In future work, we plan to incorporate constraints and seedings into GTWay to improve its performance.

Overall, our experience with the current implementation prototype demonstrates that the general architecture of GTWay is scalable. Although it currently supports automated test execution based on Java, GTWay would permit a seamless extension to support other execution platforms by integrating the parser and executor algorithms as add-on components. However, while the current GTWay prototype implementation also supports test execution for both large and small modules, more work needs to be done to ensure full support for the test execution when involving modules that require significant human intervention (e.g., GUI-based interactions). Finally, for support test generation and execution for very high levels of interaction and large parameters and values (e.g., t = 2 through t = 10 for 100 parameters and 10 values), we are currently investigating a new prototype implementation of GTWay that can be utilized within the GRID environment.

Acknowledgements

The authors acknowledge the help of Jeff Lei, Raghu Kacker, Rick Kuhn, Myra B. Cohen, and Bob Jenkins for providing us with useful comments and background materials. This research is partially funded by the USM Fundamental Research Grant, "Investigating Heuristic Algorithm to Address Combinatorial Explosion Problem" and the USM Research University Grant, "Development of Variable Strength Interaction Testing Strategy for T-Way Test Data Generation."

References

[1] J. Arshem, TVG. <http://sourceforge.net/projects/tvg> (last accessed 24.10.10).
[2] J. Bach, AllPairs Test Generation Tool, Version 1.2.1. <http://www.satisfice.com/tools.shtml> (last accessed 24.10.10).
[3] Brigham-Young-University, Logic Design. <http://www.jhdl.org/documentation/users_manual/intro.html> (last accessed 24.10.10).
[4] R. Bryce, C.J. Colbourn, Prioritized interaction testing for pairwise coverage with seeding and avoids, Information and Software Technology Journal 48 (10) (2006) 960–970.
[5] R. Bryce, C.J. Colbourn, M.B. Cohen, A framework of greedy methods for constructing interaction tests, in: Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, USA, 2005, pp. 146–155.
[6] R.C. Bryce, C.J. Colbourn, A density-based greedy algorithm for higher strength covering arrays, Software Testing, Verification, and Reliability 19 (1) (2009) 37–53.
[7] K.A. Bush, Orthogonal Arrays of Index Unity, The Annals of Mathematical Statistics 23 (3) (1952) 426–434.
[8] K.-Y. Cai, Z. Dong, K. Liu, Software testing processes as a linear dynamic system, Information Sciences 178 (6) (2008) 1558–1597.
[9] K.-Y. Cai, B.-B. Yin, Software execution processes as an evolving complex network, Information Sciences 179 (12) (2009) 1903–1928.
[10] C. Catal, B. Diri, Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem, Information Sciences 179 (8) (2009) 1040–1058.

[11] D.M. Cohen, S.R. Dalal, M.L. Fredman, G.C. Patton, The AETG system: an approach to testing based on combinatorial design, IEEE Transactions on Software Engineering 23 (7) (1997) 437–444.
[12] D.M. Cohen, S.R. Dalal, A. Kajla, G.C. Patton, The automatic efficient test generator (AETG) system, in: Proceedings of the 5th International Symposium on Software Reliability Engineering, IEEE Computer Society, Monterey, CA, USA, 1994, pp. 303–309.
[13] D.M. Cohen, S.R. Dalal, J. Parelius, G.C. Patton, The combinatorial design approach to automatic test generation, IEEE Software 13 (5) (1996) 83–88.
[14] M.B. Cohen, Designing Test Suites for Software Interaction Testing, Ph.D. Thesis, Computer Science, University of Auckland, 2004.
[15] M.B. Cohen, C.J. Colbourn, A.C.H. Ling, Constructing strength three covering arrays with augmented annealing, Discrete Mathematics 308 (13) (2008) 2709–2722.
[16] M.B. Cohen, M.B. Dwyer, J. Shi, Coverage and adequacy in software product line testing, in: Proceedings of the ISSTA Workshop Role of Software Architecture for Testing and Analysis, ACM, Portland, Maine, USA, 2006, pp. 53–63.
[17] M.B. Cohen, M.B. Dwyer, J. Shi, Exploiting Constraint Solving History to Construct Interaction Test Suites, in: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques (MUTATION 2007), IEEE Computer Society, UK, 2007, pp. 121–132.
[18] M.B. Cohen, M.B. Dwyer, J. Shi, Interaction testing of highly-configurable systems in the presence of constraints, in: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ACM, London, UK, 2007, pp. 129–139.
[19] M.B. Cohen, P.B. Gibbons, W.B. Mugridge, C.J. Colbourn, Constructing test suites for interaction testing, in: Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon USA, 2003, pp. 38–48.
[20] C.J. Colbourn, M.B. Cohen, R.C. Turban, A deterministic density algorithm for pairwise interaction coverage, in: Proceedings of the IASTED International Conference on Software Engineering, vol. 17, Innsbruck, Austria, 2004, pp. 345–352.
[21] G.T. Daich, Testing combinations of parameters made easy [software testing], in: Proceedings of the IEEE Systems Readiness Technology Conference (AUTOTESTCON 2003), 2003, pp. 379–384.
[22] Z. Ding, K. Zhang, J. Hu, A rigorous approach towards test case generation, Information Sciences 178 (21) (2008) 4057–4079.
[23] I.S. Dunietz, W.K. Ehrlich, B.D. Szablak, C.L. Mallows, A. Iannino, Applying design of experiments to software testing, in: Proceedings of the International Conference on Software Engineering (ICSE '97), ACM Press, New York, Boston, MA, 1997, pp. 205–215.
[24] M. Forbes, J. Lawrence, Y. Lei, R.N. Kacker, D.R. Kuhn, Refining the in-parameter-order strategy for constructing covering arrays, NIST Journal of Research 113 (5) (2008) 287–297.
[25] M. Grindal, J. Offutt, J. Mellin, Conflict Management when using combination strategies for software testing, in: Proceedings of 18th Australian Software Engineering Conference, Melbourne, Australia, 2007, pp. 255–264.
[26] A. Hartman, T. Klinger, L. Raskin, IBM Intelligent Test Case Handler. <http://www.alphaworks.ibm.com/tech/whitch> (last accessed 24.10.10.
[27] B. Hnich, S.D. Prestwich, E. Selensky, B.M. Smith, Constraint models for the covering test problem, Constraints 11 (2–3) (2006) 199–219.
[28] B. Jenkins, Jenny. <http://www.burtleburtle.net/bob/math/jenny.html> (last accessed 24.10.10.
[29] M.F.J. Klaib, K.Z. Zamli, N.A.M. Isa, M.I. Younis, R. Abdullah, G2Way – a backtracking strategy for pairwise test data generation, in: Proceedings of the 15th IEEE Asia–Pacific Software Engineering Conference, Beijing, China, 2008, pp. 463–470.
[30] R. Krishnan, S.M. Krishna, P.S. Nandhan, Combinatorial testing: learnings from our experience, ACM SIGSOFT Software Engineering Notes 32 (3) (2007) 1–8.
[31] D.R. Kuhn, Y. Lei, R. Kacker, Practical combinatorial testing: beyond pairwise, IT professional, IEEE Computer Society 10 (3) (2008) 19–23.
[32] D.R. Kuhn, V. Okun, Pseudo exhaustive testing for software, in: Proceeding of the 30th NASA/IEEE Software Engineering Workshop, 2006, pp. 25–27.
[33] R. Kuhn, Y. Lei, R. Kacker, Practical combinatorial testing: beyond pairwise, IEEE IT Professional 10 (3) (2008) 19–23.
[34] E. Lehmann, J. Wegener, Test case design by means of the CTE XL, in: Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000), Copenhagen, Denmark, 2000, pp. 1–10.
[35] Y. Lei, R. Kacker, D. Kuhn, V. Okun, J. Lawrence, IPOG/IPOD: efficient test generation for multi-way software testing, Journal of Software Testing, Verification, and Reliability 18 (2008) 125–148.
[36] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, IPOG: a general strategy for t-way software testing, in: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, Tucson, AZ, USA, 2007, pp. 549–556.
[37] Y. Lei, K.C. Tai, In-parameter-order: a test generation strategy for pairwise testing, in: Proceedings of the 3rd IEEE International High-Assurance Systems Engineering Symposium, Washington, DC, USA, 1998, pp. 254–261.
[38] Y. Ma, J. Offutt, Y. Kwon, MuJava: An Automated Class Mutation System, Journal of Software Testing, Verification and Reliability 15 (2) (2005) 97–133.
[39] R. Mandl, Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing, Communications of the ACM, vol. 28, New York, NY, USA, 1985, pp. 1054–1058.
[40] M.M. Mano, Digital Design, third ed., Prentice Hall Inc., 2002.
[41] C. Nie, B. Xu, L. Shi, G. Dong, Quality of software architectures and software quality-automatic test generation for N-way combinatorial testing, Lecture Notes in Computer Science 3712 (2005) 203–211.
[42] P.J. Schroeder, P. Faherty, B. Korel, Generating expected results for automated black-box testing, in: Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02), Edinburgh, UK, 2002, pp. 139–148.
[43] P.J. Schroeder, E. Kim, J. Arshem, P. Bolaki, Combining behavior and data modeling in automated test case generation, in: Proceedings of the 3rd International Conference on Quality Software (Qsic 2003), 2003, pp. 247–254.
[44] P.J. Schroeder, B. Korel, Black-box test reduction using input-output analysis, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2000), Portland, OR, USA, 2000, pp. 21–24.
[45] T. Shiba, T. Tsuchiya, T. Kikuno, Using artificial life techniques to generate test cases for combinatorial testing, in: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04), IEEE Computer Society, Hong Kong, 2004, pp. 72–77.
[46] N.J.A. Sloane, Home page. <http://www.research.att.com/~njas/> (last accessed 24.10.10).
[47] Smartware, SmartTest – Pairwise Testing tool. <http://www.smartwaretechnologies.com/smarttestprod.htm> (last accessed 24.10.10).
[48] B. Stevens, E. Mendelsohn, Efficient software testing protocols, in: Proceedings of the 8th IBM Centre for Advanced Studies Conference (CASCON '98), Toronto, ON, 1998, pp. 279–293.
[49] K.C. Tai, Y. Lei, A test generation strategy for pairwise testing, IEEE Transactions on Software Engineering 28 (1) (2002) 109–111.
[50] Y.-W. Tung, W.S. Aldiwan, Automating test case generation for the new generation mission software system, in: Proceedings of the Aerospace Conference, vol. 1, Big Sky, MT, USA, 2000, pp. 431–437.
[51] Vlad-Roubtsov, EMMA: a free Java code coverage tool. <http://emma.sourceforge.net/> (last accessed 24.10.10.
[52] A.W. Williams, TConfig. <http://www.site.uottawa.ca/~awilliam> (last accessed 24.10.10).
[53] A.W. Williams, Determination of test configurations for pair-wise interaction coverage, in: Proceedings of the 13th International Conference on Testing of Communicating Systems, 2000, pp. 57–74.
[54] A.W. Williams, Software Component Interaction Testing: Coverage Measurement and Generation of Configurations. Ph.D. Thesis, School of Information Technology and Engineering, University of Ottawa, 2002.
[55] A.W. Williams, R.L. Probert, A measure for component interaction test coverage, in: Proceedings of the ACSI/IEEE International Conference on Computer Systems and Applications (AICCSA 2001), IEEE Computer Society, Beirut, Lebanon, 2001, pp. 304–311.
[56] A.W. Williams, R.L. Probert, A practical strategy for testing pair-wise coverage of network interfaces, in: Proceedings of the 7th International Symposium on Software Reliability Engineering, 1996, pp. 246–254.
[57] J. Yan, J. Zhang, Backtracking algorithms and search heuristics to generate test suites for combinatorial testing, in: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06), vol. 1, Chicago, 2006, pp. 385–394.
[58] J. Yan, J. Zhang, A backtracking search tool for constructing combinatorial test suites, Journal of Systems and Software 81 (10) (2008) 1681–1693.

[59] M.I. Younis, K.Z. Zamli, Assessing combinatorial interaction strategy for reverse engineering of combinational circuits, in: Proceedings of the IEEE Symposium on Industrial Electronics and Applications (ISIEA2009), Kuala Lumpur, Malaysia, 2009, pp. 473–478.
[60] M.I. Younis, K.Z. Zamli, N.A.M. Isa, IRPS – an efficient test data generation strategy for pairwise testing, in: Proceedings of the 12th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES2008), Springer, Verlag, Zagreb, Croatia, 2008, pp. 493–500.
[61] M.I. Younis, K.Z. Zamli, M.F.J. Klaib, Z.C. Soh, S.C. Abdullah, N.A.M. Isa, Assessing IRPS as an efficient pairwise test data generation strategy, International Journal of Advanced Intelligence Paradigms (IJAIP) 2 (1) (2010) 90–104.
[62] K.Z. Zamli, N.A.M. Isa, M.F.J. Klaib, S.N. Azizan, A tool for automated test data generation (and execution) based on combinatorial approach, International Journal of Software Engineering and Its Applications 1 (1) (2007) 19–34.