# A Requirements-Based Partition Testing Framework Using Particle Swarm Optimization Technique

by

Afshar Ganjali

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## Abstract

Modern society is increasingly dependent on the quality of software systems. Software failure can cause severe consequences, including loss of human life. There are various ways of fault prevention and detection that can be deployed in different stages of software development. Testing is the most widely used approach for ensuring software quality.

Requirements-Based Testing and Partition Testing are two of the widely used approaches for testing software systems. Although both of these techniques are mature and are addressed widely in the literature and despite the general agreement on both of these key techniques of functional testing, a combination of them lacks a systematic approach. In this thesis, we propose a framework along with a procedural process for testing a system using *Requirements-Based Partition Testing (RBPT)*. This framework helps testers to start from the requirements documents and follow a straightforward step by step process to generate the required test cases without loosing any required data. Although many steps of the process are manual, the framework can be used as a foundation for automating the whole test case generation process.

Another issue in testing a software product is the test case selection problem. Choosing appropriate test cases is an essential part of software testing that can lead to significant improvements in efficiency, as well as reduced costs of combinatorial testing. Unfortunately, the problem of finding minimum size test sets is NP-complete in general. Therefore, artificial intelligence-based search algorithms have been widely used for generating near-optimal solutions. In this thesis, we also propose a novel technique for test case generation using *Particle Swarm Optimization (PSO)*, an effective optimization tool which has emerged in the last decade. Empirical studies show that in some domains particle swarm optimization is equally well-suited or even better than some other techniques. At the same time, a particle swarm algorithm is much simpler, easier to implement, and has just a few parameters that the user needs to adjust. These properties make PSO an ideal technique for test case generation. In order to have a fair comparison of our newly proposed algorithm against existing techniques, we have designed and implemented a framework for automatic evaluation of these methods. Through experiments using our evaluation framework, we illustrate how this new test case generation technique can outperform other existing methodologies.

**Acknowledgements**

It has been much work during these last two years. But at the same time, it has been a lot of fun. A great part of this fun is due to the people I have been involved with, both in my research and teaching duties, and in my personal life.

I have to start with showing my gratitude to Professor Ladan Tahvildari, who has officially been my supervisor, but unofficially much more. I would like to appreciate her care and support which was not limited to my research work. Her insight and direction has enriched the content of this thesis.

I would like to acknowledge the support of Research In Motion (RIM) company. Without the assistance of RIM, my time at University of Waterloo would not have been as enriching. I should thank Gary Cort, Spencer Hill, Weining Liu and Julie Rastelli for the countless chats and their kind assistance throughout this work.

I would also like to thank Professor Dasiewics and Professor Kontogiannis for accepting to be members of my dissertation committee. I must thank them for taking the time out of their busy schedules to review my thesis and for their insightful comments and suggestions.

I should also thank all the members of Software Technologies and Applied Research (STAR) group for their moral support and valuable feedbacks.

Finally, would like to express my sincere gratitude to my family. My father, my mother, my brother Yashar and my sister-in-law Hamideh have always been a great support and have had the necessary understanding. This accomplishment would have been more difficult to achieve without their constant encouragement.

*To my father and mother*
*for their infinite love, understanding and support.*

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

The development of high quality software requires considerable investment in quality assurance resources. Software testing as an important part of this process is both expensive and time consuming. The whole testing process by various estimates can take as much as 20% to more than 50% of the total development budget of a software project and adds considerably to the length of the development cycle [3, 5, 9, 53]. A *tester* is normally responsible to bring out a right mix of business process knowledge, technical expertise and cutting edge technology for the company to be able to deliver flexible and scalable services to the customers.

The Institute of Electrical and Electronics Engineers (IEEE) defines *test* [28] as "a set of one or more test cases". The IEEE also defines *testing* as "the process of analyzing a software item to detect the differences between existing and required conditions and to evaluate the features of the software item". This definition makes testers responsible for both *verification* and *validation*. Verification simply answers the question "Does the system do what it is supposed to do?". For verifying the system, a tester should investigate the accuracy or correctness of the system according to its specification. This comparison of the system's response to what is expected is straightforward if there is a well-defined specification that states what the correct system response will be. This specification is called *test standard* in the literature [49]. It is virtually impossible to automate testing if there is no standard for the expected response and the automated test program can not make on the fly subjective judgements about the correctness of the outcome. Therefore having a test standard is essential for the automation of verification process. On the other hand, validation is the process by which we confirm that the system is designed to do things in the right way and it answers the question "Is what the system doing correct?". Validation is necessary to check for problems with the specification and

to demonstrate that the system is operational.

Different testing groups from different organizations do the verification and validation using various *test approaches* [49]. There is a plethora of testing methods and testing techniques, serving multiple purposes in different life cycle phases. Classified by purpose, software testing can be divided into *correctness testing*, *performance testing*, *reliability testing* and *security testing*. Classified by life cycle phase, software testing can be classified into the following categories: *requirements phase testing*, *design phase testing*, *program phase testing*, *installation phase testing*, *acceptance testing* and *maintenance testing*. By scope, software testing can be categorized as follows: *unit testing*, *component testing*, *integration testing*, and *system testing*.

From the long list of existing techniques and methods for testing, we will focus on three of them which are widely used in software companies for testing their products. These three approaches are: *Requirements-Based Testing (RBT)*, *Partition Testing* and *Combinatorial (Interaction) Testing*. In Section 1.1, we talk about two problems related to these methods. Section 1.2 briefly describes our proposed techniques for solving these problems. Finally, Section 1.3 describes the organization of the rest of this thesis.

## 1.1 Problem Description

In this thesis, we are going to address two different problems. The first problem is related to the integration of requirements-based testing process and partition testing. Both of these techniques are mature and have been studied widely in the literature. However, despite the general agreement on both of these key techniques of functional testing, there is no systematic approach for combining them. In this thesis, we aim at presenting a simple framework for testing groups. This framework can be used in various domains as a guideline for testing departments.

The second issue that we are going to focus on is the general problem of combinatorial test case generation using software specifications such as requirements. Applying the partitioning techniques for testing, we come up with a list of *variables (parameters)*, generated from the product specifications and a set of *values* for each variable. We know that a common source of system faults is the unexpected interaction between system components [58]. Therefore, for reducing the risk of interaction problems we should test a large number of possible test configurations. A *test configuration* can simply be defined as a combination of the different values of

the variables in the system. Let us consider $k$ independent variables in the system under test. Here, independence of variables means that the selection of a particular value for one variable does not effect the selection of any other values for other variables. Now let us assume variable $i$ has $n_i$ possible values, which are enumerated as $1 \dots n_i$. A test configuration consists of a selection of values for each parameter and hence can be indicated by a $k$-tuple. Since each test configuration ends up as a different test case and each test case requires some time to be executed and investigated by testers, the number of test configurations is the major cost factor in the testing process. Clearly, the number of potential test configurations which is equal to $\Pi_1^k n_i$ grows exponentially. Testing all these possible configurations called *exhaustive testing* is almost impossible in practice due to the time and money constraints. The solution is to reduce the required test effort by putting a limit on the required coverage of the possible value combinations and reducing the number of test configurations.

In the literature, various *coverage criteria* are defined which can be used for limiting the number of test configurations. One of the most well-known coverage criteria is pairwise testing. In pairwise testing, all combinations of the values of any two variables should be covered by at least one test case [52]. Based on the observation that most faults are caused by interactions of at most two factors, empirical results show that pairwise testing is practical and effective [7, 10, 15, 32]. Independent of which coverage criteria gets used, the goal of the combinatorial test case generation is to reach the coverage goal using the minimum number of test configurations.

## 1.2 Thesis Contribution

The major contribution of this thesis is to address the two problems described in Section 1.1. For the first problem, we propose a novel framework which combines the two mature techniques of requirements-based testing and partition testing into one unified technique. We also define a procedural process for testing a system using this new "Requirements-Based Partition Testing" (RBPT) framework. For the second problem, we introduce a method for combinatorial test case generation applying *Particle Swarm Optimization (PSO)* technique. The following list describes our contributions in more details.

- Proposing a layered framework for requirements-based partition testing.

- Using particle swarm optimization for effective combinatorial test case generation.
- Proposing a simple boundary condition, called "Cyclic Walls", for PSO that can be used for solving the problems which have a finite search space.
- Developing a framework for automatic comparison of different test case generation algorithms.
- Introducing a new test case generation metric for assessing the effectiveness of the test suites generated by different combination strategies.
- Illustrating through empirical experiments that PSO can be as effective as other existing techniques for combinatorial test case generation.

## 1.3 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 presents a survey of the related works, and gives an overview on the required background material. In the first section, it introduces different software testing techniques and gives a brief definition for each of them. Section 2.2 reviews requirements-based software testing. Section 2.3 presents how partition testing works in general, and then focuses on category partition method which will be used later. Finally, the last section describes the combinatorial test case generation problem as an optimization problem, and presents a survey of existing combination strategies in the literature.
- Chapter 3 is about our first problem: integration of requirements-based testing and partition testing. The first section of the chapter presents the layered structure of the proposed framework, and the second section puts all the layers of the framework together and defines a test case generation process which traverses all the layers of the framework one by one for generating a complete test suite. The main goal of this process is to formalize and automate the required activities for testing a newly developed system.
- Chapter 4 proposes a novel technique for test case generation, using *Particle Swarm Optimization (PSO)*, an effective optimization tool which has emerged in the last decade. For comparing the results from PSO combination strategy

with other existing techniques in the literature, a benchmark framework is presented in Section 4.4.1 of this thesis. Section 4.4.1 also proposes a new effectiveness measure which produces better and more precise assessments from the output of such algorithms. Finally, in Section 4.4.2 the proposed benchmarking framework and effectiveness measure are used for executing some experiments. Results show that PSO combination strategy can be as effective as other existing algorithms.

- Chapter 5 reviews the thesis contributions, and outlines future directions.

# Chapter 2

# Concepts and Related Works

Modern society is increasingly dependent on the quality of software systems. Software failure can cause severe consequences, including loss of human life at extreme. There are various ways of fault prevention and detection that can be deployed in different stages of software development. Testing is the most widely used approach for ensuring software quality.

In this chapter we will have an overview on existing software testing techniques. The first section of the chapter briefly describes a wide range of testing methods. Then, in the second and third sections, we focus on *requirements-based testing* and *partition testing* respectively and explain what these techniques are and how they are important in the testing of software systems. Finally in Section 2.4, we show that the problem of combinatorial test case generation is an optimization problem and review some of the existing methods for handling this problem in practical cases.

## 2.1 Software Testing Techniques

Software testing is the process of executing a program or system with the intent of finding errors and defects [37]. It also involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results [25]. Unlike other physical processes, software can fail in many unexpected ways. Detecting all of the different failure modes for software is generally infeasible. There are an abundance of software testing techniques in the literature such as black or white box testing, static or dynamic testing, partition testing, requirements-based testing, mutation testing. Most of these techniques and testing

methods are not very different from 20 years ago. Although there are many tools and techniques available to use, an efficient testing technique also requires a tester's creativity, experience and intuition. Here in this section, we will have a brief review on some of the well-known existing software testing methods.

- **Static Testing Vs. Dynamic Testing:** There are many approaches and techniques that can be used in software testing. Reviews, walkthroughs or inspections are some of these methods that can be considered as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Static testing is not essential and can be omitted, even though it is very useful for avoiding a large group of defects. Dynamic testing takes place when programs begin to be used for the first time - which is normally considered the beginning of the testing stage. This may actually begin before the program is 100% complete in order to test particular sections of code (modules or discrete functions).

- **Black Box Testing Vs. White Box Testing:** The black box testing approach is a testing method in which test data are derived from the specified functional requirements without any knowledge of the final program structure [42]. It is also referred to as *requirements-based testing* [25]. Since only the functionality of the software module is of concern, black box testing is also referred to as functional testing. In this approach tester treats the software under test as a black box. The assumption is that the tester has access only to the inputs, outputs and the specification, and the functionality is determined by observing the outputs to corresponding inputs. For testing, various inputs are exercised and the outputs are compared against specification to validate the correctness. All test cases are derived from the specification documents and no implementation details of the code are considered.

  On the other hand in white box testing, contrary to black box testing, software is viewed as a white box as the structure and flow of the software under test are visible to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure. White box testing may also be referred to as *glass-box testing* or *design-based testing*.

  There are many techniques available in white box testing. Some of these techniques try to test the software exhaustively according to different coverage criteria, for example by executing each line of code at least once (statement

coverage), traversing every branch statements (branch coverage), or covering all the possible combinations of true and false condition predicates (multiple condition coverage). Control flow testing, loop testing, and data flow testing are some other examples of white box methods that map the corresponding flow structure of the software into a directed graph. Test cases are carefully selected based on the criterion that all the nodes or paths are covered or traversed at least once. By doing so we may also discover unnecessary "dead" code which is of no use, or never gets executed and can not be discovered by functional testing.

We should note that many testing strategies may not be easily classified into black box testing or white box testing. One reason is that all the testing techniques will need some knowledge of the specification of the software under test. Another reason is that the idea of specification itself is broad and it may contain any requirement including the structure, programming language, and programming style as part of the specification content.

- **Unit Testing:** In computer programming, unit testing is a procedure used to validate that individual units of source code are working properly. A *unit* is the smallest testable part of an application. In procedural programming a unit may be an individual program, function, procedure, etc., while in object-oriented programming, the smallest unit is a method; which may belong to a base/super class, abstract class or derived/child class. Unit testing is typically done by developers and not by software testers or end-users.

- **Mutation Testing:** In mutation testing, the original program code is changed and many mutated programs are created, each containing one fault. Each faulty version of the program is called a *mutant*. Test data are selected based on the effectiveness of failing the mutants. The more mutants a test case can kill, the better the test case is considered. The problem with mutation testing is that it is computationally too expensive.

- **Random Testing:** In random testing, the test case selection process is very simple and straightforward: they are randomly chosen. Study in [16] indicates that under certain very restrictive conditions, random testing can be as effective as partitioning testing. They showed consistent small differences in effectiveness between partition testing methods and random testing. These results were interpreted in favor of random testing since it is generally less work to construct test cases in random testing since partitions do not have

to be constructed. But later investigations in [23] concluded that the Duran/Ntafos model in [16] was unrealistic. One reason was that the overall failure probability was too high. Some other studies in [21] followed up these results and showed theoretically that partition testing is consistently more effective than random testing under realistic assumptions. More recent results have been produced that favor partition testing over random testing in practical cases [43]. Effectively combining random testing with other testing techniques may yield more powerful and cost effective testing strategies.

- **Combinatorial Testing and Pairwise Testing:** If we partition the input domain of a software system into a set of variables each having a set of possible values, combinatorial testing method requires that for any given $t > 1$, all $t$-wise combinations of the values of those variables be tested by at least one test case. Pairwise testing is a special case of combinatorial testing, where $t = 2$. In pairwise testing, given any pair of input variables (parameters) of a system, every combination of valid values of the two variables must be covered by at least one test.

  Exhaustive testing is impractical due to resource constraints. It is not practical to cover all the parameter interactions. We need a good trade-off between test effort and test coverage. Empirical studies show that many faults are caused by the interactions between two variables (parameters). Hence, pairwise testing can be used as an effective testing method to make a balance between test effort and test coverage. Studies in [6, 13] argue that the testing of all pairwise interactions in a software system finds a large percentage of the existing faults and provide empirical results to show that this type of test coverage is effective.

- **Performance Testing:** Not all software systems have specifications on performance explicitly. However, every system will have implicit performance requirements. The software should not take infinite time or infinite resource to execute. "Performance bugs" sometimes are used to refer to those design problems in software that cause the system performance to degrade. Performance has always been a great concern. Performance evaluation of a software system usually includes resource usage, throughput, stimulus-response time and queue lengths (the average or maximum number of tasks waiting to be serviced by selected resources). Typical resources that need to be considered include network bandwidth requirements, CPU cycles, disk space, disk access operations, and memory usage [50]. The goal of performance testing

can be performance bottleneck identification, performance comparison and evaluation, etc. The typical method of doing performance testing is using a benchmark designed to be representative of the typical system usage [54].

- **Reliability Testing:** Software reliability refers to the probability of failure free operation of a system. It is related to many aspects of software, including the testing process. Directly estimating software reliability by quantifying its related factors can be difficult. Testing is an effective sampling method to measure software reliability. Software testing (usually black box testing) can be used to obtain failure data, and an estimation model can be further used to analyze the data to estimate the present reliability and predict future reliability. Therefore, based on the estimation, the developers can decide whether to release the software, and the users can decide whether to adopt and use the software. Risk of using software can also be assessed based on the reliability information.

  Hamlet in [22] advocates that the primary goal of testing should be to measure the dependability of tested software. There is agreement on the intuitive meaning of *dependable software*: it does not fail in unexpected or catastrophic ways [22]. *Robustness testing* and *stress testing* are variances of reliability testing based on this simple criterion. IEEE defines the robustness of a software component as the degree to which it can function correctly in the presence of exceptional inputs or stressful environmental conditions [1]. Robustness testing differs with correctness testing in the sense that the functional correctness of the software is not of concern. It only watches for robustness problems such as machine crashes, process hangs or abnormal termination. *Stress testing*, or *load testing*, is often used to test the whole system rather than the software alone. In such tests the software or system are exercised with or beyond the specified limits. Typical stress includes resource exhaustion, bursts of activities, and sustained high loads.

- **Security Testing:** Software quality, reliability and security are tightly coupled. Flaws in software can be exploited by intruders to open security holes. With the development of the Internet, software security problems are becoming even more severe. Many critical software applications and services have integrated security measures against malicious attacks. The purpose of security testing of these systems include identifying and removing software flaws that may potentially lead to security violations, and validating the effectiveness of security measures. Simulated security attacks can be performed to

find vulnerabilities.

## 2.2 Requirements-Based Software Testing

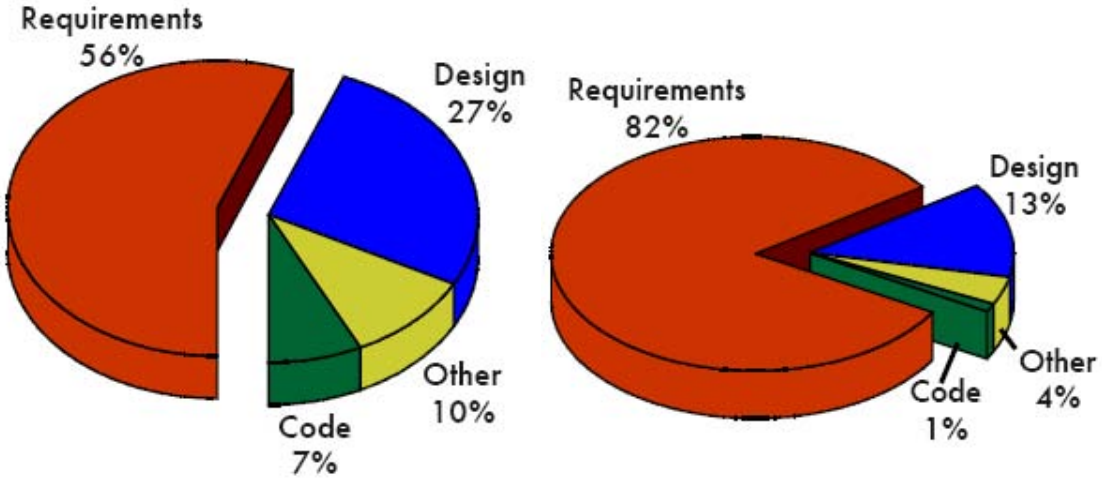
One of the first sources of information for the testers should be requirements. We know that testing the software is an integral part of building a system. However, if the software is based on inaccurate requirements, then even with a well-written code, the software will be unsatisfactory. The specification must contain all the requirements that are to be solved by our system. The specification should also explicitly specify everything our system must do and the conditions under which it must perform. In order for the requirements to be considered testable, the requirements ideally should have all of the following characteristics [36]:

- **Deterministic:** Given an initial system state and a set of inputs, one must be able to predict exactly what the outputs will be.
- **Unambiguous and Readable:** All project members must get the same meaning from the requirements; otherwise those requirements are ambiguous.
- **Correct:** The relationships between causes and effects must be described correctly.
- **Complete:** All requirements should be included. No omissions are allowed.
- **Non-redundant:** The requirements should provide a non-redundant set of functions and events.
- **Placed under change control:** Requirements, like all other deliverables of a project, should be placed under change control.
- **Traceable:** Requirements must be traceable to each other, to the objectives, to the design, to the test cases and to the code.
- **Written in a consistent style:** Requirements should be written in a consistent style to make them easier to understand.
- **Explicit:** Requirements must never be implied.
- **Logically consistent:** There should be no logic errors in the relationships between causes and effects.

- **Reusable:** Good requirements can be reused on future projects.
- **Terse:** Requirements should be written in a brief manner, with as few words as possible.
- **Annotated for criticality:** Not all requirements are critical. Each requirement should note the degree of impact a defect in it would have on production. In this way, the priority of each requirement can be determined, and the proper amount of emphasis placed on developing and testing each requirement.
- **Feasible:** If the software design is not capable of delivering the requirements, then the requirements are not feasible.

The *Requirements-Based Testing (RBT)* process [36], which is one of the most widely used software testing techniques, is based on these characteristics of good requirements and addresses two major issues: first, validating that the requirements are correct, complete, unambiguous, and logically consistent; and second, designing a necessary and sufficient (from a black box perspective) set of test cases from those requirements to ensure that the design and code fully meet those requirements. One of the most important issues to be overcome in the process is to reduce the immensely large number of potential tests down to a reasonable size test set.

According to recent studies, the majority of defects have their root cause in poorly defined requirements [35] (see Figure 2.1). On the other hand, the cost of fixing an error is cheaper the earlier it is found. If a defect was introduced while coding, you just fix the code and recompile. However, if a defect has its roots in poor requirements and is not discovered until integration testing then you must redo the requirements, design, code, the tests, the user documentation, and the training materials. All this extra work can send projects over budget and over schedule. If a defect introduced during the requirements phase is not found until integration testing or production, it will cost hundreds or even thousands of times more than the case where it is found and fixed in the requirements phase. Therefore, the overall RBT strategy is to integrate testing throughout the development life cycle and focus on the quality of the requirements specification. Testing starts at the beginning of the project, not at the end of the coding and we apply tests to assure the quality of the requirements. This leads to early defect detection which has been shown to be much less expensive than finding defects during integration testing or later. The RBT process also has a focus on defect prevention, not just defect detection and




(a) Distribution of Bugs (b) Distribution of Effort to Fix Bugs

Figure 2.1: Distribution of Bugs and Required Effort for Fixing Them [35]

hence it minimizes expensive rework by minimizing requirements related defects that could have been discovered, or prevented, early in the project's life.

One of the most challenging aspects of the requirements-based testing is communicating with the people who are supplying the requirements. If we have a consistent way of recording requirements, we can make it possible for the stakeholders to participate in the requirements process. This way, as soon as a requirement becomes visible we can start testing it and ask the stakeholders detailed questions. We can apply a variety of tests to ensure that each requirement is relevant, and that everyone has the same understanding of its meaning. We can ask the stakeholders to define the relative value of requirements. We can also define a quality measure for each requirement, and we can use that quality measure to test the eventual solutions.

Prioritizing the requirements [4, 55] is another important issue which should be considered in RBT process. If we can establish the relative priorities of the requirements, then it helps greatly in establishing the rank of the tests that are designed to verify the requirements and the amount of test coverage that will be provided. Ranking provides a valuable tool for designers and developers to pass on their knowledge and assumptions of the relative importance of various features in the system.

## 2.3 Partition Testing Techniques

The term *partition testing* refers to a very general family of testing strategies. The primary characteristic of these strategies is that the program's input domain is divided into subsets, with the tester selecting one or more element from each sub-domain. In the testing literature, it is common not to restrict the term *partition* to the formal mathematical meaning of a division into disjoint subsets, which together span the space being considered. Instead, testers generally use it in the more informal sense to refer to a division into (possibly overlapping) subsets of the domain. The goal of such a partitioning is to make the division in such a way that when the tester selects test cases based on the subsets, the resulting test set is a good representation of the entire domain. A partition can be defined using all the information about a program. It can be based on requirements or specifications (one form of black box testing), on features of the code (structural testing), even on the process by which the software was developed, or on the suspicions and fears of a programmer [23]. Ideally, the partitioning divides the domain into sub-domains with the property that within each sub-domain, either the program produces the correct answer for every element or the program produces an incorrect answer for every element. Such a sub-domain is called *revealing* [56] or *homogeneous* [23]. If a partition's sub-domains are revealing, one need only randomly select an element from each subset and run the program on that test case in order to determine program faults. Informal guidelines for creating such a partition and theoretical properties are discussed in [44, 56]. In practice, it is common for the division of the input domain to be into non-disjoint subsets, and it is extremely unusual for the sub-domains to be truly revealing.

Partition testing has two extreme cases: exhaustive testing and random testing. Exhaustive testing requires that every element of the input domain be explicitly tested. As a partition testing technique, therefore, exhaustive testing simply corresponds to the division of the input domain into single element sub-domains. The other extreme case is random testing. In this case, the partition consists of one class, namely, the entire domain. Random testing can, therefore, be viewed as a degenerate form of partition testing.

The strength of partition testing is its ability to use any and all available information during the software development life cycle, and to examine information in combinations that may not have been thought of during development. Intuitively, the source of program bugs and defects is some unlikely combination of requirements, design, and programmer inattention. By including these factors in

the sub-domain definition, we can be confident that nothing is missed in testing. Good sub-domains are defined and refined throughout development as information arises.

The *category partition method* [39] is one of the most effective partitioning techniques. This method provides a way to quickly translate a design specification to a test specification. It guides the tester to create functional test cases by decomposing functional specifications into test specifications for major functions of the software. It identifies those elements that influence the functionality and generates test cases by methodically varying the elements over all values of interest. Thus, it can be considered a black box integration technique. The category partition method provides a general systematic procedure for creating test specifications. The testers main job is to develop *categories*, which are defined to be the major characteristics of the input domain of the function under test, and to partition each category into equivalence classes of inputs called *choices*. By definition, choices in each category must be disjoint, and together the choices in each category must cover the input domain. The steps below show the method in brief:

1. Analyze the specification to identify the individual functional units that can be tested separately.
2. Identify the input domain, that is the categories (also called parameters or variables) that affect the behavior of the function.
3. Partition each category into choices (also called values).
4. Specify combinations of choices to be tested.
5. Convert the test frames produced by the tool into test cases, and organizes the test cases into test scripts.

Using this method, the obvious tests could be enumerated quickly and completely, leaving more time to think about more subtle issues. One of the first benefits of this method is that we can achieve a fairly uniform coverage across a large problem space.

## 2.4 Combinatorial Test Case Generation as an Optimization Problem

Many problems of both practical and theoretical importance can be expressed as a problem of choosing a "best" configuration or set of parameters to achieve some goal. In the domains of *computer science* and *operations research* a hierarchy of such problems has emerged, together with a corresponding collection of techniques for their solution. The most general problem of such kind is the general *nonlinear programming problem*:

$$\begin{array}{lll} \text{Find x to} & & \\ \text{minimize} & f(x) & \\ \text{subject to} & g_i(x) \geq 0 & i = 1, \ldots, m \\ & h_j(x) = 0 & j = 1, \ldots, p \end{array}$$

where $f$, $g_i$ and $h_j$ are general function of the parameter $x \in R^n$. Defining some specific conditions on the functions $f$, $g_i$ and $h_i$ results in different sets of problems. The techniques for solving such problems in different sets are studied separately in different branches of mathematics, operations research and computer science. For example when $f$ is convex, $g_i$ concave and $h_j$ linear, we have what is called a *convex programming problem*. Inequalities involving concave functions define a convex feasible region for the problem and the problem concerns the minimization of a convex function on a convex set. The most well-known property of this problem is that if a local minimum exists, then it is a global minimum [41].

In another situation when $f$ and all the $g_i$ and $h_j$ functions are linear, we come up with the *linear programming problem*. Linear programming is an important field of optimization and many practical problems in operations research and engineering fields can be expressed as linear programming problems. Any problem in this class reduces to the selection of a solution from among a finite set of possible solutions. The problem is what we can call *combinatorial*. The finite set of candidate solutions is the set of vertices of the convex polytope defined by the linear constraints. The widely used *simplex algorithm* [12] finds an optimal solution to a linear programming problem in a finite number of steps, though it is not a polynomial time algorithm. This algorithms is based on the idea of improving the cost by moving from vertex to vertex of the polytope.

Another set of optimization problems is the *integer linear programs*. These come about when we consider linear programs and try to find the best solution with the restriction that it should have integer valued coordinates. The general integer linear

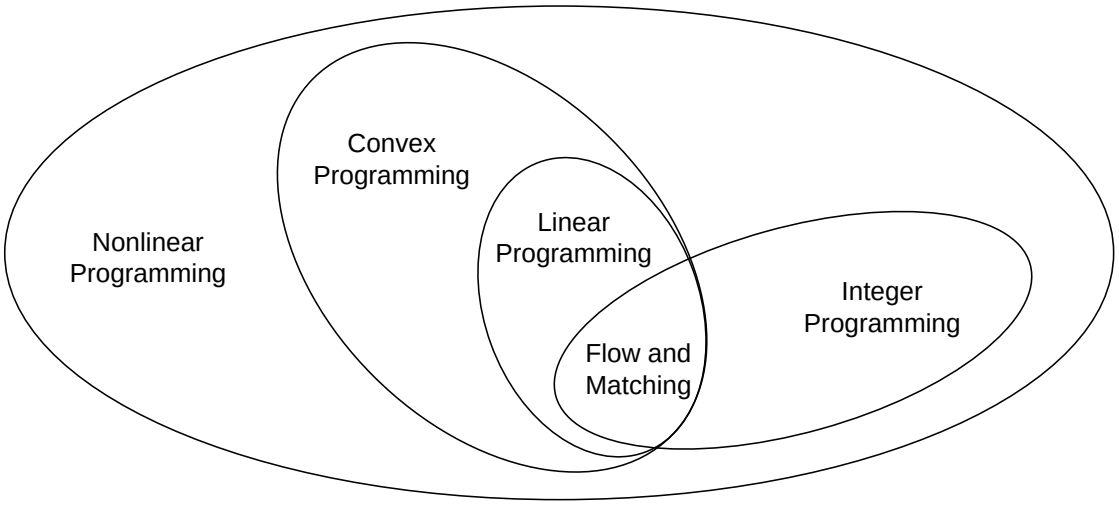



Figure 2.2: Nesting of the Optimization Problem Categories [41]

programming problem is itself NP-complete.

*Flow and matching problems* which are special cases of both linear programs and integer linear programs are another subset of optimization problems that can be solved much more efficiently than even general linear programs. Figure 2.2 indicates the nesting of the problems mentioned so far.

In general an *optimization problem* can be defined as a pair $(F, c)$ where $F$ is any set, the domain of feasible points (solutions) and $c$ is the cost function which is a mapping:

$$c : F \rightarrow R$$

The problem is to find an $f \in F$ for which:

$$c(f) \leq c(y) \text{ for all } y \in F$$

Such a point $f$ is called a *globally optimal solution* or simply an *optimal* solution.

Considering the definition of combinatorial test case generation problem from Section 1.1, we can say that this problem is an instance of optimization problem. The set $F$ will be the set of all possible test suites that we may use for testing the software under test. The $c$ will refer to a combination of two functions. First the cost of executing the test cases included in $F$ and second the coverage gained from execution of those test cases. A globally optimal solution will be a test suite which

results in higher coverage of value interactions by including a minimum number of test cases.

Optimization problems seem to divide into two categories: those with *continuous* variables and those with *discrete variables*. The latter is called *combinatorial*. In the continuous problems, we are generally looking for a set of real numbers or even a function whereas in the combinatorial problems, we are looking for an object from a finite or possibly countably infinite set. The test case generation problem because of its discrete nature is a combinatorial optimization problem.

Now it is clear the we can reduce the test case generation problem to an optimization problem. Hence, we may try to use the mature techniques in the research operation and computer science fields to tackle the problem. However we do not have that much chance because studies in [33] shows that combinatorial test case generation problem is NP-complete. Therefore, we get limited to the approaches that are practical for solving the NP-complete problems of moderate size, namely: approximation, enumerative techniques, and local search methods.

In the next section, we review some of existing test case generation techniques which try to find near-optimal solutions for combinatorial test case generation problem.

## 2.5 Overview of Existing Combination Strategies

*Combination strategies* are a class of test case selection methods where test cases are identified by choosing interesting values, and then combining those values of test object parameters. The combinations are selected based on some combinatorial strategy.

There are various combination strategies introduced in the literature. They can get classified into different categories according to their specifications. For example we have a group of deterministic combination strategies like *Orthogonal Arrays (OA)* [11, 57] which enable us to anticipate the number of test frames. Contrary to this, non-deterministic algorithms like *Genetic Algorithms (GA)* may result in different test frames or even different number of test frames in each execution. Iterative combination strategies can be stopped by reaching the required number of test cases or the expected coverage. On the other hand instant algorithms such as OA generate the whole set of test frames together and can not be stopped in the middle. Some of the combination strategies are designed to generate the required

test frames for a fixed predetermined amount of coverage such as OA which is used for reaching pairwise coverage. On the other hand some other algorithms such as GA are flexible and can be configured to generate the desired extent of coverage over the values. These characteristics of different algorithms should be considered for selecting a suitable algorithm in different cases.

In this section, we briefly introduce some of the well-known combination strategies that are used for test case generation.

### 2.5.1 Ant Colony Algorithm

One of the combination strategies for generating test cases is based on *Ant Colony Algorithm (ACA)* [48]. ACA was first used to solve the traveling salesman problem (TSP) [14], but has been successfully used to solve other combinatorial problems. An ACA was inspired by the behavior of natural ant colonies in finding paths from the colony to food. The concept of an ACA is to mimic this behavior with simulated ants crawling the graph representing the possible solutions for the problem. Each ant represents one candidate solution.

An ACA algorithm is based on a set of assumptions. The first assumption is that each path from a starting point to an ending point in the solutions graph is associated with a candidate solution to a given problem. The second idea in the algorithm comes from the concept of pheromone deposition by ants. When an ant reaches the ending point, the amount of pheromone deposited on each edge of the path followed by this ant is proportional to the quality of the corresponding candidate solution. The third assumption is that when an ant has to choose among different edges at a given point, the edge with a larger amount of pheromone is chosen with higher probability. As a result, the ants eventually converge to a short path, hopefully the optimum or a near-optimal solution to the target problem. Algorithm 1 shows the outline of the ACA procedure for test case generation.

### 2.5.2 Genetic Algorithms

*Genetic Algorithm (GA)* [48] mimics the evolution of simple, single celled organisms. It is based on the concept that the candidate solution created by swapping two good candidates is also good. GAs have been widely used in solving problems ranging from optimizations to machine learning.

**Algorithm 1** : Ant Colony Algorithm Outline

```
1: Let UC be a set of all tuples of parameter values that are not yet covered by
   the selected test frames;
2: while UC is not empty do
3:    Place m ants at the starting point (initialize the population of candidates);
4:    for a specified number of iterations do
5:       for all ant k do
6:          Generate a candidate test frame TF_k;
7:          Evaluate TF_k;
8:          Lay pheromone;
9:       end for
10:      Apply pheromone evaporation;
11:      Each ant leaves more pheromone on the traversed path;
12:   end for
13:   Let TF be the best test frame found;
14:   Add TF to the test set;
15:   Remove those tuples in UC that are covered by TF;
16: end while
```

In a GA each candidate solution must be encoded as a chromosome which is usually a string of values; by evolving the population of chromosomes, a good individual (solution) is eventually obtained. In test case generation problem, however, a test frame can be directly treated as a chromosome because a test frame is simply a string of values. The fitness function is used to estimate the goodness of a candidate solution. We define the fitness function for a test frame as the number of new $t$-wise combinations that are not covered by the given test set but are covered by that test frame. At the initialization, the initial population of candidate test frames is generated at random. After the initialization, the GA goes into the evaluation loop. The GA continues to evolve until the stopping conditions are met. At each generation, the best chromosomes in the population are kept and survive to the next generation intact. The remaining test cases in the next population are created by selecting a set pf *parent* chromosomes and applying an appropriate type of *crossover* and *mutation* on those parents. Algorithm 2 shows the outline of the GA method for test case generation.

### 2.5.3 Simulated Annealing

The *Simulated Annealing (SA)* algorithm is modeled after the effect of a slow cooling process on the molecules of a metallic substance [40]. Just as cooling brings these molecules to an optimal rest energy, this algorithm slowly converges the state

**Algorithm 2** : Genetic Algorithms Outline

1: Let $UC$ be a set of all tuples of parameter values that are not yet covered by the selected test frames;
2: **while** $UC$ is not empty **do**
3: Create an initial population $P$ consisting of $m$ candidates;
4: **for** a specified number of iterations **do**
5: Identify *Elite* individuals for survival consisting of $\sigma$ best individuals from $P$;
6: Apply selection to individuals in $P$ to create $Pmating$, consisting of $(m-\sigma)$ individuals;
7: Crossover $Pmating$;
8: Mutate $Pmating$;
9: $P = Elite + Pmating$;
10: **end for**
11: Let $TF$ be the best test frame found;
12: Add $TF$ to the test set;
13: Remove those tuples in $UC$ that are covered by $TF$;
14: **end while**

being examined toward an optimal state. Prior to running, an initial positive temperature $T_{initial}$ and a decimal decrement factor $\alpha$ lying strictly between 0 and 1 must be provided as input. The algorithm's simulated cooling schedule is determined entirely by these two parameters, as $T_{initial}$ is multiplied successively by $\alpha$ after each pass through the main loop.

At each step, the algorithm randomly selects one candidate solution from the neighbourhood of the current state and evaluates its fitness. The neighborhood can be defined in various ways according to the specifications of the problem domain. If the neighbouring solution is more fit than the current solution, then the neighbouring array becomes the new candidate test frame. However, should the selected neighbouring solution be less fit than the current solution, the SA heuristic is employed. This heuristic is the main feature of the SA algorithm and guarantees that at each step, there is a non-zero probability of moving to a state which is less fit than the current state. SA can accept worse states according to some probability which is called the *acceptance probability*. The acceptance probability is dependent on the current temperature and the cost difference of the two candidate solutions. Normally as the temperature decreases, the probability of accepting a worse state decreases. Algorithm 3 indicates the high level outline of the SA algorithm for test case generation.

**Algorithm 3** : Simulated Annealing Outline

```
1: Let UC be a set of all tuples of parameter values that are not yet covered by
   the selected test frames;
2: Let T_initial, T and T_final be the initial, current and final temperatures respec-
   tively;
3: while UC is not empty do
4:    Generate a random initial solution;
5:    T = T_initial;
6:    while T > T_final do
7:       for a specified number of iterations do
8:          Generate a candidate solution which is neighbor to the current one;
9:          if new solution is better than the current one then
10:            Accept the move;
11:         else
12:            Probabilistically accept the move;
13:         end if
14:      end for
15:      Update temperature: T = T * α;
16:   end while
17:   Let TF be the best test frame found;
18:   Add TF to the test set;
19:   Remove those tuples in UC that are covered by TF;
20: end while
```

### 2.5.4 Tabu Search

The *Tabu Search (TS)* algorithm [19, 51] is similar to SA algorithm in this way that both of the algorithms allow the selection of a new state which is less fit than the current state. In TS unlike SA, each neighbourhood search is exhaustive, rather than random, ensuring that the state chosen at each step is the best neighbouring option possible. The TS algorithm has only one input parameter, called the *tabu length* and denoted by $L$. At each step of the search, the TS algorithm evaluates the fitness of every solution in the neighbourhood of the current one. The solution in the neighbourhood which is the most fit is selected as the new state, regardless of whether it is more or less fit than the current state.

One concern in such an algorithm is that it is entirely possible for it to be caught in an infinite loop and to stop propagating through the search space [19]. To avoid this situation, the TS algorithm uses a List of forbidden moves, called the tabu list. At any step, this list contains a history of the $L$ most recent moves, $L$ being the specified tabu length. Prior to deciding which neighbouring solution shall become the new state, the algorithm verifies that the move resulting in the most fit solution

is not contained in the tabu list. If the move is not forbidden, the fittest solution is selected as the new state. Otherwise, the algorithm considers the move resulting in the next best solution, and then the next until the move is not contained in the tabu list. If every locally available move is forbidden, then the algorithm stops and gets restarted using a new initial solution.

**Algorithm 4** : Tabu Search Outline

```
1: Let UC be a set of all tuples of parameter values that are not yet covered by
   the selected test frames;
2: while UC is not empty do
3:    Initialize the tabu memory;
4:    Generate a random initial solution;
5:    for a specified number of iterations do
6:       Generate the complete neighbourhood of current solution;
7:       Update current solution to the best neighbor solution which is not re-
         stricted by the tabu list;
8:       Update the tabu list;
9:    end for
10:   Let TF be the best test frame found;
11:   Add TF to the test set;
12:   Remove those tuples in UC that are covered by TF;
13: end while
```

### 2.5.5 AETG: Automatic Efficient Test Generator

The *Automatic Efficient Test Generator (AETG)* algorithm proposed in [9, 10] is a greedy algorithm which constructs a test set by repeatedly adding a test frame that covers a large number of non-covered value tuples (interactions). Because of this, the resulting test set has the property that a test frame created earlier has more significant impact on interaction coverage. This is an important practical advantage because in practice it is often the case that not all test cases are executed due to time or cost constraints; even in such a situation, the tester can maximize the interaction coverage by simply performing the tests in an earliest first manner.

The AETG algorithm works in an incremental manner. This algorithm starts with an empty test set and adds one test frame at a time until the 100% coverage for all $t$-tuples of values is achieved. The AETG algorithm uses a greedy strategy in selecting each test frame; it creates many different candidate test frames and selects from them the one that covers the greatest number of new combinations. Algorithm 5 presents a high level outline for AETG method for test case generation.

**Algorithm 5** : AETG Outline

```
1: Let UC be a set of all tuples of parameter values that are not yet covered by
   the selected test frames;
2: while UC is not empty do
3:    for k times do
4:       {Make a new candidate test frame:}
5:       Select the variable and the value included in most tuples in UC;
6:       Put the rest of the variables into a random order;
7:       For each variable in the sequence determined by previous step, select the
         value that together with previous selected values in the candidate test
         frame is included in or covers most tuples in UC;
8:    end for
9:    Let TF be the test frame among these k generated test frames that covers
      the most tuples in UC;
10:   Add TF to the test set;
11:   Remove those tuples in UC that are covered by TF;
12: end while
```

The number of test frames generated by the AETG algorithm is related to the number of constructed candidates ($k$ in the algorithm) for each test frame. In general, larger values of k yield smaller numbers of test frames. However, Cohen et al. [9] report that using values larger than 50 will not dramatically decrease the number of test frames.

### 2.5.6 IPO: In-Parameter-Order

For a system with two or more variables, the *In-Parameter-Order (IPO)* combination strategy [33, 52] generates a test suite that satisfies 100% pairwise coverage for the values of the first two variables. The test suite is then extended to satisfy pairwise coverage for the values of the first three variables, and continues to do so for the values of each additional variable until all variables are included in the test suite. To extend the test suite with the values of the next variable, the IPO strategy uses two algorithms. The first algorithm presented as Algorithm 6 which is related to horizontal growth of the test suite extends the existing test frames in the test suite with values of the next variable.

The second algorithm, vertical growth, shown as Algorithm 7, creates additional test frames such that the test suite satisfies pairwise coverage for the values of the new variable.

IPO algorithm has some advantages over AETG algorithm. AETG is funda-

**Algorithm 6** : IPO Outline (Horizontal Growth)

Let $\tau$ be a test suite that satisfies pairwise coverage for the values of parameters $p_1$ to $p_{i-1}$;
Assume that parameter $p_i$ contains the values $v_1, v_2, \ldots, v_q$;
Let $\pi$ be pairs between values of $p_i$ and values of $p_1$ to $p_{i-1}$;
**if** $|\tau| \leq q$ **then**
  **for** $1 \leq j \leq |\tau|$ **do**
    Extend the $j$th test frame in $\tau$ by adding value $v_j$;
    Remove from $\pi$ pairs covered by the extended test frame;
  **end for**
**else**
  **for** $1 \leq j \leq q$ **do**
    Extend the $j$th test frame in $\tau$ by adding value $v_j$;
    Remove from $\pi$ pairs covered by the extended test frame;
  **end for**
  **for** $q < j \leq |\tau|$ **do**
    Extend the $j$th test frame in $\tau$ by adding one value of $p_i$ such that the resulting test covers the most number of pairs in $\pi$;
    Remove from $\pi$ pairs covered by the extended test frame;
  **end for**
**end if**

**Algorithm 7** : IPO Outline (Vertical Growth)

Let $\tau$ be the set of already selected test frames;
Let $\pi$ be the set of still uncovered pairs;
Let $\tau'$ be an empty set;
**for** each pair in $\pi$ **do**
  {Assume that the pair contains value $w$ of variable $p_k$, $1 \leq k < i$, and value $u$ of $p_i$}
  **if** $\tau'$ contains a test frame with "-" (non-determined value) as the value of $p_k$ and $u$ as the value of $p_i$ **then**
    Modify this test frame by replacing the "-" with $w$;
  **else**
    Add a new test frame to $\tau'$ that has $w$ as the value of $p_k$, $u$ as the value of $p_i$, and "-" as the value of every other parameter;
  **end if**
**end for**
$\tau = \tau + \tau'$;

mentally non-deterministic, whereas IPO is deterministic and this characteristic makes it more predictable in terms of the size of the final test suite. Also comparing the efficiency, AETG has a higher order of complexity, both in terms of time and space, than IPO.

### 2.5.7 CATS Algorithm

The *Constrained Array Test System (CATS)* algorithm for generating test cases is based on a heuristic algorithm that can be custom designed to satisfy $t$-wise coverage. The algorithm was described by Sherwood [47] and an outline of the algorithm that generates a test suite to satisfy $t$-wise coverage is shown in algorithm 8.

**Algorithm 8** : CATS Outline

1: Let $UC$ be a set of all tuples of parameter values that are not yet covered by the selected test frames;
2: Let $Q$ be the set of all possible combinations (test frame) not yet selected;
3: **while** $UC$ is not empty **do**
4: Select test frame $TF$ from $Q$ by finding the combination that covers most pairs in $UC$. If more than one combination covers the same amount select the first one encountered;
5: Add $TF$ to the test set;
6: Remove $TF$ from $Q$;
7: Remove those tuples in $UC$ that are covered by $TF$;
8: **end while**

The CATS algorithm has some similarities with AETG. The main difference is that CATS examines the whole list of unused test frames to find one that adds as much new coverage as possible while AETG constructs test case candidates one parameter value at a time based on coverage information. The constructed candidates are then evaluated to select the best possible, which is added to the test suite. In CATS it is guaranteed that best test frame is always selected while in AETG there are no guarantees. However, for large test problems, AETG is more efficient since only a small set of test frames has to be evaluated in each step. The non-deterministic nature of the AETG algorithm makes it impossible to exactly calculate the number of test frames in a test suite generated by the algorithm.

## 2.6 Summary

This chapter presented an overview of existing software testing methods. We discussed about requirements-based software testing and partition testing and reviewed the reasons why these techniques are popular among software testers. We will come back to these two techniques in the next chapter that we try to put them together in an integrated framework. We also reinvestigated the combinatorial test case generation problem as an optimization problem and surveyed a group of different combination strategies in the literature which try to solve the NP-complete test case generation problem using enumerative and local search techniques.

# Chapter 3

# A Framework for Requirements-Based Partition Testing

"Requirements-Based Testing" is a validation testing technique where we consider each requirement and derive a set of test cases for that requirement. There is also another technique in the literature, called "Partition Testing", which is used for minimizing the number of permutations and combinations of input data. In this second technique the assumption is that input data and output results often fall into different classes where all members of a class are related. Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member. Hence, we just need test cases to be chosen from each partition. Both of these techniques have advantages for the testing process.

Although both of these techniques are mature and are addressed widely in the literature and despite the general agreement on both of these key techniques of functional testing, a combination of them has not been studied systematically. In this chapter we propose a framework along with a procedural process for testing a system using "Requirements-Based Partition Testing" (RBPT). The idea is putting the two techniques together might give a solid technique which can be used in various domains for functional testing of large systems.

## 3.1 Proposed Layered Architecture for the RBPT Framework

In our RBPT framework the process starts from a list of *requirements*[1]. These requirements normally come from a *requirements group* in the organization. In most cases, these requirements are defined in natural language and do not follow any formal structure. Our goal is to process these requirements to come up with a list of *test cases* that can verify the whole set of requirements effectively. As Figure 3.1 illustrates, we need some intermediate structured components to transmit the available requirements through a process which ends up to a list of required test cases.

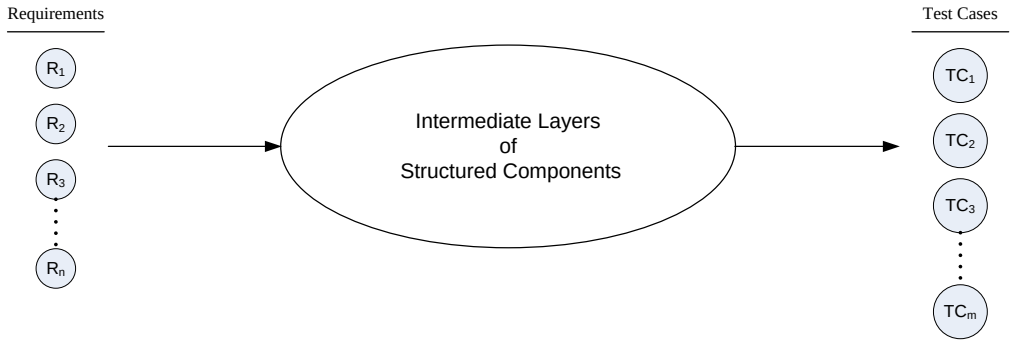



Figure 3.1: The RBPT process starts with a list of requirements and ends up to a list of required test cases.

What we propose in our framework is a layered architecture and a step by step process. This process can transform the requirements smoothly through those intermediate layers toward the final list of required test cases. We use five layers of intermediate components (other than requirements and test cases themselves) for our process, namely: i) Features, ii) Atomic Features, iii) Test Scenarios, iv) Frame Sets, and v) Test Frames (Test Configurations).

Figure 3.2 shows these layers of components and the relations from each layer to the next one. In the remainder of this section, we will discuss these intermediate components layer by layer and explain how we can go forward from each layer to the next one.

[1]In this chapter our emphasis is on functional requirements.

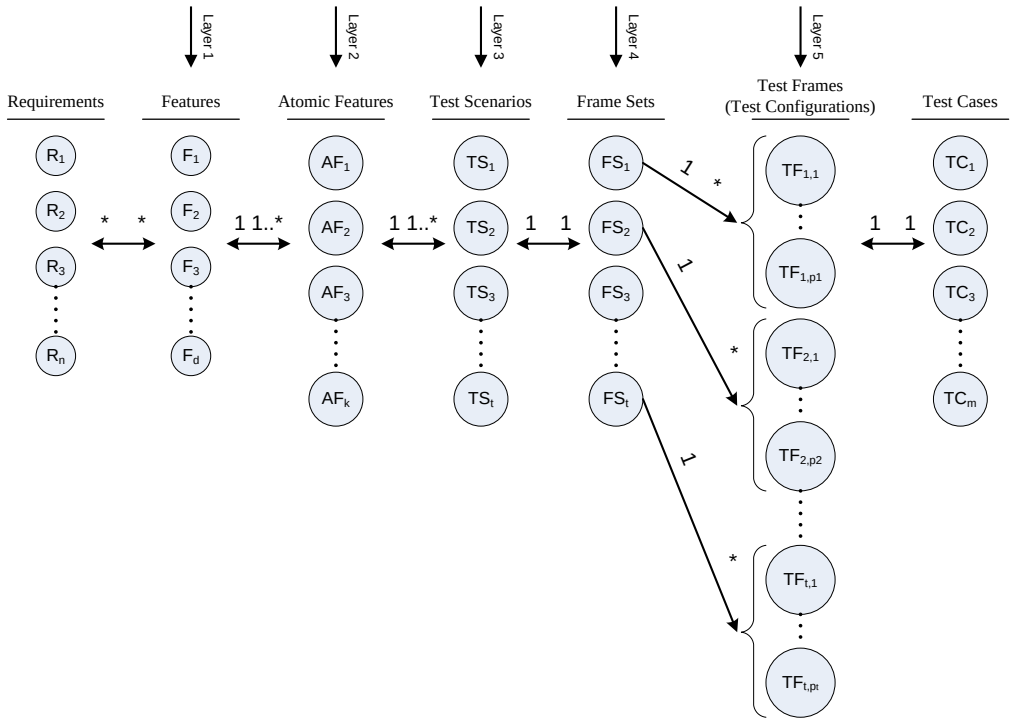



Figure 3.2: Five Intermediate Layers of Components Between Requirements and Test Cases and Their Relations

### 3.1.1 Features Layer

As mentioned before, requirements normally come from a requirements group in the organization and do not follow any formal structure. This lack of accuracy in defining the requirements may cause various problems for testers. Here is a partial list of such problems.

- The requirements may not be understandable by testers.
- The requirements may need to be *validated* to assure that they define the various aspects of the system in a correct way. They also may need to be checked not to have any contradictions or inconsistencies among them.
- The language used in the definition of the requirements may not be suitable to be used in the testing group. Testers sometimes prefer to use their own words and language for describing the system. This helps them to understand the requirements in a more clear manner.
- Sometimes a group of the requirements may not be testable. They may need to be changed in a way that can be verified by testers.

- Some requirements may refer to the same thing and cause some redundancy or one requirement may refer to more than one aspect of the system and may be required to get broken.

In our framework, we think about testing group as an independent department in the organization. This means that, because requirements come from a different group, testers can not change them directly. Requirements are defined by requirements group and have got approved. Testers can only use this output as their baseline and define their process based on that.[2]

To handle these problems, we put a layer in the process called "Features Layer". This layer is shown in Figure 3.2 as Layer 1. In one sentence we can say that features are a translation of requirements for testing group. Testers should generate a list of features considering the existing requirements. Putting such a layer in the process gives testers this freedom to rephrase the requirements however they prefer, using their own words and language. This layer in the process has a bunch of benefits for testers. They can use it to get along with the problems they face in terms of the requirements:

- Features are more understandable for testers. Simply because they have defined them themselves! However they need to have a clear understanding of the requirements to be able to generate the list of features. According to Ostrand [39]:

  > ...the tester has to ask the specification writer to clarify the intention of a particular section or sentence. These questions are themselves a form of testing, as they may expose errors in the specification before any code is written or in the implementation before any code is executed.

  If testers be able to produce the features from requirements, we can make sure that they have reached a high level of comprehension of the requirements.

- Testers need to define the features according to the existing requirements. This gives them this opportunity to validate the requirements at the same

[2]This does not mean that we deny the cooperation between different departments in the organization. Testing group may be able to give some feedback to the requirements group to change the defined requirements. However, since the requirements are a type of approved *configuration item* and may get used by other departments, testers won't be able to make changes in them freely as they like.

time. At the end of the day when the list of features is ready we know that they are consistent and complete and can be used for the rest of the process. Again according to Ostrand [39]:

> ... not only do natural language specifications lack structure, but they are frequently incomplete, ambiguous, or self-contradictory. The process of transforming the specification into an intermediate representation can be useful for revealing such problems.

- Unlike the requirements which come from requirements group, features are items inside the testing group. Testers will be able to change them whenever they like and also they can use their own words for describing the system. This makes the whole process more understandable for testers.
- If a requirement is not testable according to the understanding of testers, they can consider it in the features they produce. They may rephrase the requirement or transform it to an equivalent form to make it testable. Even they may produce a null feature assigned to that specific requirement to show that they have not covered that requirement.
- The redundancy and complexity problem of the requirements can also get fixed using the features layer. Redundant or similar requirements can get mixed in one feature and complex requirements, on the other side, may get broken to two or more simpler features.

We should note that although we put features layer as an intermediate component layer in the process to manipulate the requirements in a better way, we still need to keep an eye on the requirements themselves. By defining the features we do no remove the requirements. They are still our baseline and our final goal is to verify the system under test against those approved requirements. Features help us to make the the testing process more smooth but we should keep track of the relations between the requirements and their correspondent features continuously. As Figure 3.2 indicates the relation from requirements to features is a many to many relationship. This provides complete flexibility to a tester to arrange the features according to his/her own preference. A many to many relationship also supports removing non-testable requirements in features layer, breaking one requirement to two or more features, combining similar requirements into one feature and adding extra features required by testers. All these relations should be recorded. We will be able to use these data in order to trace back the process to the requirements and measure the requirements coverage of the generated final test cases.

### 3.1.2 Atomic Features Layer

The second intermediate layer in our framework is called "Atomic Features Layer", as Figure 3.2 depicts. The purpose of this layer is breaking the features to a set of atomic, non-breakable items which are named *atomic features* in this context.

After generating a list of testable features and assigning them to the corresponding requirements, we should continue the process toward the generation of final test cases. In most cases, the requirements and hence the related features are expressed in a general manner. These general features may associate with different aspects of the system under test or even different components in the system. Normally we can not test such general features directly. Hence, before taking any other step, we need to decompose these features as much as we can. For doing this, testers should break each one of these general features to a non-empty set of atomic features which can not get broken anymore. These atomic features should refer to an atomic functionality of the system. They should expose a one-directional view from the system under test.

Since features and atomic features are defined in natural language, giving a comprehensive definition of them is not straightforward. However, the goal of the layer is clear: we need to make features as much simple and pure as we can. Later in the process we may come up with a situation where we find out that some atomic features can be decomposed into simpler features. In those cases, we need to come back to the atomic features and polish them for resolving the inconsistencies occurred in the process.

Some people may argue that we can combine Layers 1 and 2 in the framework and decompose the requirements to atomic features directly. This is possible, however it has some drawbacks. Separating features layer from atomic features makes the process more smooth and applicable for testers. These two layers have some similarities but they have different goals embedded in their nature. Features layer tries to rephrase the requirements for the purpose of making them understandable for testers while the main concern for atomic features is that these features get separated from each other as much as possible. Putting these layers together makes the process hard to comprehend and manage for testers in practice.

As Figure 3.2 shows for the relations from Layer 1 to Layer 2, each feature should get broken to one or more atomic features in the second layer. We assign exactly one atomic feature to a feature on the first layer (i.e. copy the feature itself to the next layer as an atomic feature) when it is atomic itself and can not

get decomposed anymore. This way we have exactly one parent feature for each atomic feature and we can trace back to the requirements layer if required.

### 3.1.3 Test Scenarios Layer

Now that we have decomposed features to atomic features, next step would be to start testing these atomic features. Actually requirements layer and the first two intermediate layers in the framework help testers to get the requirements as an input, understand and prepare them for testing. From the second intermediate layer forward the main testing process gets started. In this third layer, testers should try to figure out how they are supposed to test each atomic feature by writing a "Test Scenario" for it.

A test scenario is a story that describes a hypothetical situation. In testing, we check how the program copes with this hypothetical situation. The ideal scenario test is i) motivating, ii) credible, iii) complex and iv) easy to evaluate [29]. Cem Kaner, in his paper published at 2003, talks about these characteristics of an ideal test scenario in detail. He also provides a thorough list of guidelines for generating good scenarios. A good test scenario should also contain a list of *pre-conditions*, *steps* and *post-conditions* which should be concerned during the testing of its related atomic feature.

The test scenarios in our framework are slightly different from what Kaner introduces. We limit the test scenarios to be defined for each atomic feature separately. This means that testers go through atomic features one by one and define a set of scenarios for testing that specific atomic feature. This way each test scenario would be assigned to exactly one atomic feature. Normally, the scenarios described by Kaner and used in *scenario testing*, try to test the business process flows from end to end. This results in the following problems:

- Finding good scenarios will be a complex process and will require a large amount of detailed and technical knowledge from different aspects of the system under test. Gaining such information will take a large portion of testers' time.
- Figuring out which requirements are covered and which ones still need other scenarios to be defined for them, will need an extraordinary amount of effort.

For getting along with these problems, we define the test scenarios by focusing

on the atomic features, one at a time. When we limit the domain to just one atomic feature we can benefit from following enhancements:

- Testers deal with just a small part of the system under test, so they can prepare suitable effective scenarios for testing that portion of the system more efficaciously.
- By defining a set of scenarios for testing each atomic feature we get assured that we have not missed any requirement to be tested. Since atomic features cover all the requirements, we reach 100% requirement coverage by defining the required test scenarios for each atomic feature.

The relation between atomic features and test scenarios in Figure 3.2 emphasizes that we need at least one test scenario for each atomic feature.

### 3.1.4 Frame Sets Layer

In this step, we have a list of test scenarios which we should verify them. In our framework, as its names implies, we use "Partition Testing" and more specifically "Category Partition Method" [39] for this purpose. Again, we go through test scenarios one by one. For verifying a specific test scenario we follow the standard approach for category partition method, described by Ostrand et al. in their 1988 paper [39]:

> The tester specifies *categories* of environments and input values, partitions each category into a set of mutually exclusive *choices*, and describes *constraints* that govern the interactions between occurrences of choices from different categories.

As mentioned above testers should analyze the test scenarios one by one. According to the *input values* and *environment conditions* embedded in each test scenario and its associated atomic feature, testers generate a list of *categories* (called also "Variables" or "Parameters" in the literature). We use the term "Variable" in the remainder of this chapter for referring to these categories. A "Variable" is a major property or characteristic of an input parameter or environment condition that can affect the test scenario's execution behaviour. After determining the variables involved in the test scenario, testers partition each variable into its distinct *choices*.

Each choice is in fact a partition class of the possible values that can be assigned to one variable. In another word, we partition the possible values of each variable to a number of mutually exclusive partition classes and call each class a choice. Sometimes in the literature the term "Value" itself may be used to refer to a choice (a class of values). Following that nomenclature we also use the term "Values" instead of choices. Actually each value is a representative for one of the partition classes of its related variable. We need just one representative from each partition class because we assume that all the elements in that partition are equivalent in terms of their effectiveness to reveal a defect. According to Ostrand [39]:

> The idea is that all elements within an equivalence class are essentially the same for the purposes of testing. If the testing's main emphasis is to attempt to show the presence of errors, then the assumption is that any element of a class will expose the error as well as any other one. If the testing's main emphasis is to attempt to give confidence in the software's correctness, then the assumption is that correct results for a single element in a class will provide confidence that all elements in the class would be processed correctly.

After determining the variables and values involved in one test scenario, we put all those variables and their related values together in a group called "Frame Set". Then associate that specific frame set to its corresponding test scenario. As Figure 3.2 depicts, frame sets have a one-to-one relation with test scenarios and contain the variables and values that affect the execution of their associated test scenario. We call them frame sets because as we will see later, in the next chapter, the variables and values of each frame set get used for generating a set of *test frames*.

Another important issue is the *priority* of the variables, inside their containing frame set. In another word, testers should prioritize the variables relatively. These priorities will be used later for generating effective test frames. Assigning a high priority to a variable means that as a tester we are supposed to consider different values of that variable in testing to a higher extend of coverage. On the other hand, assigning a low priority to a variable means that a lower extend of coverage is required for testing the test scenario behind each frame set against the values of that specific variable. We call a group of variables with same priority a "Priority Set" as Figure 3.3 illustrates.

Most of the time testers will need extra information from other departments such as "marketing" and/or "customer service" to find out which values or variables

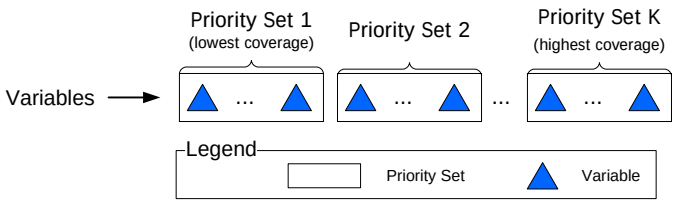



Figure 3.3: A Frame Set contains a set of prioritized variables.

should be considered as testers' main concern in the process. The historical data can be also considered as a major factor in the prioritization of variables, if they are available from previous similar projects and their results.

### 3.1.5 Test Frames Layer

Next layer in our framework is the "Test Frames Layer". As its names implies, in this layer we get the variables and values of each frame set and generate the required "Test Frames" (Test Configurations). A test frame is an instance of a frame set. For producing a test frame for an specific frame set, we assign to each variable included in that frame set, one of its possible values.

Each frame set can be thought about as a two dimensional matrix. The columns of that matrix are indexed by the variables included in the frame set. Each row of the matrix corresponds to one test frame, by assigning one possible value to each variable in the frame set. Figure 3.4 shows two frame sets, their variables and generated test frames. The length of the test frames (the number of values included in the test frame) may vary from one frame set to another. Actually the length of the test frames inside each frame set is equal to the number of variables associated with that specific frame set.

For generating test frames, there are different algorithms which combine the values of the different variables based on some combinatorial strategy. These algorithms are called "Combination Strategies" in the literature. Each combination strategy is a *test case selection* method [20]. We had an overview of different combination strategies in Section 2.5.

### 3.1.6 Test Cases Layer

The last layer of components in our framework is the "Test Cases Layer". Test cases are our final entities to be produced. The ultimate goal of our framework

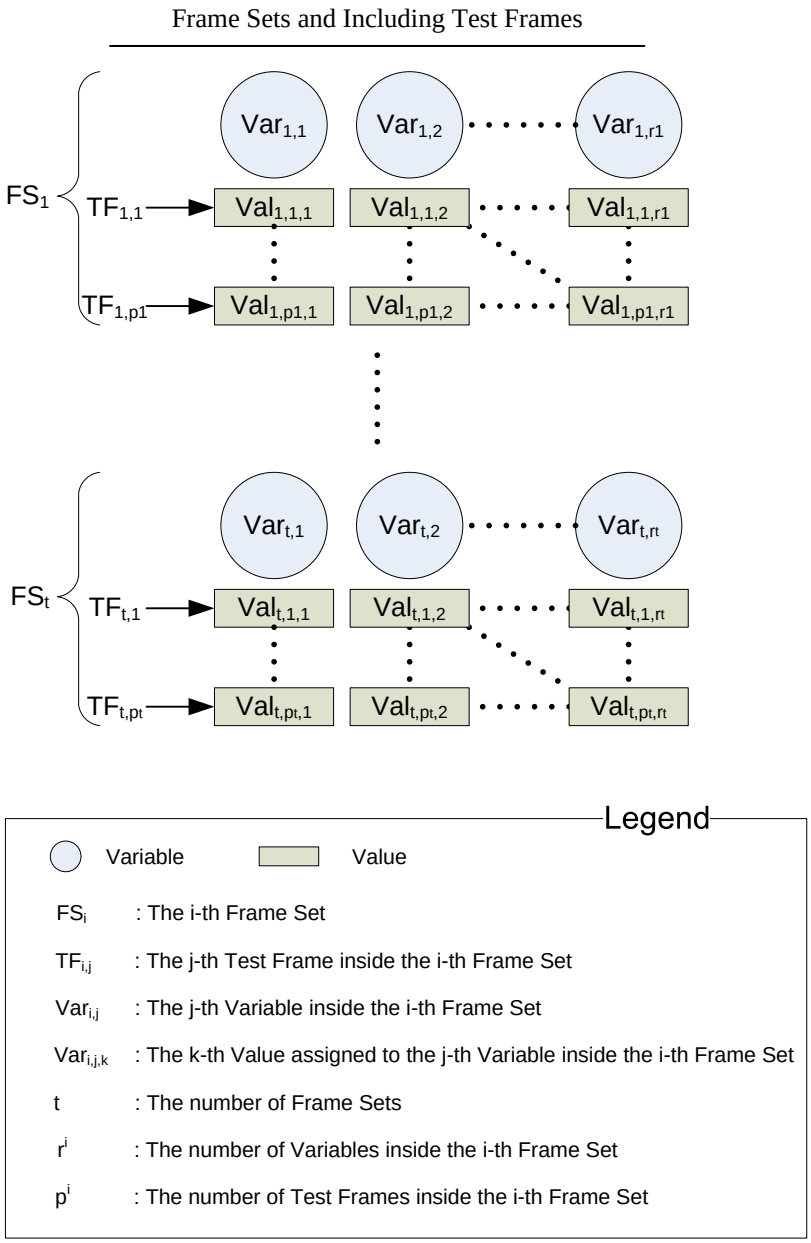



Figure 3.4: Each Test Frame assigns one possible value to each variable included in the frame set.

was to generate a thorough list of test cases to verify the system under test against its predefined requirements. In this last step, we generate one test case for each test frame. There is a one-to-one relationship between test cases and test frames as shown in Figure 3.2.

Each test case consists of a test frame (test configuration) and a scenario behind that test frame. Hence for producing a test case we simply combine each test frame with its associated test scenario. Since each test frame is associated with exactly one frame set and each frame set has a one-to-one relation with exactly one test scenario, so each test frame will have exactly one test scenario assigned to that. By combining each test frame with its corresponding test scenario we end up to an executable test case.

The test frames layer and the test cases layer can be produced automatically, unlike the previous layers which require manual process by testers.

## 3.2 RBPT-Based Test Case Generation Process

Now that we have got introduced with the layered architecture of the proposed framework, the next step will be to put everything together and define a test case generation process which traverses these layers for generating a complete test suite. The main goal of this process will be to formalize and automate the required activities for testing a newly developed system.

Figure 3.5 presents our proposed process. It consists of two main stages: "Requirements Modeling" and "Test Case Generation". The first stage helps testers build a requirements model that provides a semi-formal representation of the requirements. This model is then used in test case generation stage to automatically generate the optimal set of test cases that is needed to test the product adequately. Next, testers execute those test cases (outside the scope of the framework) and record the results. The following sections elaborate further on each aforementioned stages.

### 3.2.1 Requirements Modeling

The first stage of our proposed process deals with the first two intermediate layers of our framework: Features and Atomic Features Layers. The main goal of this stage is to address the validation of the requirements to make sure that they are

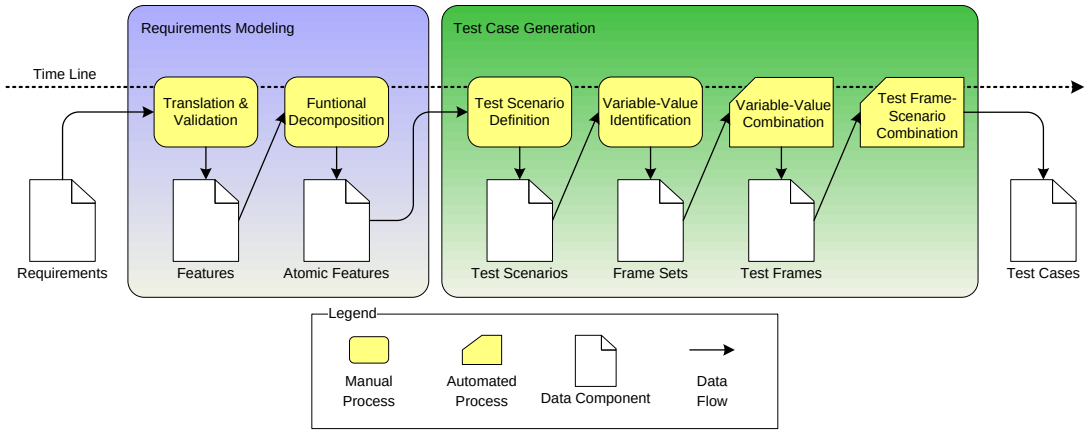



Figure 3.5: Test Case Generation Process Diagram

correct, complete, unambiguous, and logically consistent. As Figure 3.5 illustrates, requirements modeling stage consists of two main steps, namely: i) Translation and Validation and ii) Functional Decomposition.

In the first step mentioned above testers review the requirements. They try to understand the requirements and validate them against their expectations from the system under test. They also check that those requirements are in consistency with each other. This step is critical in the whole process, because in this step we develop a foundation for further testing and our future steps will rely heavily on the correctness and completeness of the product requirements. Next, testers rewrite the requirements as they like, using their own language and make a list of features as described in Section 3.1.1. The output of this step will be a list of consistent features which build the first intermediate layer in our proposed framework. The relation between requirements and features should be carefully tracked during this step and stored for further use.

The second step in the requirements modeling stage is functional decomposition of the features. As we discussed in Section 3.1.2 we need to break the features as much as possible from functionality point of view. In this step we do this decomposition and generate a list of atomic features which build the second intermediate layer of our framework. This decomposition helps us to concentrate on different modules and issues more accurately and increase the effectiveness of our testing process extensively. Again the relation between features and atomic features should be stored for future use in the process.

### 3.2.2 Test Case Generation

Test case generation is the second stage of our proposed process. In this stage our goal is to attain a necessary and sufficient set of test cases from software requirements to ensure that the design and code fully satisfy those requirements. Figure 3.5 depicts a four step plan for achieving this goal. These steps are called: i) Test Scenario Definition, ii) Variable-Value Identification, iii) Variable-Value Combinations, and iv) "Test Frame"-Scenario Combination.

The first step tries to use kind of scenario testing for verifying the atomic features generated in the previous step. Identifying good test scenarios requires considering an extended list of guidelines and specifications [29].

The second step starts to implement the category partition method [39] in our process. In this step testers define a set of variables and their associated values for each test scenario.

"Variable-Value Combination" step is simply for combining the chosen values of the participating variables with the purpose of producing efficient and complete set of test frames (test configurations).

In the final step, testers should put together the generated test frames and their corresponding test scenarios. For having a meaningful test case we need to have useful scenarios and also the parameters and variables inside that scenario should get assigned an appropriate value. Test scenarios in the third layer of our proposed framework are responsible for generation of perfect scenarios and test frames of the fifth layer are in charge for producing efficient combinations of values. In this step testers combine these two layers and the output will be a set of desired test cases which can get executed for verifying the system under test against predefined requirements.

## 3.3 Summary

In this chapter we designed a layered framework along with a procedural process for testing a system using "Requirements-Based Partition Testing" (RBPT). We discussed about each layer of the framework and how it is useful. The chapter also put all the layers in the framework together and defined a test case generation process. This process can be used to traverse the layers in the framework for generating a complete test suite.

# Chapter 4

# Particle Swarm Optimization for Test Case Generation

Choosing appropriate test cases is an essential part of software testing that can lead to significant improvements in efficiency, as well as reduced costs of combinatorial testing. Finding minimum size test sets is NP-complete. Therefore, artificial intelligence-based search algorithms have been widely used for generating near-optimal solutions. In this chapter, we propose a novel technique for test case generation, using *Particle Swarm Optimization (PSO)*, an effective optimization tool which has emerged in the last decade. Through some experiments, we illustrate how this new technique can outperform other existing test generation methodologies.

## 4.1 Introduction

Software testing is a fundamental part of the software development life cycle (SDLC). This stage of SDLC usually takes a big portion of the time required before software release. Different testing methodologies are available to unravel any bugs that might have been committed during previous phases. Choosing appropriate test cases, can significantly increase the efficiency of testing, and at the same time reduce any costs associated with it. This problem has been studied extensively in the software engineering literature.

*Partition testing* and *combinatorial testing* refer to two sets of effective techniques which are widely used for generating effective test cases. Partition testing techniques [56, 39] divide the input domain of the program under test into subsets with the testers choosing one or more elements from each subset. The assumption

is that all the elements in one subset are the same for the purpose of testing and revealing bugs. These partitioning techniques result in input parameter models, which are representations of the input space of the system under test via a set of parameters and values for these parameters. Combinatorial testing [6] is usually referred to a group of test case selection algorithms and techniques which use combinatorial designs for generating efficient test suites. Usually the large number of parameters and values in the input parameter model needs a large set of combinations to be tested. Combination strategies try to minimize the required number of combinations for testing and generate test suites of reasonable size without loss of effectiveness of test cases.

There are various combination strategies in the literature [20]. Some of them are deterministic and generate a fixed number of test frames for the same input while the others are non-deterministic and include some sort of randomness. Non-deterministic algorithms may result in different test frames or even different number of test frames during each execution on the same input. A group of deterministic algorithms use algebraic notions such as orthogonal arrays and covering arrays. Since construction of minimum size test set is NP-complete [33] it is unlikely to find an efficient polynomial algorithm which always generates the optimal test set. Hence, most of the non-deterministic algorithms get help from heuristic-based techniques and artificial intelligence-based search algorithms. Various search-based algorithms have been developed in the literature which try to generate a near-optimal test set in a very short time. *Genetic algorithms*, *ant colony algorithm* and *simulated annealing* are some of the well-known artificial intelligence-based techniques which are used for test case generation [48, 38]. However these algorithms have started getting competition from other heuristic search techniques, such as the *Particle Swarm Optimization (PSO)*. Various works (e.g. [59, 8, 24, 26]) show that particle swarm optimization is equally well-suited or even better than some other techniques in different domains. At the same time, a particle swarm algorithm is much simpler, easier to implement and has just a few number of parameters that the user has to adjust. These properties make PSO an ideal technique for test case generation. In this paper we propose a new algorithm based on swarm particle optimization technique (Section 4.3).

For comparing the resulting PSO combination strategy with other existing techniques in the literature, we developed a framework which can be used for automatic evaluation of different algorithms (Section 4.4.1). Normally, the size of the generated test suites gets used as a metric for comparison of different test case generation techniques. In Section 4.4.1, we also propose a new effectiveness measure which

produces better and more precise assessment from the output of such algorithms. Finally in Section 4.4.2, we use our proposed framework and effectiveness measure for executing some experiments and show that PSO combination strategy can be as effective as other existing algorithms and even beat them in some cases.

## 4.2 Test Suites as Covering Arrays

In the combinatorial design literature a *covering array* $CA_\lambda(N; t, k, v)$, is an $N \times k$ array on $v$ symbols such that every $N \times t$ sub-array contains all the $t$-tuples from those $v$ symbols at least $\lambda$ times [11]. In other words, any subset of $t$-columns of this array will contain each $t$-tuples of the symbols at least $\lambda$ times. When $\lambda = 1$ we use the notation $CA(N; t, k, v)$. In such an array, $t$ is called the strength, $k$ the degree and $v$ the order. A covering array is optimal if it contains the minimum possible number of rows. We use the notation $CA_\lambda(t, k, v)$ when the number of rows is not determined yet. The following table shows a covering array with 9 rows which covers all the 3-tuples of the 2 symbols (0 and 1 in this example) from 4 variables (parameters).

| Test Frames (Configurations) | Variables (Parameters) |
|---|---|
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0100 |
| 4 | 0111 |
| 5 | 1000 |
| 6 | 1010 |
| 7 | 1101 |
| 8 | 1110 |
| 9 | 1011 |

Table 4.1: $CA(N = 9; t = 3, k = 4, v = 2)$

This covering array is equivalent to a test suite which contains 9 test frames. Each test frame assigns values to 4 variables where each variable has 2 values. The whole test suite provides 3-wise coverage on the values of these 4 variables. The mentioned covering array in the above example is not optimal. We can generate a covering array with 8 (instead of 9) rows which gives us the same coverage. Finding an optimal covering array of strength $t$, or equivalently generating a minimum size test suite with $t$-wise coverage, is an NP-complete problem. If the number of values

for each variable is different we use *mixed level covering arrays*. A mixed level covering array $MCA_\lambda(N; t, k, (v_1 v_2 \dots v_k))$ is a covering array where each variable has $v_i$ distinct values and $v = \sum_{i=1}^{k} v_i$. Each column $i$ of the mixed level covering array contains only elements from the $v_i$ values of the $i$th variable. We use a shorthand notation to describe covering arrays by combining equal entries in $(v_i : 1 \leq i \leq k)$. For example three entries each equal to two can be written as $2^3$. Consider a $CA(N; t, (w_1^{k_1} w_2^{k_2} \dots w_s^{k_s})$. In this array we have:

$$k = \sum_{i=1}^{s} k_i \text{ and } v = \sum_{i=1}^{s} k_i w_i$$

The same shorthand notation can also be used for mixed level covering arrays.

## 4.3 Particle Swarm Optimization for Software Testing

In this section, we will have an overview on particle swarm optimization technique and we will show how we can use it for test case generation. Section 4.3.1 describes the outline of the PSO, and Section 4.3.2 provides the details of applying PSO in test case generation for combinatorial testing.

### 4.3.1 PSO Technique

*Particle Swarm Optimization* (PSO) is a very effective optimization tool, which has emerged in the last decade. It was first introduced in 1995 by Kennedy and Eberhart [17, 31]. Although, the original aim was to simulate the behavior of a group of birds or a school of fish looking for food, it was quickly realized that it can be applied in optimization problems.

PSO is similar to GA (Genetic Algorithm) and ACA (Ant Colony Algorithm) in the way that it is a population-based meta heuristic algorithm. It is an approach that manipulates a number of candidate solutions at once. A *solution* is referred to as a *particle*, the whole population is referred to as a *swarm*. Each particle represents a solution and moves in the search space to find better positions in the space or in another word better solutions for the problem. Each particle also holds the information essential for its movement such as:

- Its current position: $\mathbf{x_i}$
- Its current velocity: $\mathbf{v_i}$
- The best position it has achieved so far which is called *personal best*: $\mathbf{pBest_i}$
- The best position achieved by the particles in its neighborhood which is called *local best*: $\mathbf{lBest_i}$
- The best position achieved by the particles in the whole swarm which is called *global best*: $\mathbf{gBest_i}$

Particles adjust their velocity to move towards their personal best, local best and the swarm's global best.

PSO starts with a set of random solutions by assigning a random position to each particle. Then similar to other local search algorithms it iteratively updates the position of the particles in the hope of finding better solutions. During these iterations each particle explores the search space by changing its position according to an *update rule*. Update rule normally guides each particle toward the best positions achieved by the particle itself, its neighbor particles and the best position achieved by the whole swarm. This leads to further explorations of regions that turned out to include more profitable solutions. Figure 4.1 shows the general outline of the PSO algorithm.

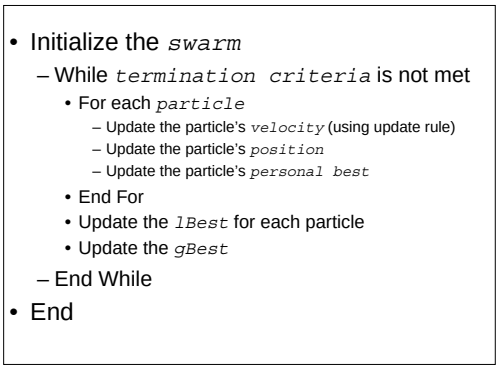



Figure 4.1: A Simple Outline for PSO with Synchronous Update

PSO can be *synchronous* or *asynchronous* depending on the location of **lBest** update. This can be done outside the **for** loop as Figure 4.1 illustrates or can be moved inside the loop. The former is called synchronous PSO and the latter is asynchronous. Asynchronous version usually produces better results as it causes

the particles to use a more up-to-date information, however this might not be the case, depending on the underlying problem.

### 4.3.2 PSO for Test Case Generation

Here we propose our method for test case generation using PSO.

**Particle Initialization in Discrete PSO:** In PSO each particle is a vector. The order of each vector is the same as the order of the problem's search space. In the test case generation problem, we want to find test frames that give us better coverage on the values of the related variables. If each test frame contains $D$ variables then the search space and hence the particle vectors will also be $D$-dimensional. Since PSO is designed for solving continuous problems, each dimension of the particles should be able to hold any real number. However, the test case generation problem and many other optimization problems are set in a space featuring discrete variables, so we require the use of a discrete version of PSO for dealing with these problems. For dealing with this problem, first we initialize each particle vector with discrete values. Each particle will be a $D$-dimensional vector say $\mathbf{x_j} = (x_j^1, x_j^2, \ldots, x_j^D)$ where each dimension is an integer between 0 and $v_i$ (number of the values of the $i$th variable). Also, during the execution of the algorithm, we simply round the calculated velocities to the nearest integer number. Since the initial positions and the velocities of particles are integers, each particle is guaranteed to have integer positions at all times.

**Particle Motions:** In PSO particles move around the search space using an *update rule*. The update rule is normally in the following form:

$$
\begin{aligned}
v_j^d(t) &= \omega v_j^d(t-1) && (4.1)\\
&+ cr_j^d(pBest_j^d(t-1) - x_j^d(t-1)) && (4.2)\\
&+ c'r'^d_j(lBest_j^d(t-1) - x_j^d(t-1)) && (4.3)
\end{aligned}
$$

In each iteration of PSO, the velocity of each particle ($\mathbf{v_j}$) gets updated according to the update rule and the particle moves around in the search space by adding the newly calculated velocity to its current position. As we mentioned before, we round the value of each dimension to the nearest integer number after each update.

In the above rule $t$ is the *time* or the *iteration number* where $j$ and $d$ refer to the *particle index* and the *dimension* respectively.

The first line of the update rule is called the *inertia* component which accommodates the fact that a particle should not change its direction of movement suddenly. The $\omega$ factor is the *inertia weight* which can be adjusted to increase or decrease the amount of freedom a particle has for changing its direction. This parameter regulates the trade-off between the global (wide-ranging) and local (nearby) exploration abilities of the swarm. A small inertia weight facilitates global exploration (searching new areas), while a large one tends to facilitate local exploration, i.e. fine-tuning the current search area. A suitable value for the inertia weight usually provides balance between global and local exploration abilities and consequently results in a reduction of the number of iterations required to locate the optimum solution. According to [2], it is better to initially set the inertia weight to a large value, in order to promote global exploration of the search space, and gradually decrease it to get more refined solutions.

The second and third lines of the update rule are called *cognitive* and *social* components respectively. Here, $c$ and $c'$ factors in the beginning of these two components are *acceleration coefficients* which adjust the weight between cognitive and social components of the update rule. Increasing $c$ shifts the weight toward cognitive component and causes the particles trust their own experience more and move toward their **pBest**. On the other hand increasing $c'$ makes the social component more impressive for particles and guides them toward their **lBest**. These factors should get configured considering the problem domain.

For generating randomness in the update rule, two random factors $r$ and $r'$ are used which are random real numbers between 0 and 1. These two random factors are generated for each dimension of each particle. Using the same random number for all the dimensions of a particle results in *linear PSO* which usually produces sub-optimal solutions.

**Boundary Condition:** In the test case generation problem each variable has a fixed number of values and each dimension of the particle vector should refer to one of the values of its correspondent variable. In other words, for a particle $j$, all dimensions of its position $\mathbf{x_j}$ should lie in $[0, v_i]$ where $v_i$ is the number of the values for the $i$th variable. For meeting this condition during the execution of the algorithm, we need to define a higher and lower bound for velocity dimensions and set a *boundary condition* to handle the overflow situations where a particle flies out of the permitted search space.

Setting the the maximum velocity dimension allowed for the particles, $V^i_{max}$, is an important factor in PSO. If the maximum velocity is too high, particles can fly past optimal solutions easily, resulting in poor final results. On the other hand, if it is too low, particles can get stuck in local optimum. Since the particle dimensions should be bound to the numbers between 0 and $v_i$, defining $V^i_{max} = v_i/2$ and restricting the velocity dimensions to be in $[-V^i_{max}, V^i_{max}]$, seems to be a good choice for the test case generation problem because it both bounds the velocity and provides coverage on the whole space.

Even after limiting the velocity of each particle, we need a boundary condition to handle the cases where a particle goes out of the permitted limits. There are a few different boundary conditions described in the literature such as *absorbing walls*, *reflecting walls* and *invisible walls* which are proposed in [45]. For the *absorbing walls* boundary condition when the particle reaches the boundary of a dimension, the velocity in that dimension changes to zero. For the *reflecting walls* boundary condition the sign of the velocity in the related dimension toggles when particle reaches the boundary. Finally for the *invisible walls* boundary condition the particle is allowed to fly through the boundary of the dimension, but the fitness of such a particle outside the boundaries is not computed. *Damping walls* [27] is another boundary condition which tries to lie in between the two absorbing and reflecting techniques. In this boundary condition whenever a particle tries to escape the search space in any of the dimensions, part of the velocity in that dimension gets absorbed by the boundary and the particle is then reflected back to the search space with a damped velocity along with a reversal of sign. Figure 4.2 depicts how these boundary conditions work.

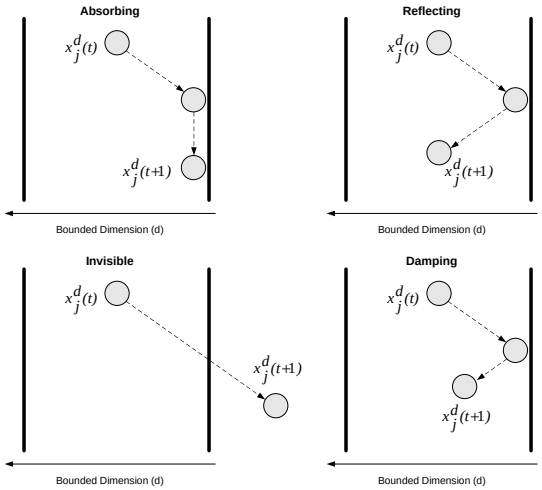



Figure 4.2: Boundary conditions keep particles inside the limited area by changing their velocity in the appropriate direction.

The problem with these boundary conditions is that they all try to keep the particle in the limited area by manipulating the velocity of the particle. Previously, we mentioned that each particle adjusts its velocity to move towards its personal best, local best and the swarm's global best. This is how the particle gets directed toward better solutions. The quick change of the velocity by boundary conditions interrupts the particle's smooth motion toward the target point. This is unavoidable in the situations where the search space is not bounded from one or more directions. However, in some other optimization problems such as test case generation, the search space is limited from all sides. In order to avoid the interruption in the velocity in such problems, we propose a novel boundary condition which we call "*cyclic walls*". This boundary condition can be used for the problems with finite search space.

Figure 4.3 illustrates how cyclic walls boundary condition works. With this boundary condition, whenever a particle tries to scape the search space in any of the dimensions, it continues its motion with the same velocity, starting from the other bound of that dimension. This happens by reseting the position in that dimension to the other end point of the limited interval for that specified dimension. This way the particles resides inside the search space without any extra manipulation of its velocity. This can be helpful because the velocity and moving direction of the particles are more important than their current position. Actually this is the moving direction that guides the particles according to the experience of the swarm toward more profitable regions and any interference in the velocity outside of the update rule is not desirable.

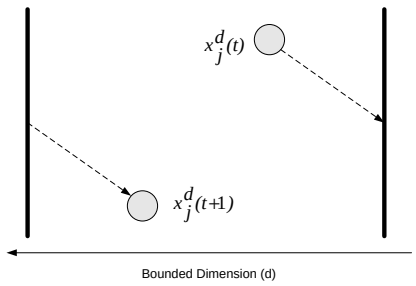



Figure 4.3: Cyclic Walls Boundary Condition: Particle resides in the search space by jumping to the other end point of the dimension without any interference in its velocity.

**Neighborhoods:** As the above update rule implies, the motion of each particle gets affected by the **pBest** of its neighbors. In another word particles share their personal best information with each other. Selecting a proper neighborhood has influence on the algorithm in many ways. It can improve the convergence speed of the algorithm and helps in avoiding getting stuck at local optimum.

Various neighborhood topologies have been introduced [18, 30]. The most obvious topology is the **gBest** model in which all the particles of the swarm are neighbor to each other and **lBest** is actually equal to **gBest**. This model causes fast propagation of information in a swarm but can get stuck easily in local optimum.

Other models define a more specific neighborhood for each particle. One way is to select the neighbors of a particle among those ones that are closest to it in the search space. Being close gets defined based on the distance of the particles in Cartesian space. This approach is more accurate but might be computationally expensive. For reducing the cost we define the neighborhood based on the data structure which maintains the particles. For example if the particles are stored in a matrix (or array), we consider those ones next to each other in the matrix as neighbors. Finally in order to have a fixed size for all the neighborhoods in the swarm, we assume a cyclic nature for any of the data structures, when selecting neighbor particles, even if that is not the case in reality. Still the size of the neighborhood should be adjusted according to the experimental results in the domain.

**Fitness Function:** The *fitness function* is used to estimate the goodness of a candidate solution. We define the fitness function $F(s)$ for a test case "$s$" as the number of new $t$-tuples from the values of the related variables that are not covered by the given test set but are covered by "$s$".

**Stagnation Condition:** In stagnation situation where there is no improvement

of **gBest** over a number of iterations, called stale period, we reset the position of all the particles to refresh the search one more time.

**Termination Criteria:** The same as other local search techniques, various termination criteria can be used for PSO such as: i) Reaching a maximum number of iterations, ii) Reaching a maximum number of evaluations, iii) Reaching an acceptable solution, and iv) Reaching a maximum number of stagnations.

We use the second criteria for this problem and stop the algorithms after $M$ iterations where $M$ is a fixed number which should be determined before evaluations. We choose this criteria because first of all it gives a better estimate of the cost of the algorithm in comparison with other criteria (The major cost in local search algorithms is usually related to the evaluation part.). Secondly it results in a fairer comparison of different algorithms. Those algorithms are better which generate better solutions using fewer number of evaluations and accordingly causing less cost.

## 4.4 Empirical Experiments

For comparing our proposed PSO method with other algorithms, we have developed a framework which can be used for automatic evaluation of different techniques. Section 4.4.1 describes this framework from a high level point of view. This framework is used in the Section 4.4.2 for a comprehensive empirical comparison between different test case generation techniques.

### 4.4.1 Test Case Generation Framework

For having a precise evaluation of our proposed technique and a fir comparison with other existing methods, we have developed a test case generation framework. Figure 4.4 schematically presents the high level structure of this framework.

**Test Case Generation Process:** The framework obtains its settings as an input from user (Figure 4.4). These settings include parameters such as the number of test sets to be generated, number of variables in each test set and maximum number of values for each variable. Then "Automatic Random Test Set Generator" builds a group of required test sets and "Input Tables" and stores the settings of each test set in one corresponding input table. The input tables also maintain the

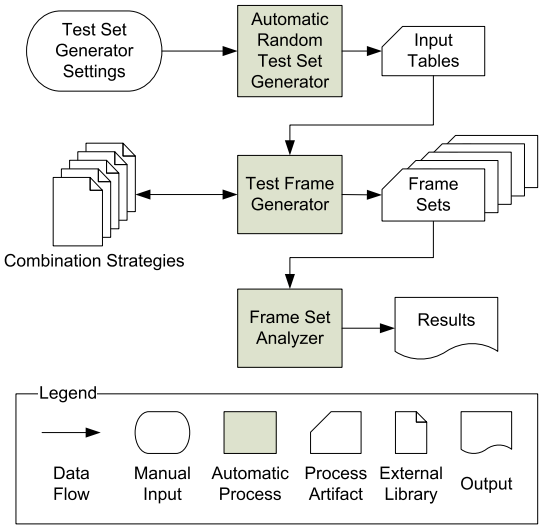



Figure 4.4: Test Case Generation Framework

relationships and constraints between different variables or values. As shown in Figure 4.4, these input tables get passed to the "Test Frame Generator" which has access to a group of "Combination Strategies". Then, the test frame generator executes each combination strategy on all of the "Input Tables" for a fixed number of times and places the "Test Frames" generated by each combination strategy for each input table in a separate "Frame Set". Finally, "Frame Set" analyzer goes through all of the frame sets and computes the required metrics and gathers a list of predetermined statistics. The output of the framework will be a set of "Results" according to the values of those metrics which ranks the combination strategies by their efficiency.

**Generated Test Objects:** Although lots of research has been done on different combination strategies there is no any comprehensive set of input data to be used for evaluation of these techniques. To deal with this lack of required input we decided to create a large set of test sets that feature different grades of complexity. AS we mentioned in Section 4.2, each test set with variables having varying number of values is equivalent to a multi level covering array $MCA(N; t, k, (v_1 v_2 \dots v_k))$. $k$ and $v_i$ which refer to the number of variables and values can be any positive integer greater than 2. We defined an upper bound of 20 for $k$ and an upper bound of 10 for $v_i$ which is the case in most of the practical situations. This way we will have a $19 \times 9$ table, each cell of which with indices $(i, j)$ refers to test sets with $i$ variables and maximum number of $j$ values for each variable. According to such a table, "Automatic Random Test Set Generator" 4.4 builds one test set related to each

cell of the table. This provides us 171 test sets with various amount of complexity which can be used in benchmarking of different combination strategies.

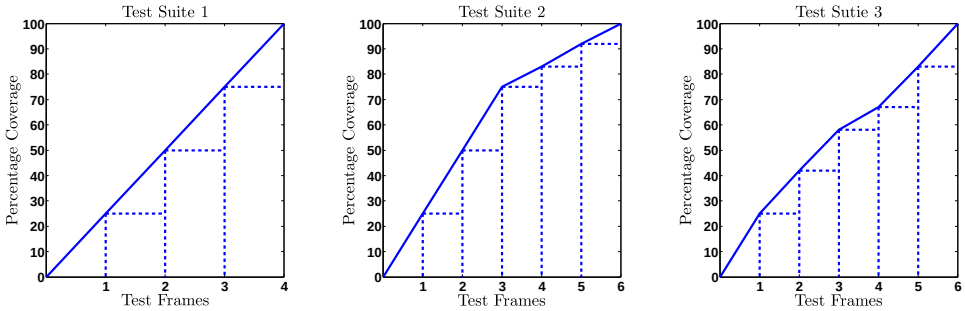
**Test Case Generation Metric:** After executing different combination strategies on the test objects produced by the framework's test set generator, we need to evaluate the output of these different algorithms. Normally the size of the generated test suites gets used as a criteria for evaluation of each technique. Since most of the existing techniques perform similar in terms of efficiency, the size of the produced test suites will be also close to each other. In order to have a better evaluation of the results and exaggerate the distance between these techniques, we need a finer metric to be used for comparison. Here, we propose a novel metric for assessing the output of the test case generation techniques which helps us to have a more precise benchmarking.

One of the main goals in the test case generation problem is to reach higher coverage using fewer number of test cases. Using the test suite size as the evaluation metric just shows a snapshot from the final result at the end point where the combination strategies have reached 100% coverage. The rate of the coverage growth during the test case generation process gets ignored, though it plays a big role in recognition of profitable combination strategies. In practice, a test suite is more preferable if it reaches higher coverage faster than the others. We formalized this goal by introducing a measure called "*Percentage Uncovered Tuples Extension* (PUTE)". For a test set $S$ with strength $t$ and size $n$, we define PUTE as the summation of the uncovered percentage of the $t$-tuples over all of the test frames $TF_i$ included in the test set:

$$PUTE(S) = \sum_{i=1}^{n} U(TF_i) \tag{4.4}$$

Where $U(TF_i)$ is the percentage of the uncovered $t$-tuples after adding the $i$th test frame. PUTE can get any value greater than zero while lower PUTE numbers mean faster (better) coverage growth rates. PUTE is similar to AFPD metric defined in [46] which is a measure of how rapidly a prioritized test suite detects faults.

To illustrate this measure, consider an example test set with 3 variables of size 2 and three test suites, $S1$, $S2$ and $S3$, generated by different algorithms on this test set in Table 4.2. These three test suites provide full 2-wise coverage on the values of the variables in the test set in different ways. Figure 4.5 shows the




(a) PUTE= 150 (b) PUTE= 175 (c) PUTE= 225

Figure 4.5: An example of 3 test suites, illustrating the PUTE measure.

| **Variable** | **Values** |
|---|---|
| $Var_1$ | a, b |
| $Var_2$ | A, B |
| $Var_3$ | 1, 2 |

| **Test Suite** | **Test Frames** | **Percentage Coverage** | **Percentage Uncovered** |
|---|---|---|---|
| $S1$ | a A 1 | 25% | 75% |
| | a B 2 | 50% | 50% |
| | b A 2 | 75% | 25% |
| | b B 1 | 100% | 0% |
| $S2$ | a A 1 | 25% | 75% |
| | a B 2 | 50% | 50% |
| | b A 2 | 75% | 25% |
| | b B 2 | 83% | 17% |
| | b A 1 | 92% | 8% |
| | b B 1 | 100% | 0% |
| $S3$ | a A 1 | 25% | 75% |
| | a A 2 | 42% | 58% |
| | a B 1 | 58% | 42% |
| | a B 2 | 67% | 33% |
| | b A 1 | 83% | 17% |
| | b B 2 | 100% | 0% |

Table 4.2: An example test set and 3 test suites which provide 100% 2-wise coverage on the values of the variables in the test set.

percentage of the covered tuples versus the number of test frames generated by each test suite. For example in Figure 4.5a, the first test frame covers 3 of the 2-tuples, producing 25 percent 2-wise coverage. The second test frame covers 3 more tuples and add another 25 percent to whole coverage resulting in 50% 2-wise coverage. In Figure 4.5, the area inside the inscribed rectangles (dashed boxes) represents the percentage of tuples covered over the corresponding number of test frames of the test suite. The solid lines connecting the corners of the inscribed rectangles interpolate the gain in the percentage of tuples covered. The area above the curve thus represents the the summation of the uncovered percentage of the 2-tuples over all of the test frames included in the test set. This area is the test suite's percentage uncovered tuples extension (PUTE). The first test suite reaches full coverage by just 4 test frames while the other two test suites include 6 test frames. Although $S2$ and $S3$ have the same number of test frames included, $S2$ is more preferable, because it reaches the higher coverage faster. The PUTE measure gives us the ranking we need by reflecting lower values for better solutions and we can use it for comparison of combination strategies even if they have the same number of test frames in their final result.

### 4.4.2 Experimental Comparison with Other Algorithms

Various combination strategies have been developed and deployed for combinatorial test case generation. Here, in this section we compare our PSO technique with some of these existing algorithms. Table 4.3 summarizes the selected algorithms and the settings that we have used for executing them along with a list of references for each algorithm. These techniques belong to different categories. For example, ACA, GA and PSO are population-based search algorithms while SA and TS are trajectory search methods. AETG and IPO are also 2 other well-known greedy techniques which are included in our empirical study. We have also included Random Testing method to see how much other techniques perform above random. All these algorithms are implemented as part of our test case generation framework.

For our proposed PSO method, we used some of the settings suggested in [59] and [34] as shown in Table 4.3 and executed the algorithm in asynchronous update mode, with a neighborhood size of 10.

Our experiments are done using the 171 test objects which are generated randomly by our random test set generator. In order to acquire results with sufficient statistical significance, all our experiments were repeated 5 times. The comparison of algorithms is in terms of 2-wise coverage.

| **Common Settings** | |
|---|---|
| **Parameter** | **Value** |
| Coverage Criteria | 2-wise Coverage |
| Stale Period | 10 |
| Max No. of Evaluations | 1200 |
| **Parameter** | **Value** |
| No. of Particles | 40 |
| Inertia Weight ($\omega$) | Linearly decreased from 0.9 to 0.4 |
| Acceleration Coefficients $(c, c')$ | 1.49445 |
| Max Velocity Dimension $V^i_{max}$ | $v_i/2$ |
| Boundary Condition | Cyclic Walls |
| *pBest* Update Mode | Asynchronous |
| Neighborhood Size | 10 |
| **Parameter** | **Value** |
| No. of Ants | 20 |
| Initial Pheromone | 0.4 |
| Update Factor | 0.01 |
| Decay Factor | 0.5 |
| **Parameter** | **Value** |
| Population Size | 25 |
| Elite Size | 1 |
| Selection Probability | 0.8 |
| Crossover Probability | 0.75 |
| Mutation Probability | 0.03 |
| Massive Mutation Probability | 0.25 |
| **Parameter** | **Value** |
| Initial Temperature | 1000 |
| Final Temperature | 0.01 |
| Temperature Reduction Factor | 0.85 |
| No. of Iterations per Temperature | 100 |
| **Parameter** | **Value** |
| Tabu Length | 15 |
| **Parameter** | **Value** |
| Repeat | 50 |
| **Random Testing** | |

Table 4.3: Selected Combination Strategies and their Settings

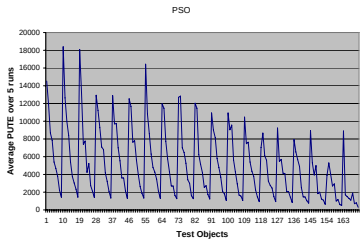
Figure 4.6 reflects the results of our experiments. It can be seen that PSO algorithm produces test suites at least as effective as those produced by other algorithms.

## 4.5 Summary

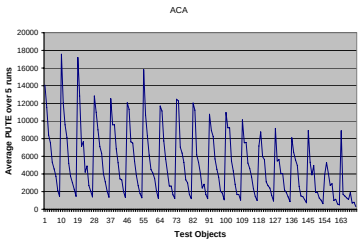
Particle Swarm Optimization (PSO) is a relatively recent heuristic search method that is based on the idea of collaborative behavior of animals living in swarms. In this chapter, we proposed a new test case generation algorithm for $t$-wise testing based on particle swarm optimization technique. Other contributions of this work are as follows:

- We proposed a simple boundary condition, called cyclic walls, for PSO that can be used for solving the problems which have a finite search space.
- We developed a framework for automatic comparison of different test case generation algorithms.
- We introduced a new test case generation metric for assessing the effectiveness of the test suites, generated by different combination strategies.
- We illustrated through empirical experiments that PSO can be as effective as other existing techniques for combinatorial test case generation.

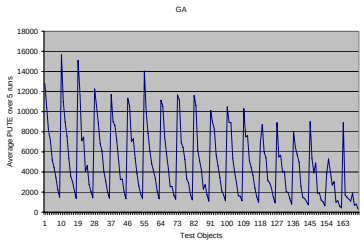
We would like to note that more experiments with further test objects taken from various application domains must be carried out in order to be able to make more general statements about the relative performance of particle swarm optimization and other techniques when applied to software testing. Systematically varying the algorithm settings for the experiments would also help to draw more comprehensive conclusions.



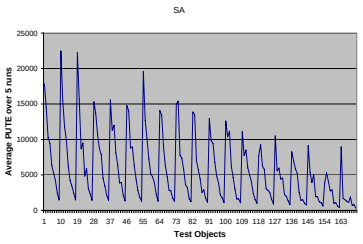

(a) Particle Swarm Optimization



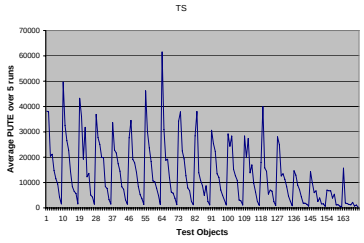

(b) Ant Colony Algorithm



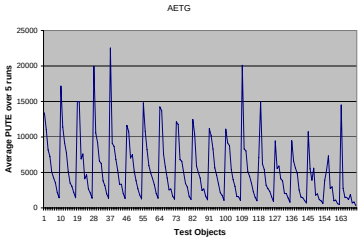

(c) Genetic Algorithms



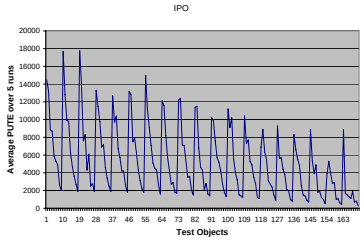

(d) Simulated Annealing



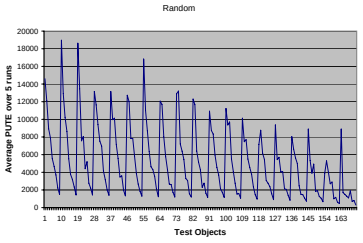

(e) Tabu Search




(f) Automatic Efficient Test Case Generator




(g) In Parameter Order




(h) Random Testing

Figure 4.6: Comparison of PSO with other existing test case generation algorithms.

# Chapter 5

# Conclusion and Future Work

In this chapter, we summarize the findings of the thesis and outline future directions that could be pursued from this research. Section 5.1 summarizes the contributions of the work presented in the thesis while Section 5.2 outlines some potential future work to extending this research.

## 5.1 Thesis Contributions

This work addressed two problems. First was the problem of not having a systematic clear approach for using requirements-based testing and partition testing together. We proposed a layered framework for this purpose and showed that how a testing group can use this framework as a guideline for testing any software system against its requirements defined by the requirements group of an organization. The second problem was about effective combinatorial test case generation. We argued that since this is an NP-complete problem, typically artificial intelligence-based algorithms and enumerative and local search methods are used for finding a near-optimal solution. We also proposed a novel algorithm using particle swarm optimization technique and discussed that this method can be as effective as existing techniques and can outperform most of them in average. The major contributions of the framework can be summarized as follows:

- Design and develop a novel layered framework which integrates requirements-based testing and partition testing techniques into one simple and straightforward testing process.

- Design and develop a novel test case generation technique using particle swarm optimization technique. We showed that this technique can outperform other existing methods.
- Propose a simple boundary condition, called cyclic walls, for PSO that can be used for solving the problems which have a finite search space.
- Design and develop a framework for automatic comparison of different test case generation algorithms.
- Propose a new test case generation metric for assessing the effectiveness of the test suites, generated by different combination strategies.
- Set-up empirical experiments which confirm that PSO can be as effective as other existing techniques for combinatorial test case generation.

## 5.2 Future Work

There are numerous ways to extend this research work. Adding more capabilities to our proposed RBPT framework using other testing techniques can be the subject of future research. Some other issues which need more investigation and research are: realizations of the framework and empirical studies on the performance of the framework; using different variations of PSO algorithm; using different parameters for PSO; considering constraints among variables and values for combinatorial testing; and prioritizing test cases according to the available data through RBPT framework.

- **Adding Capabilities to RBPT Framework:** As we mentioned earlier in the thesis, a large list of various software testing methods have been emerged during time. Normally each technique has some advantages over other methods in specific situations. We can use the strengths of different testing techniques and apply them on different layers of our proposed framework to add more capabilities to our testing process.
- **Realizations of the RBPT Framework and Empirical Studies:** Unfortunately lack of required data prevented us to be able to realize our framework and compare its effectiveness against existing approaches. More studies would help us better understand pros and cons of the framework. Future studies

can be conducted on a large set of software systems, ideally from different domains. Large software systems from industry, the likely users of approaches such as this, are good candidates for such experimentation.

- **Using Variations of PSO:** Different variations of PSO have been studied in the literature. Future studies can use these variations with different neighborhood topologies, different boundary conditions and also cooperative and adaptive versions of PSO to conduct more experiments, in hope of reaching a more solid test case generation algorithm.

- **Considering Variable-Value Constraints:** In some cases, there will be conflicts among different variables and values in the input domain of the software under test. A conflict exists when the result of combining two or more values of different variables does not make sense. In their basic forms, combination strategies generate test suites that satisfy the desired coverage without using any semantic information. Hence, invalid test cases may be selected as part of the final test suite. How to handle these conflicts has not been adequately investigated. Future studies can clarify which constraint handling methods work for which combination strategies and how the size of the test suite and efficiency of the algorithms is affected by the constraint handling method.

- **Test Case Prioritization:** One of the major concerns in our RBPT framework is to consider the required data for test case prioritization. Test case prioritization techniques schedule test cases for execution in an order that attempts to maximize some objective functions. A variety of objective functions are applicable such as rate of fault detection. An improved rate of fault detection during regression testing can provide faster feedback on a system under regression test and let debuggers begin their work earlier than might otherwise be possible. Future studies can investigate integration of these prioritization techniques into our RBPT framework. Lots of useful raw data can be gathered in different layers of the framework which can be used as an input for these prioritization techniques.

# References


[1] IEEE standard glossary of software engineering terminology. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00159342, 1990. 10

[2] T. Beielstein, K. E. Parsopoulos, and M. N. Vrahatis. Tuning PSO parameters through sensitivity analysis. *Sonderforschungsbereich (SFB) 531*, 2002. 48

[3] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold/co Wiley, 1984. 1

[4] P. Berander and A. Andrews. *Requirements Prioritization*, pages 69–94. 2005. 13

[5] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional, 1999. 1

[6] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proceedings of the International Conference on Software Testing Analysis and Review*, pages 503–513, 1998. 9, 43

[7] K. Burroughs, A. Jain, and R. L. Erickson. Improved quality of protocol testing through techniques of experimental design. In *Proceedings of the IEEE International Conference on Communications*, pages 745–752 vol.2, 1994. 3

[8] B. Clow and T. White. An evolutionary race: A comparison of genetic algorithms and particle swarm optimization for training neural networks. In *Proceedings of the International Conference on Artificial Intelligence*, volume 2, pages 582–588, 2004. 43

[9] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23:437–444, 1997. 1, 23, 24

[10] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13:83–88, 1996. 3, 23

[11] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the* $25^{th}$ *International Conference on Software Engineering*, pages 38–48, 2003. 18, 44

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001. 16

[13] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the International Conference on Software Engineering*, pages 285–294, 1999. 9

[14] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 26:29–41, 1996. 19

[15] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing: experience report. In *Proceedings of the* $19^{th}$ *International Conference on Software Engineering*, pages 205–215, 1997. 3

[16] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10:438–444, 1984. 8, 9

[17] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings of the* $6^{th}$ *International Symposium on Micro Machine and Human Science*, pages 39–43, 1995. 45

[18] R. C. Eberhart, R. Dobbins, and P. K. Simpson. *Computational Intelligence PC Tools*. Morgan Kaufmann Pub, 1996. 51

[19] F. Glover and F. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997. 22

[20] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15:167–199, 2005. 37, 43

[21] W. J. Gutjahr. Partition testing vs. random testing: the influence of uncertainty. *IEEE Transactions on Software Engineering*, 25:661–674, 1999. 9

[22] D. Hamlet. Foundations of software testing: Dependability theory. In *Foundations of Software Engineering*, pages 128–139, 1994. 10

[23] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16:1402–1411, 1990. 9, 14

[24] R. Hassan, B. Cohanim, and O. Weck. A comparison of particle swarm optimization and the genetic algorithm. In *Proceedings of the* $46^{th}$ *AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, 2005. 43

[25] W. C. Hetzel and B. Hetzel. *The Complete Guide to Software Testing*. John Wiley and Sons, Inc., 1991. 6, 7

[26] R. J. W. Hodgson. Partical swarm optimization applied to the atomic cluster optimization problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 68–73, 2002. 43

[27] T. Huang and A. S. Mohan. A hybrid boundary condition for robust particle swarm optimization. *IEEE Antennas and Wireless Propagation Letters*, 4:112–117, 2005. 49

[28] M. L. Hutcheson. *Software Testing Fundamentals: Methods and Metrics*. Wiley, 1st edition, 2003. 1

[29] C. Kaner. The power of 'what if...' and nine ways to fuel your imagination. *Software Testing and Quality Engineering Magazine*, 5:16–22, 2003. 34, 41

[30] J. Kennedy. Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance. In *Proceedings of the Congress on Evolutionary Computation*, volume 3, pages 1931–1938, 1999. 51

[31] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995. 45

[32] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In *Proceedings of the* $27^{th}$ *Annual NASA Goddard/IEEE Software Engineering Workshop*, pages 91–95, 2002. 3

[33] Y. Lei and K. C. Tai. In-Parameter-Order: A test generation strategy for pairwise testing. In *Proceedings of the* $3^{rd}$ *IEEE International High-Assurance Systems Engineering Symposium*, pages 254–261, 1998. 18, 24, 43

[34] J. J. Liang, A. K. Qin, P. N. Suganthan, and S. Baskar. Comprehensive learning particle swarm optimizer for global optimization of multimodal functions. *IEEE Transactions on Evolutionary Computation*, 10:281–295, 2006. 56

[35] J. Martin. *An Information Systems Manifesto.* Prentice Hall, 1984. viii, 12, 13

[36] G. Mogyorodi. Requirements-based testing: An overview. In *Proceedings of the* $39^{th}$ *International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, pages 286–295, 2001. 11, 12

[37] G. J. Myers. *The Art of Software Testing.* John Wiley and Sons, 1979. 6

[38] K. J. Nurmela and P. R. J. Ostergard. Constructing covering designs by simulated annealing, 1993. 43

[39] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31:676–686, 1988. 15, 31, 32, 35, 36, 41, 42

[40] R. H. J. M. Otten and L. P. P. P. Van Ginneken. *The Annealing Algorithm.* Kluwer Academic Publishers, 1st edition, 1989. 20

[41] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications, 1998. viii, 16, 17

[42] W. E. Perry. *A Standard for Testing Application Software.* Auerbach Publications, 1986. 7

[43] S. C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Proceedings of the* $4^{th}$ *International Software Metrics Symposium*, pages 64–73, 1997. 9

[44] D. J. Richardson and L. A. Clarke. A partition analysis method to increase program reliability. In *Proceedings of the* $5^{th}$ *International Conference on Software Engineering*, pages 244–253, 1981. 14

[45] J. Robinson and Y. Rahmat-Samii. Particle swarm optimization in electromagnetics. *IEEE Transactions on Antennas and Propagation*, 52:397–407, 2004. 49

[46] G. Rothermel, R. H. Untch, C. Chengyun, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27:929–948, 2001. 54

[47] G. Sherwood. Effective testing of factor combinations. In *Proceedings of the* $3^{rd}$ *International Conference on Software Testing, Analysis & Review*, 1994. 26

[48] T. Shiba, T. Tsuchiya, and T. Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Proceedings of the* $28^{th}$ *Annual International Computer Software and Applications Conference*, volume 1, pages 72–77, 2004. 19, 43

[49] S. Siegel. *Object Oriented Software Testing: A Hierarchical Approach*. John Wiley & Sons, 1st edition, 1996. 1, 2

[50] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Pub, 1990. 9

[51] J. Stardom. *Metaheuristics and the search for covering and packing arrays*. PhD thesis, Simon Fraser University, 2001. 22

[52] K. C. Tai and Y. Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28:109–111, 2002. 3, 24

[53] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the* $13^{th}$ *IEEE International Conference on Automated Software Engineering*, pages 285–288, 1998. 1

[54] F. I. Vokolos and E. J. Weyuker. Performance testing of software systems. In *Proceedings of the* $1^{st}$ *International Workshop on Software and Performance*, pages 80–87, 1998. 10

[55] I. Weerd, S. Brinkkemper, R. Nieuwenhuis, J. Versendaal, and L. Bijlsma. Towards a reference framework for software product management. In *Proceedings of the* $14^{th}$ *IEEE International Requirements Engineering Conference*, pages 312–315, 2006. 13

[56] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE-6:236–246, 1980. 14, 42

[57] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Proceedings of the IFIP TC6/WG6.1* $13^{th}$ *International Conference on Testing Communicating Systems: Tools and Techniques*, pages 59–74, 2000. 18

[58] A. W. Williams and R. L. Probert. A measure for component interaction test coverage. In *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, page 304, 2001. 2

[59] A. Windisch, S. Wappler, and J. Wegener. Applying particle swarm optimization to software testing. In *Proceedings of the* $9^{th}$ *Annual Conference on Genetic and Evolutionary Computation*, pages 1121–1128, 2007. 43, 56