A survey on Automated Combinatorial Testing for Software Tool (ACTS) with experimental revise based on T-way test generation

S. Bhuvana and Dr. M.V.Srinath
Sengamala Thayaar Educational Trust Women’s College, Mannargudi, Tamilnadu, India

ABSTRACT

A study on Automated Combinatorial Testing for Software (ACTS) tool with experimental revise based on T-way test generation. Purpose: software testing certifies the excellence of the software result. ACTS tool which is used for test generation based on constructing T-way Combinatorial test sets. Currently, it supports T-way test set generation with t ranging from 1 to 6. Combinatorial testing has been shown very effective in detecting faults that are caused by unexpected interactions between different contributing factors. The tool provides both command line and Graphical user interface (GUI), Pair wise testing has become a popular approach to software quality assurance because it often provides effective error detection at low cost. However, pair wise (2-way) coverage is not sufficient for assurance of mission-critical software. Combinatorial testing beyond pair wise is rarely used because good algorithms have not been available for complex combinations such as 3-way, 4-way, or more. In addition, significantly more tests are required for combinations beyond pair wise testing, and testers must determine expected results for each set of inputs. This article describes how new tools for automating the production of complete test cases covering up to 6-way combinations in various experiments.

Key words: Combinatorial, software, Graphical user interface, command line, test cases.

INTRODUCTION

Software systems are complex and can incur exponential numbers of possible tests. Testing is expensive and trade-offs often exist to optimize the use of resources. Many testers are familiar with the most basic form of Combinatorial testing all pairs or pair wise testing, in which all possible pairs of parameter values are covered by at least one test. Pair wise testing uses specially constructed test sets that guarantee testing every parameter value interacting with every other parameter value at least once. This is the core feature of ACTS. A system is specified by a set of parameters and their values. A test set is a T-way test set if it satisfies the following property: Given any t parameters (out of all the parameters) of a system, every combination of values of these t parameters is covered in at least one test in the test set.

Currently, ACTS supports T-way test set generation for $1 \leq t \leq 6$. Empirical studies show that $t$ being up to 6 is sufficient for most practical applications. A special form of 1-way testing, called base-choice testing is implemented in ACTS. Base-choice testing requires that every parameter value be covered at least once and in a test in which all the other values are base choices. Each parameter has one or more values designated as base choices. Informally, base choices are “more important” values, e.g., default values, or values that are used most often in operation. Several test generation algorithms are implemented in ACTS.
These algorithms include Iraq Policy and Operations Group (IPOG), IPOG-D, IPOG-F, and IPOG-F2. In general, IPOG, IPOG-F, and IPOG-F2 work best for systems of moderate size (less than 20 parameters and 10 values per parameter on average), while IPOG-D is preferred for larger systems. ACTS supports two test generation modes, namely, scratch and extend. The former allows a test set to be built from scratch, whereas the latter allows a test set to be built by extending an existing test set. In the extend mode, an existing test set can be a test set that is generated by ACTS, but is incomplete because of some newly added parameters and values, or because of a test set that is supplied by the user and imported into ACTS. Extending an existing test set can save earlier effort that has already been spent in the testing process. This paper is planned as pursues, first it exemplifies about ACTS software testing mechanism in various platforms. Then it describes about result and discussion. And finally explains about conclusion and future enhancement.

**ACTS tool method based on T-way test generation**

1. **New Symbolic Model Verifier model checker**

   New Symbolic Model Verifier (NuSMV) could easily have been constructed manually; the procedures introduced in this method can and have been used to produce tens of thousands of complete test cases in a few minutes once the SMV model has been defined for the system under test. The methods in this method still require human intervention and engineering judgment to define a formal model of the system under test and for determining appropriate abstractions and equivalence classes for input parameters. But by automating test generation we can provide much more thorough testing than is possible with most conventional methods. In addition, the testing has a sound empirical basis in the observation that software failures have been shown to be caused by the interaction of relatively few variables.

   By testing all variable interactions to an appropriate strength, we can provide stronger assurance for critical software. In Combinatorial test development is to determine what output should be produced by the system under test for each set of input parameter values, often referred to as the oracle problem in testing. The conventional approach to this problem is human intervention to design tests and assign expected results or, in some cases, to use a reference implementation that is known to be correct (for example, in checking conformance of various vendor products to a protocol standard). Because Combinatorial testing can require a large number of tests, an automated method is needed for determining the expected results for each set of input data. To solve this problem, we use the open-source NuSMV model checker (an enhanced version of the well-known SMV model checker).

2. **Sequence-Covering Arrays**

   In testing event-driven software, the critical condition for triggering failures often is whether or not a particular event has occurred prior to a second one, not necessarily if they are back to back. This situation reflects the fact that in many cases, a particular state must be reached before a particular failure can be triggered. For example, a failure might occur when connecting device A only if device B is already connected. The methods described in this paper were developed to address testing problems of this type, using Combinatorial methods to provide efficient testing. Sequence covering arrays, as defined here, ensure that any t events will be tested in every possible T-way order.

   For this problem we can define a sequence-covering array, which is a set of tests that ensure all T-way sequences of events have been tested. The t events in the sequence may be interleaved with others, but all permutations will be tested. For example, we may have a component of a factory automation system that uses certain devices interacting with a control
program. There are $6! = 720$ possible sequences for these six events, and the system should respond correctly and safely no matter the order in which they occur.

Operators may be instructed to use a particular order, but mistakes are inevitable, and should not result in injury to users or compromise the enterprise. Because setup, connections and operation of this component are manual, each test can take a considerable amount of time. It is not uncommon for system-level tests such as this to take hours to execute, monitor, and complete. We want to test this system as thoroughly as possible, but time and budget constraints do not allow for testing all possible sequences, so we will test all 3event sequences.

### 3. Input Parameter Modeling

The input space of a system must be modeled before Combinatorial testing can be applied to this system. The effectiveness of Combinatorial testing to a large extent depends on the quality of the input space model. For a large system that has too many features, we first use the divide-and-conquer strategy to divide the system into smaller systems. There are two general strategies. One is to divide the system vertically, e.g., based on features. That is, we could apply Combinatorial testing to one feature or a group of related features at a time. The other strategy is to divide the entire system into several subsystems, where each subsystem may be involved in multiple features. Next, we model the input space of each system. This modeling process consists of two major steps, Input Structure Modeling (ISM) and Input Parameter Modeling (IPM). ISM tries to capture the structural relationship among the different components in the input space.

After we have the input parameter model for each module, we generate test cases from the model using Combinatorial testing tools such as ACTS. These test cases are abstract test cases because the parameters and values in the model are abstract. Thus, it is necessary to derive concrete test cases from these abstract test cases before the actual testing can be performed. Note that an abstract test case typically represents a set of concrete test cases, from which one representative is typically selected to perform the actual testing.

### 4. Modified condition decision coverage

Modified condition decision coverage (MCDC) is a test strategy required by the US Federal Aviation Administration for life critical software. It is important to emphasize that MCDC is a coverage criterion for test suites normally applied to an entire program. In the analysis below, we analyze test sets with respect to one expression at a time. This analysis helps to explain why MCDC is effective. In practical applications, an MCDC test set applied to an entire program would be likely to have better Combinatorial coverage than the results for individual expressions.

**MCDC Lower Bounds**

An exhaustive analysis of all Boolean expressions of up to six variables has shown that of $n+ 1$ test can provide MCDC coverage for nearly all expressions, where $n$ is the number of variables involved in a Boolean expression. A natural question to ask is then, what level of Combinatorial coverage is provided by MCDC tests? This question can be addressed in two ways, by empirical data on Combinatorial coverage for MCDC test sets, and by evaluating the MCDC test construction using methods described in NISTIR 7878. Shown below is one result on minimum coverage from MCDC tests, followed by data on Combinatorial coverage of MCDC test sets for various numbers of variables.
In the unique-cause form of MCDC, it must be shown that the expression changes value as one variable value is switched while others remain fixed. For example, \((a+b) (c+d) = 1\) for the first test. Changing the value of \(a\) from 1 to 0 while other values remain fixed causes the expression to evaluate to 0. In this manner, one of the \(n\) variables is changed with each additional test, for a total of \(n+1\) test. Also relevant are methods and tools for extending an array to provide T-way coverage. This problem was considered in and several currently available covering array generators provide the capability, including ACTS.

5. Fault Detection Effectiveness

A T-way covering array can detect T-way faults; however they generally include other combinations beyond T-way as well. For example, a particular test set of all 5-way combinations is shown capable of detecting all seeded faults in a test program, despite the fact that it contains up to 9-way faults. This poster gives an overview of methods for estimating fault detection effectiveness of a test set based on Combinatorial coverage for a class of software. Assume deterministic software that computes the same output for a given set of input parameters and values.

Faults are also deterministic in that we assume a failure-triggering combination of input values will always produce a failure if it is present in the input. Under these assumptions, two factors in fault detection effectiveness are the fault distribution within the SUT, and Combinatorial coverage of the tests. If faults are detected if and only if a failure inducing combination appears in a test, then the probability of detection can be estimated within a certain range using the T-way coverage of tests and an approximate distribution of T-way faults.

6. ABAC Testing methodologies

How should an ABAC system be tested? Confirming that access will be granted for users with the right attributes is easy: we can simply read off the attribute conditions for each grant expression and verify that the access control system returns an authorization in each case. The number of such tests is linear in the number of grant conditions. However, it is much more difficult to ensure that no invalid combination of attributes will result in authorization. With \(n\) Boolean attributes or variables there are \(2^n\) possible combinations of attributes.

Covering array generation tools, such as ACTS, make it possible to include constraints that prevent the inclusion of variable combinations that meet criteria specified in a first order logic style syntax. For example, if we are testing applications that run on various combinations of operating systems and browsers, we may include a constraint such as ‘\(\text{OS} = \text{“Linux”} \Rightarrow \text{browser} \neq \text{“IE”}\)’. Constraints are typically used in situations such as this, where certain combinations do not occur in practice, and therefore should not be included in tests.

7. Heuristic t-wise Combination Strategies

Burroughs, Jain, and Erickson and Cohen, Dalal, Kajla, and Patton report on the use of a tool called Automatic Efficient Test Generator (AETG), which contains an algorithm for generating all pair-wise combinations. Cohen et al. later described a heuristic greedy algorithm for achieving t-wise coverage, where t is an arbitrary number. This algorithm was implemented in the AETG tool. It shows an algorithm that will generate a test suite to satisfy pair-wise coverage. A genetic algorithm was proposed by Shiba, Tsuchiya, and Kikuno. The algorithm was inspired by AETG, and adds one test case at a time to a test suite until enough test cases have been identified to satisfy the desired coverage criterion.
8. **T-way Testing Strategies**

Many combinations of input parameter, hardware configuration, software configuration and system platforms need to be tested and verified against the given requirements. Due to time and cost constraints, exhaustive testing is impractical and not cost efficient. As a result, many sampling strategies have been developed including that of equivalence partitioning, boundary value, cause and effect graphing as well as decision table mapping. While these traditional static and dynamic sampling strategies are useful for fault detection and prevention, they are not sufficiently effective to tackle bugs due to interaction.

Addressing this issue, many T-way strategies (where by t indicates the interaction strength) have been developed in the literature in the last 15 years including IPOG, MC-MIPOG, AETG, TCG, Density, TVG, PICT and many more. Although offering inherent advantages in terms of reducing the testing costs and efforts, T-way strategies are still hardly accepted by practitioners and software engineering community at large. Thus, it is important to study the current achievements and drawbacks in order to enhance their applicability.

9. **Computing Combinatorial Coverage**

Tools are available to compute the measures discussed in this article. Several covering array generators can compute total coverage, and NIST-developed tools that are freely available can compute a variety of additional measures, and produce the reports included in examples above. The tools also include embedded constraint solvers, making it possible to produce counts of covered combinations excluding those that are not possible physically, or should be excluded because of constraints among variables.

This is an essential feature for real-world use. It is also possible to generate additional tests to supplement those analyzed, to bring coverage up to any desired level. Combinatorial coverage provides valuable information for decision-makers because it measures the proportion of the input space that is covered relevant to testing. Because only a small number of variables are involved in failures, testing all settings of 4-way to 6-way combinations can provide strong assurance.

10. **Factor covering Combinatorial testing**

As described Combinatorial coverage measures the extent to which T-way combination settings have been included in a test set. Combinatorial coverage is useful in a variety of testing problems, but estimating the usefulness of T-way testing also requires some understanding of the complexity of test value combinations that are needed. For example, an application that has been tested and used extensively is likely to have few single-factor faults, because these would have already been detected. But a new, untested application may have a fairly high proportion of 1-way and 2-way faults. In this case, we may confine initial testing of the new application to 2-way or 3-way covering arrays, since we are likely to detect faults with even limited testing.

That is, the 2-way and 3-way arrays are likely to cover the combinations that trigger faults for this example, but less likely to cover the remaining faults in the extensively tested application. We can think of the relationship between fault distribution and Combinatorial coverage as fault coverage. Fault coverage is useful in gauging the effectiveness of a test set because it measures coverage of combinations related to fault detection, allowing testers to estimate if tests are sufficient or if more should be produced to cover relevant portions of the input space.
11. Tuple relationship tree Combinatorial testing

First, we construct a TRT for a failing test configuration. A TRT is a tree in which each node represents a tuple in the failing test configuration and each edge represents a direct parent and child relation from one node to another node. In fact, some tuples in a TRT can be easily determined to be faulty tuple or healthy tuple from the results of the executed test configurations. Thus we do not need to generate extra test configurations to analyze them. We can determine the type of a tuple using static review as follows:

First, the root tuple in a TRT must be a faulty tuple. This is because all the possible test configurations contain the root tuple is original configuration, which failed during testing. So it is a faulty tuple by definition. We mark the root tuple as a faulty tuple. Second, tuples that appear in one or more passed test configurations are healthy tuples by definition. We mark such tuples as healthy tuples. After the above steps, the TRT evolves as where dark nodes represent faulty tuples, grey nodes represent healthy tuples, and white nodes represent unknown tuples.

DISCUSSION AND CONSEQUENCES

From the table 1 we can conclude that ACTS tool have been used for various testing methods which results in better software testing performance are discussed. Using a sequence covering array for system testing described here made it possible to provide greater confidence that the system would function correctly regardless of possible dependencies among peripherals. T-way Testing Strategies has an advantage in terms of reducing the costs and effort.

Table -1 Survey on methods of ACTS tool from various approaches

<table>
<thead>
<tr>
<th>Authors Name</th>
<th>Technique</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>D. Richard Kuhn,</td>
<td>NuSMV model checker</td>
<td>This approach is to model the system as a simple state machine, then use NuSMV to evaluate the model and post-process the results into complete test cases.</td>
</tr>
<tr>
<td>James M. Higdon</td>
<td>Sequence-Covering Arrays</td>
<td>To address testing problems of this type, using Combinatorial methods to provide efficient testing. Sequence covering arrays, as defined here, ensure that any t events will be tested.</td>
</tr>
<tr>
<td>M. N. Borazjany</td>
<td>Input Parameter Modeling</td>
<td>The flat structure has no hierarchy, where components are equal peers. For example, a flat input structure can be used to model the command-line options of a program. The graph structure represents the composition relation between different components in a graph, where one component may be composed of several other components.</td>
</tr>
<tr>
<td>Raghu N. Kacker</td>
<td>Modified condition decision coverage</td>
<td>It is important to emphasize that MCDC is a coverage criterion for test suites normally applied to an entire program. In the analysis below, we analyze test sets with respect to one</td>
</tr>
<tr>
<td>Name</td>
<td>Topic</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Yu Lei</td>
<td>Fault detection effectiveness</td>
<td>This poster gives an overview of methods for estimating fault detection effectiveness of a test set based on Combinatorial coverage for a class of software.</td>
</tr>
<tr>
<td>Vincent Hu</td>
<td>ABAC Testing methodologies</td>
<td>Generation tools make it possible to include constraints that prevent the inclusion of variable combinations that meet criteria specified in a first order logic style syntax.</td>
</tr>
<tr>
<td>Mats Grindal</td>
<td>Heuristic t-wise Combination Strategies</td>
<td>Automatic Efficient Test Generator (AETG), which contains an algorithm for generating all pair-wise combinations. Cohen et al. later described a heuristic greedy algorithm for achieving t-wise coverage.</td>
</tr>
<tr>
<td>Rozmie R. Othman</td>
<td>T-way Testing Strategies</td>
<td>Although offering inherent advantages in terms of reducing the testing costs and efforts, T-way strategies are still hardly accepted by practitioners and software engineering community at large.</td>
</tr>
<tr>
<td>Dr. Richard Kuhn</td>
<td>Computing Combinatorial Coverage</td>
<td>It is also possible to generate additional tests to supplement those analyzed, to bring coverage up to any desired level.</td>
</tr>
<tr>
<td>Raghu N. Kacker</td>
<td>Factor covering Combinatorial testing</td>
<td>Fault coverage is useful in gauging the effectiveness of a test set because it measures coverage of combinations related to fault detection, allowing testers to estimate if tests are sufficient or if more should be produced to cover relevant portions of the input space.</td>
</tr>
<tr>
<td>Xintao Niu</td>
<td>Tuple relationship tree Combinatorial testing</td>
<td>A TRT is a tree in which each node represents a tuple in the failing test configuration and each edge represents a direct parent and child relation from one node to another node.</td>
</tr>
</tbody>
</table>

**CONCLUSION**

This paper presents a survey of some most prominent techniques of automated test data generation, including NuSMV model checker, MCDC, TRT, FCT, T-way Testing Strategies, Heuristic t-wise Combination Strategies and so on. The survey characterizes about To conquer the test case problem, our survey spotlight on ACTS tool with T-way testing method. This paper presents a case study on applying Combinatorial testing to test a Combinatorial test generation tool called ACTS. The Combinatorial testing is effective in terms of achieving high code coverage and fault detection. We plan to conduct similar studies of other real-world applications. The goal is to develop a set of guidelines, with significant examples, that can be used by practitioners to apply Combinatorial testing in practice. From the overall analysis it shows that ACTS tool has excellent methodologies for providing better quality software and to detect the fault in higher performance than others.
ACKNOWLEDGMENT

I would like to thank all the anonymous reviewers for giving a comprehensive review on our paper. Special thanks to Dr. M.V. Srinath, STET Women’s College, Mannargudi, Thiruvarur District, Tamil Nadu, for his valuable efforts and comments to improve the paper.

REFERENCES


