Design and Implementation of Interface Testing Technique for Better Selection of COTS Components

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Jagdeep Singh
(Roll No. 801131014)

Under the supervision of:
Dr. Shivani Goel
(Assistant Professor)

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

July 2013

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"Design and Implementation of an Interface Testing Technique for Better Selection of COTS Components"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of ***Dr. Shivani Goel*** and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Jagdeep Singh

(Jagdeep Singh)
Roll No. 801131014

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Shivani Goel.

(Dr. Shivani Goel)
Assistant Professor
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean of Academic Affairs
Thapar University
Patiala

Acknowledgement

First of all I would like to thank the Almighty, who has always guided me to work on the right path of the life. This work would not have been possible without the encouragement and able guidance of my supervisor **Dr. Shivani Goel**. I thank my supervisor for her time, patience, discussions and valuable comments. Her enthusiasm and optimism made this experience both rewarding and enjoyable.

I am equally grateful to **Dr. Maninder Singh**, Associate Professor and Head, Computer Science & Engineering Department, for motivation and inspiration that triggered me for the thesis work. I will be failing in my duty if I don't express my gratitude to **Dr. S. K. Mohapatra**, Senior Professor and Dean of Academic Affairs the University, for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field. I am also thankful to the entire faculty and staff members of Computer Science and Engineering Department for their direct-indirect help, cooperation, love and affection, which made my stay at Thapar University memorable.

Last but not least, I would like to thank my parents for their wonderful love and encouragement, without their blessings none of this would have been possible. I would also like to thank my sister, since they insisted that I should do so. I would also like to thank my close friends for their constant support.

Abstract

Commercial Off The Shelf (COTS) components are third party components that are shipped in binary format. They are not provided with their source code, so making changes to them is always difficult. The user is always stuck up with the decision to select COTS components when there are multiple candidates for same requirement. Our approach is based on interface testing that helps the user to select multiple COTS components and run test cases on it. According to the best score of COTS component, the user can select best candidate among them. Our technique is based on using public interfaces of COTS components, so there is no need of source code. This technique can also be used for understanding behavior of COTS components using technique like fault injection.

Table of Contents

List of Figures

List of Tables

Abbreviations

COTS	Commercial Off The Shelf
BABoK	Business Analyst Body of Knowledge
SRS	Software Requirement Specification
UML	Unified Modeling Language
MS	Mutation Score
BCT	Behavior Capture and Test technology
COM	Component Object Model
CFG	Control Flow Graph
DCOM	Distributed Component Object Model
CORBA	Common Object Request Broker Architecture
ASP	Active Server Pages
SQL	Structured Query Language
DLL	Dynamic Link Library

Chapter 1

Introduction

1.1Background

Within past few years, there has been a remarkable shift in the use of commercial products in software intensive systems. The use of COTS (Commercial Off The Shelf) components in software development has increased to a great extent. Most of the organizations believe that by using COTS components there can be reduction in software development costs [this is because COTS components can be bought or licensed instead of being developed from scratch] and long-term maintenance. In software development, many regarded COTS as a silver bullet, but COTS development came with many not-so-obvious tradeoffs—initial cost and development time can definitely be reduced, but there is a lot of integration efforts associated with using COTS components in software development process[1]. COTS components are usually third party software modules and are provided as binary code; mostly they are not shipped with source code and proper documentation [2]. So the selection of COTS components and predicting their quality has always been a key issue faced by software industry. Lack of both source code and complete specification always hinders the applicability of various conventional and classical testing techniques [3].

1.2Difference between COTS development process and traditional development process

The traditional software development process starts with requirement analysis. This system view is essential when software must interact with other elements such as hardware, people, and databases and this system view is generated in software design phase. The development process in traditional software development e.g. waterfall model includes the following activities:

A. Requirements Gathering and Analysis
B. Software Design
C. Implementation and Integration
D. Testing (Verification and Validation)
E. Deployment (or Installation)
F. Maintenance

A. Requirement Gathering and Analysis: Business requirements are gathered in this phase. This phase is the main focus for the project managers and stake holders. This phase usually includes meetings with managers, stake holders and users in order to understand the requirements like:

- Who is going to use the system?
- How will they use the system?
- What are the inputs that are needed by the system?
- What type of output should be produced by the system?

The above questions are general questions that are answered during the requirements gathering phase. The BABoK (Business Analyst Body of Knowledge) lists various techniques that can be used for gathering requirements like Brainstorming, Document Analysis, Interview, Prototyping, Reverse Engineering and Surveys. At the end of requirement gathering phase, the requirements are also analyzed whether they are valid and technically feasible. The various possibilities of incorporating the requirements in the system to be development are also studied. The output of this phase is a Software Requirement Specification (SRS) document which serves the purpose of guidelines for the next phase of the software development.

B. Design: In this phase the system and software design is prepared from the requirement specifications which were studied in the requirement analysis. System design helps in specifying high level design of the software like overall architecture of the system. A number of UML (Unified Modeling Language) diagrams can be

used for producing system and software design. Software design focuses on four distinct attributes of a program and includes the following questions:

- What type of data structures can be used?
- How will be the overall software architecture of the system look like?
- How the interface should look like?
- What are the procedures (algorithms) that should be used for software development?

The design process translates requirements into a representation of the software that can be used for assessing the quality of system before coding begins. Like requirements, the design is also documented and becomes part of the software configuration.

C. **Implementation / Coding:** On receiving system design documents, the work is divided according to various modules/units and actual coding is done. This phase is the main focus for the developer and requires a lot of technical skills. This phase of the software development life cycle is longest phase and requires a lot of effort. In this phase, actual requirements are converted in working replicas.

D. **Testing:** After the implementation phase is over, the software developed is tested against the requirements to make sure that the product is actually fulfilling the requirements that are gathered during the requirements phase. Number of testing techniques can be applied like integration testing, acceptance testing, and integration testing for accessing the quality of software and find errors and faults.

E. **Deployment:** After successful testing the product is delivered / deployed to the customer so that they can use it. This phase usually includes installation of the product at user site.

F. **Maintenance:** When the customers start using the developed system then the actual problems come up and need to be solved during its life span. This process where the

care is taken for the developed product is known as maintenance. This is a long time process that goes on until the software is either declared retired and support is withdrawn.

The maintenance may be corrective maintenance which means the errors in software products are corrected. There is possibility of addition of new features in the software product, which is called enhancement. If new upgrade is designed for the same product it is called up gradation. In order to adapt a software product to certain changing platform requirements, adaptive maintenance is done. It is shown in Figure 1.1.

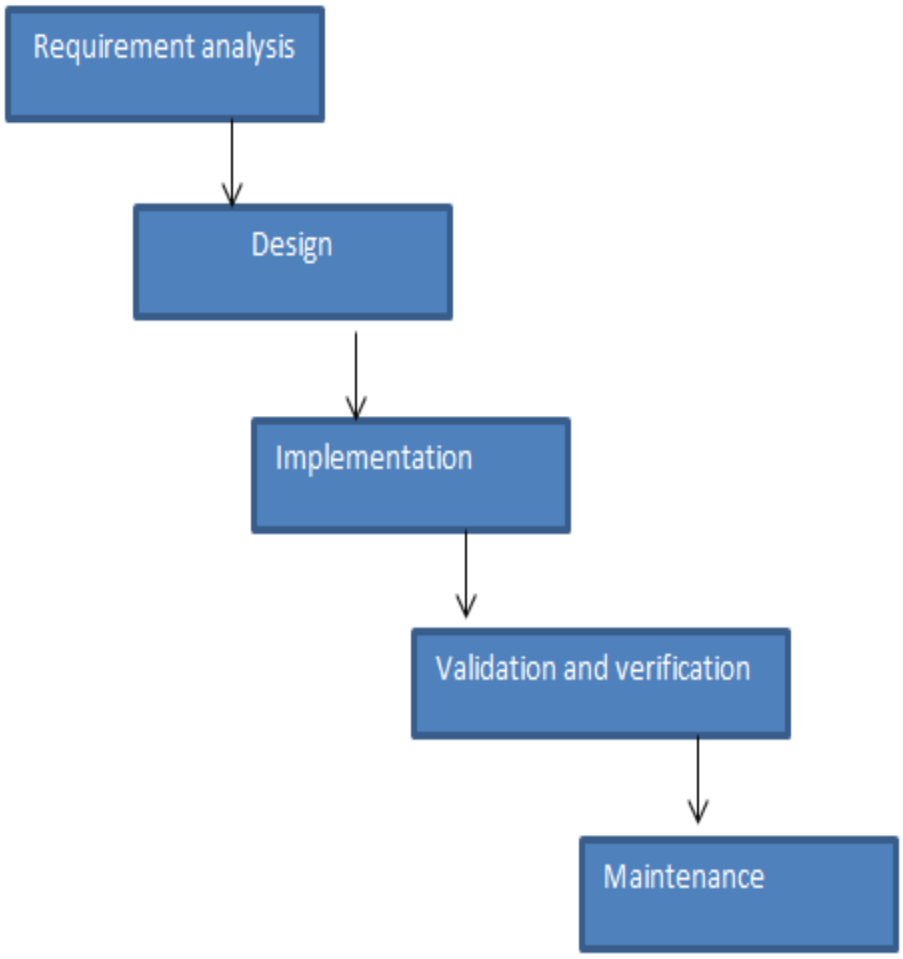


Fig.1.1 Waterfall model [4]

The COTS development is different from the traditional software development. The COTS development process also includes dependence on vendor. All the functionality or features of the COTS components are controlled by the vendor. So customizing the COTS components according to the need has always been a problematic issue for the developer. So a lot of issues are needed to be considered while selecting COTS component. One of these is - how famous is the vendor is in the market is a crucial decision. There is always a fear that if the vendor goes out of the business, then how to handle the COTS components that were bought from the vendor. COTS are basically shipped in binary format, even if the source code is available to the user, the effort required to understand the details and working behind of COTS components is always a problem for the developer.

The development process of COTS includes the following activities:

A. Requirements Analysis
B. Design
C. Coding
D. Integration

A. Requirement Analysis: First of all make versus buy decision is made. If COTS component cost is more than the cost of development from scratch, then it is better to develop it from scratch. COTS selection has been treated as a managerial task than technical task. Technical person should be included in this task, as they can judge COTS selection on technical parameters also, which in turn can improve the quality of software development. Generally all the decisions are made outside project team, and team expertise is also ignored. So later in the project, there is conflict between requirements and COTS functionality.

A.1 Requirement definition

Requirements of the software project are sketched briefly. If the domain of the application is stable then this step can be skipped as requirements are already clear.

B.1 COTS identification and selection

Number of available COTS is evaluated from multiple vendors using vendor documentation and reviews. The main goal of this activity is to reduce number of candidates and select the best candidate.

C.1 COTS familiarization

In this phase the COTS selected are actually used to get familiar with the functionality (not only just claimed). The development process is shown in fig. 1.2

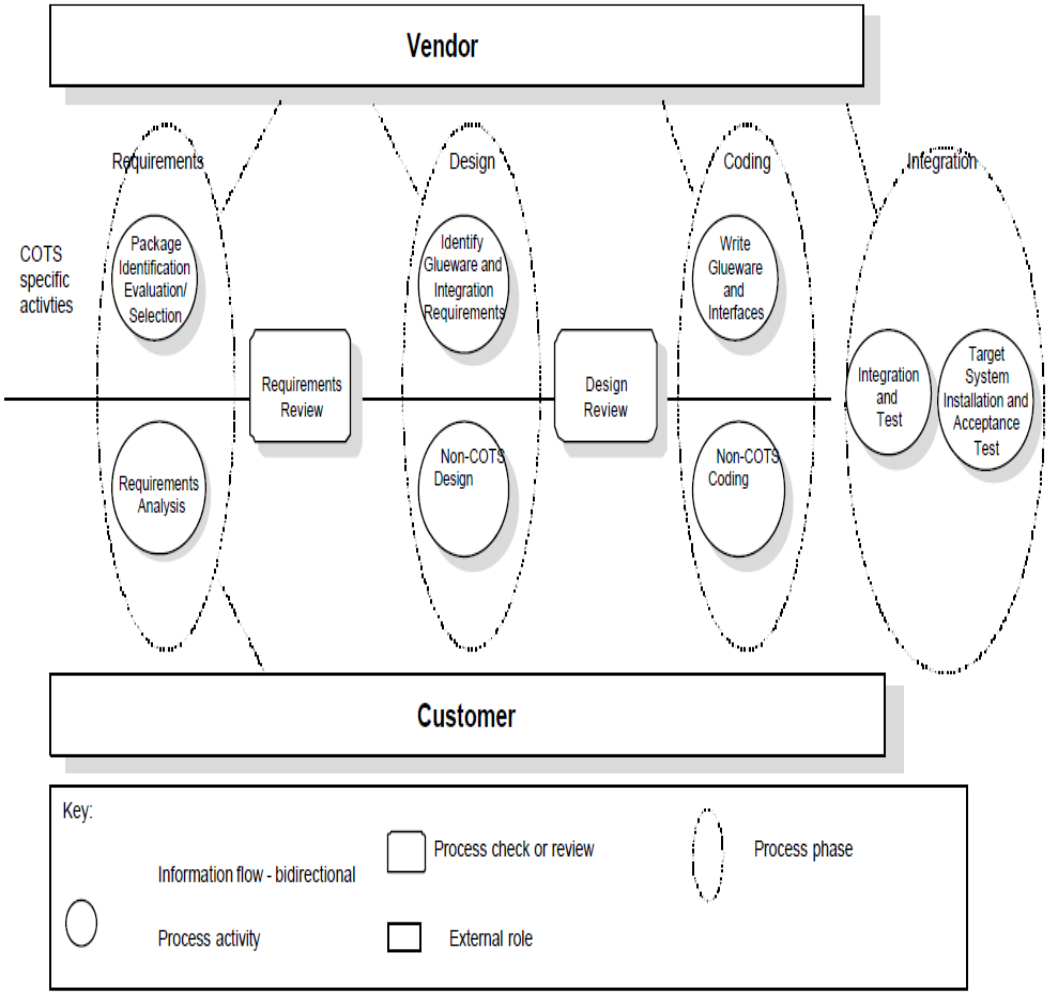


Fig.1.2 COTS development process [7]

B. Design: This phase includes high level design activity and the main focus is on COTS integration and newly developed software. This is particularly demanding when system includes number of COTS each with different architecture style. This phase ends with design review. So now, a better overview of risk involved with COTS integration is gained.

C. **Coding and integration:** Coding involves possibly COTS modification only if source is provided and writing glueware for COTS. Integration puts the pieces together and traditional software to assemble the final application at hand.

1.3 Features of COTS

COTS based products have following features:

- The buyers have no access to the source code.
- The COTS based development introduces the dependence on vendor because all the rights regarding use of COTS, features and customization are legally covered by the vendor.
- The vendor also controls the version, development and its features.
- The COTS component is generic. They are developed to fulfill common requirements across domains.
- The COTS component does not fulfill all the requirements of the user, so the user needs to relax some of its requirements to use the COTS. Even if user requires that COTS fulfill their requirements, COTS needs to be customized.

Some of the important hypothesis regarding COTS- based systems is presented below:

- More than half of the features that are presented in large COTS software products go unused. Most of the features are not used in COTS based systems [5].
- Although glue-code development accounts for less than half the total COTS based software development effort, the effort per line of glue code averages about three times the effort per line of developed-applications code [5].

The table 1.1 shows the effort required for various processes that need to carried on for using COTS. Glue Code requires most of the effort because it is the main process for integration of COTS in the system. Tailoring also needs a lot of effort because customization needs part of COTS to be changed according to the user requirements.

Table 1.1 Effort required for various COTS- directed development activities [5]

Activity	**Average effort (%)**	**± Standard deviation (%)**
Glue code	37	±36
Tailoring	26	±30
Assessment	24	±20
Volatility	13	±11

Some of the important commandment of COTS is as follows:

- Do not believe in silver bullets: COTS is the answer of quick developing application and it is the silver bullet. COTS do not work that way and this concept has been proved wrong [6].
- Understand the impact of integration process: There is a lot of integration effort that has to be spent while integrating COTS in software system.
- Understand the impact on testing process: The testing process of COTS is totally a different process rather than the traditional development process.

1.4 Importance of Testing in COTS

Software testing is the process of determining the quality of software developed by developer. Software testing helps to find the defects in the software. Reliability of the software is an important factor which is determined by testing. As the use of COTS products in software development has increased to a large extend, the need of testing the COTS is of great importance. There are a number of issues associated with COTS. Firstly their source code is not available because vendor always ships COTS in binary format. Even if source code is available, their documentation is very poor. Traditional software testing techniques are not feasible when source code is not available. There are a number of techniques presented by various authors which are capable of testing COTS product

without any need of source code of COTS. Some of famous techniques presented for testing COTS are given in the following section.

1.4.1 Mutation testing

Mutation testing was originally proposed by Richard Lipton as a student in 1971, and first developed and published by DeMillo, Lipton and Sayward [17]. During mutation testing, faults are introduced into a program by creating changes to the program, each of which contains one fault. Test suite is executed on faulty programs. Faulty programs are mutants of the original, and a mutant is killed when a test case causes it to fail. In this way the requirement of identifying a useful test case if satisfied. Mutation testing is all based on fault injection criteria to identify test cased adequacy of test suites.

Calculation of mutation score

Mutation testing is a fault based criteria for evaluating adequacy of test cases. In order to calculate mutation score we have to find number of mutants killed. A mutant is killed if on same input to original and mutated program, mutated program differs. If a mutant is not killed, have the same output like original program and there is no way of differentiating it from original program then it is called equivalent mutant. Higher the mutant score more reliable is test case adequacy of test suite. The mutation score can be defined by the following relation.

Mutation score= $D/(M_c-E_c)$

D= number of killed mutants

M_c= number of contract mutants

E_c= number of equivalent mutants

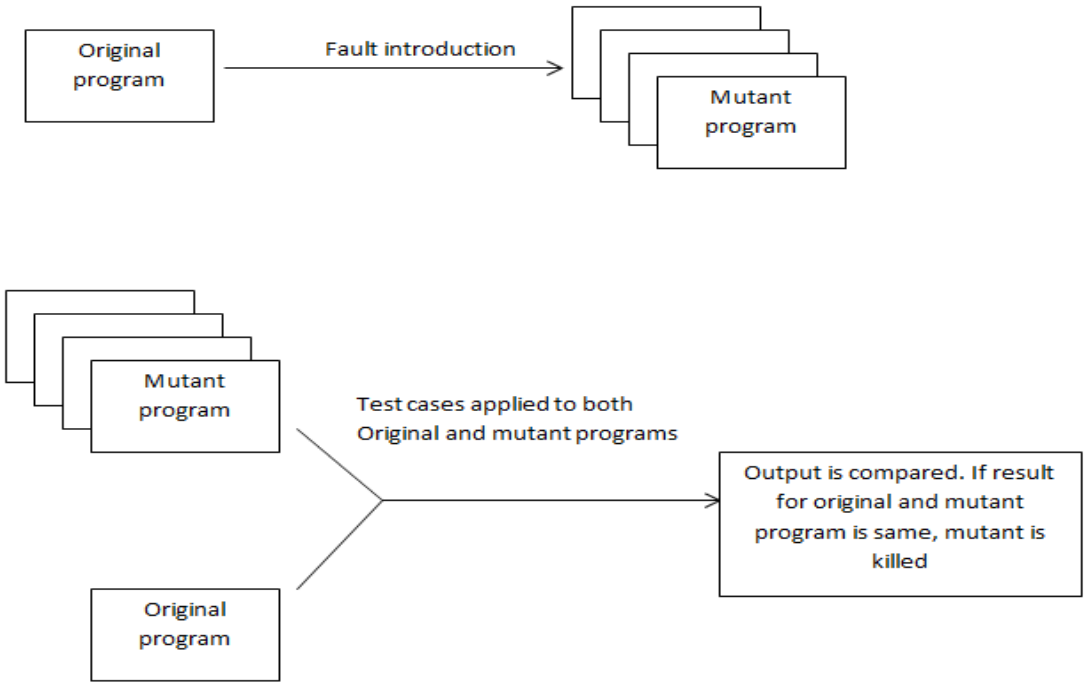


Fig.1.3 Overview of mutation testing

Advantages of Mutation Testing:

- It is a fault injection technique and it is capable of detecting most of the faults that can occur in the system.
- This testing is capable of testing the mutant program and also helps to get most reliable and stable system.

Disadvantages of Mutant testing:

- Mutation testing is costly and takes a lot of time and this testing cannot be done without an automation tool.
- A number of test cases are needed to be run on mutated program. Since there are number of mutant for each original program, testing each mutant is very time consuming
- As this method involves source code changes, it is not at all applicable when source code is not available

1.4.2 Unit testing

The primary goal of unit testing is to take the smallest piece of testable software in the application, and determine whether its behavior is exact as it was expected. Each unit is tested separately in order to find errors and faults before integrating them into modules to test the interfaces between modules. Unit testing has proven its value as a large percentage of defects are identified during its use. The main idea is to develop self-testable COTS components by vendor through which unit test can be embedded into COTS, so that COTS component can be capable of showing behavior to the third part users.

Advantages of unit testing

- *Find problems early:* Unit testing helps the developer to find problems early in the development process. The user can create the graphs of code and see whether all the paths were covered by him.
- *Facilitates changes:* Unit testing allows the developer to refactor code. So if some error occurs later, it can be quickly identified and changed.

1.4.3 Integration testing

A typical software project consists of multiple software modules that are coded by different programmers. Integration testing focuses on testing links between these modules and data communication amongst these modules. Hence it is also termed as **'I & T'** (Integration and Testing). A Module in general is coded by an individual software developer so understanding and programming logic may differ from other programmers. Integration testing is very important in order to verify that these software modules work in unity. Integration test cases differ from other test cases in the sense that it focuses mainly on the interfaces & flow of data/information between the modules. Priority is given to the integrating links rather than the unit functions which are already tested. Integration testing is of great importance as COTS are integrated into the software system. Testing interfaces between COTS and other applications can solve many issues regarding integration of COTS and software system, impact of using COTS to the rest of system.

1.5. Summary

In this chapter concepts related to COTS testing were explained. The difference between traditional software development process and COTS based development process was discussed. COTS development introduces vendor dependence which was a major issue as no source code is shipped with the COTS. Some of software testing techniques which can be applied to COTS were also discussed and will be reviewed in the next chapter.

1.6 Thesis outline

The first chapter briefly describes the background of COTS development process, its features and importance of testing of COTS.

The second chapter covers the literature survey, in which various issues of related to COTS selection, integration and testing are covered.

The third chapter covers the problem definition and the scope of the thesis work. It describes the problem faced by the user while selecting the COTS components.

The fourth chapter covers the work done for design and implementation for technique for use of interface testing for better selection of COTS components.

The fifth chapter includes the results obtained from the experimental procedure.

The sixth chapter includes conclusion and future scope.

Chapter 2

Literature Review

As importance of testing in COTS development is discussed in previous chapter, selecting the appropriate component, with desired capability is of great concern. Numbers of issues are being faced while selecting COTS component to be used in software development as there is no source code available, dependence on vendor and poor documentation. So in this chapter, we will discuss different type of testing techniques that can be applied to testing of COTS. Delamaro et al. have presented mutation-based criteria that can address various issues at integration level [9]. Leonardo et al. have presented compatibility and regression testing for COTS- component based software that tackles the problem of quickly identifying the COTS that are compatible with interface specification [10]. Ying et al. have presented contract based mutation testing that addresses the problems at interface level as components interact with each other and other system with their interfaces [11]. Built in test design for component is good technique for effective maintenance in which component can behave in two modes normal and maintenance mode, and there is a test case script associated with each interface of component[12][13]. Zhenyi and Offbut have presented coupling based criteria for testing of connection at interface level [14]. Haddox et al. have presented an approach that helps to gain an understanding that how a COTS component interact with the rest of the system with the help of a layer called a wrapper that encapsulate the component [15].

2.1 Interface Mutation Testing

During software development, one begins with the development of individual modules that provides functionality to the software system. Components from previously developed product can also be reused. It is assumed that these units are already tested, before these are integrated at system level. During testing of unit one gives one or more inputs and executes the unit. The inputs are referred as test set. There are number of

criteria presented to test how good is a test set is for a given unit. These criteria are known test adequacy criteria. For example dataflow testing [16], mutation testing [17] [18], boundary value analysis [19], control flow testing [20], equivalence class testing provides a variety of adequacy criteria. These criteria's can also be applied for interaction between subsystems.

Interface mutation is proposed to evaluate how well the interaction between modules or subsystem has been tested. This idea was motivated by the need of testing the interaction between units and subsystem as there can be number of faults that may have arisen while integration. Mutation testing is a fault injection criteria proposed by Demillo et al. can be applied at integration level [17].

To access the adequacy of test set T, original program P and each mutant has to be executed according to the test cases in the test T. After execution calculate number of killed mutants, number of alive mutants, and test adequacy can be measured as follows:

$$MS(P,T) = \frac{\text{\#of dead mutants}}{\text{\#total mutants- \# of equivalent mutants}}$$

The technique of mutation testing is very resource consuming, so there is need of reducing number mutants generated during mutation testing [21] [22].

2.1.1 Integration testing

The goal of integration testing is to put the unit into the intended environment and test their interaction with other units and subsystem to find out faults and defects that may arise while integration of subsystem. In this approach all the connection may be tested or some quantitative methods can be used. Various types of techniques have been proposed for testing inter-procedural test set. Haley et al. have classified integration error as computational error and domain integration error [23]. Linnenkugel et al. have defined control flow and data flow-based criteria [24]. Jin and Offut have proposed coupling based testing technique [14].

Interface mutation differs from the above approaches as it does not require the computation of interprocedural associations or interprocedural paths that need to be covered, thus avoiding the complicated techniques required to deal with variable aliasing. The fig. 2.1 shows interconnection between units

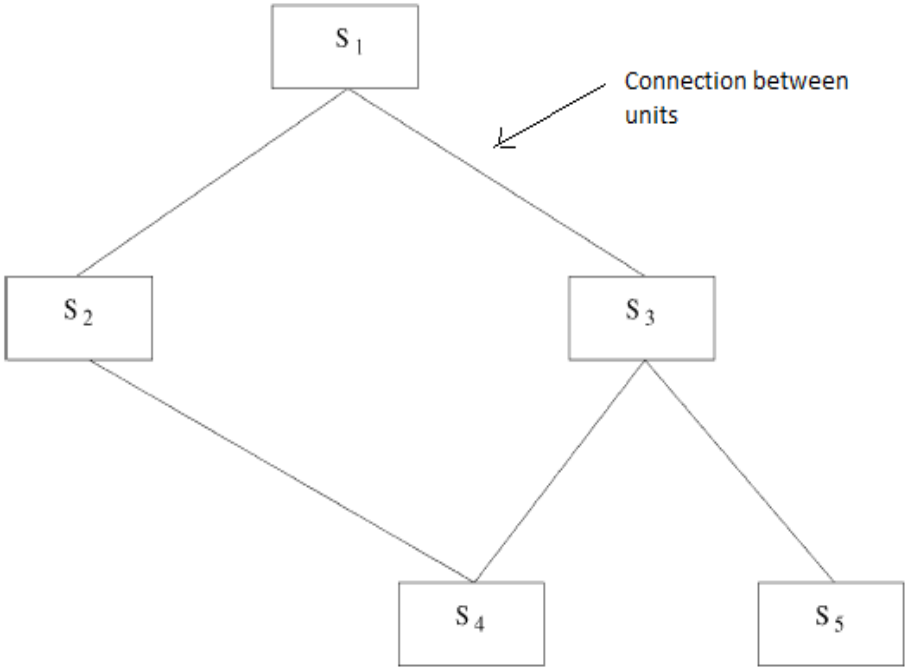


Fig.2.1 Interconnection between units

2.1.2 Integration errors

According to Haley and Zweben [23], an integration error occurs when an incorrect value is passed through a unit connection. So three types of errors has been classified as follows:

Type 1: Upon entering G, $S_I(G)$ does not have the expected values and these values cause an erroneous output (a failure) before returning from G.

Type 2: Upon entering G, $S_I(G)$ does not have the expected values and these values lead to an incorrect $S_O(G)$, which in turn causes an erroneous output (a failure) after returning from G

.

Type 3: Upon entering G, $S_{I}(G)$ has the expected values, but incorrect values in $S_{O}(G)$ are Created inside G and these incorrect values influence an erroneous output (a failure) after returning from G.

The integration error that can occur at during interconnection between units is shown in fig. 2.2

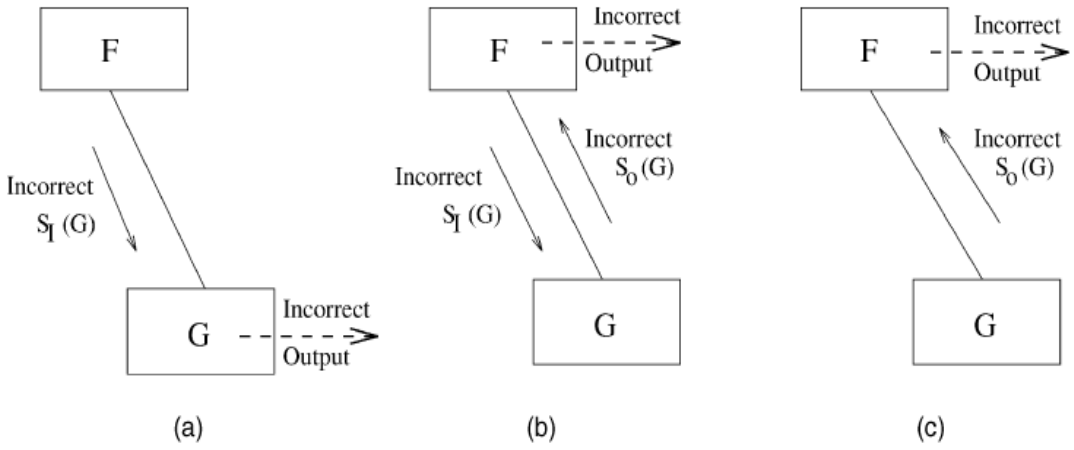


Fig. 2.2 Integration errors [9]

2.1.3 Interface mutation

In order to explain interface mutation, there is a need of understanding nature of data exchange between units. The use of interface mutation helps us to explore the adequacy criteria at integration level. Fig. 2.3 shows the data exchanged between two units. The nature of data exchanged form unit A to unit B can be categorized into four ways.

1. Data can be passed from unit a to unit b via input parameter (pass by value).
2. Data can be passed from unit a to unit b or returned to via input/output parameters (pass by reference).
3. Data can be passed by global variables
4. Data can be passed through return variables

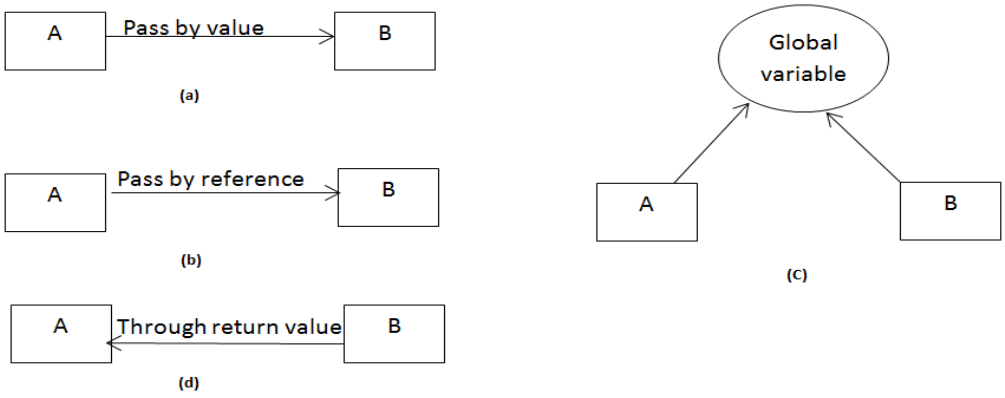


Fig. 2.3 Data exchanged between two units

Mutant operator is dependent on programming language, but same concept can be applied to number of programming languages. The following group can be created for interface mutation testing:

Group I: In this group operators are applied inside the Called Function. Following mutant can be created in this group.

- Direct Variable Replacement Operators
- Indirect Replacement Operators
- Increment and Decrement Operators
- Unary Operator Insertion
- Return Statement Operators
- Coverage Operators

Group II: In this group operators are applied inside the Calling function. Following mutants can be created in this group.

- Argument Replacement
- Argument Switch
- Argument Elimination
- Unary Operator Insertion
- Function Call Deletion

For interconnection of two units F-G, mutants generated are shown in fig 2.3.

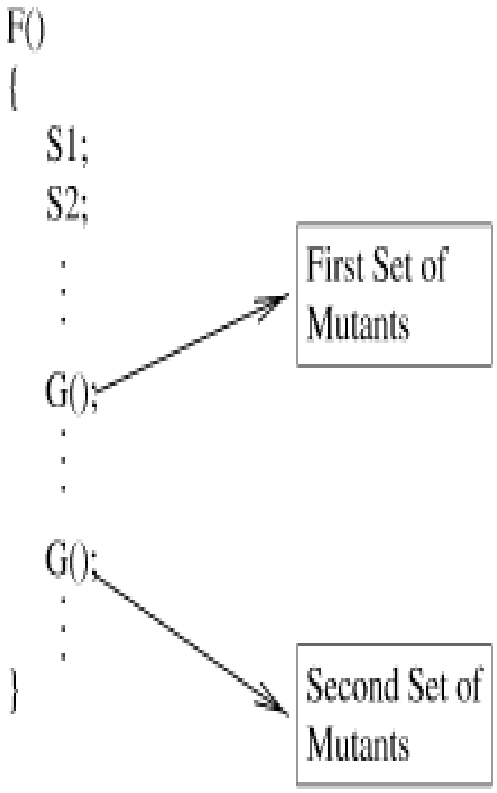


Fig. 2.3 Sets of mutants generated when applying Interface Mutation operators [9]

2.2 Compatibility and Regression testing

The data exchanged between software components and system is of great use to identify potential faults. This technique first generates compatibility and regression test suites based on data exchanged between components and sequence of invocation from component's interfaces. Compatibility test suite helps in identifying and discarding the software components that are compatible with specifications. Regression test suite helps to improve the efficiency of regression testing and helps us to identify the potential problems that can occur at integration. This technique is based on inter- component behavior model, and these model are automatically derived while testing the previous version of software. Fig. 2.4 shows the generation of compatibility and prioritized regression test suite.

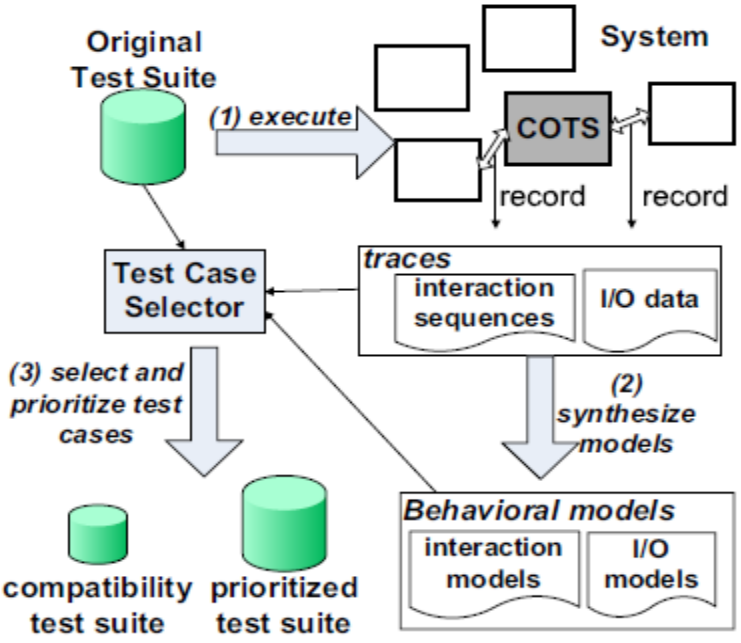


Fig. 2.4 Compatibility and prioritized regression test suites [10].

This technique automatically derives small compatibility test suites to efficiently evaluate several alternative candidate components that can replace a COTS component within a software system, thus helping in selection of COTS components and also helps to create prioritizing test cases to improve the efficiency of regression testing of COTS components, when integrated in new software systems to update obsolete components.

Compatibility test suite and prioritization test suites are generated when testing the original software and the same test suite are used for updating the original program. The fig. 2.5 shows its applicability. The system integrator uses compatibility suite to quickly check the component that are need to be considered and discards incompatible components. After selecting component system integrator executes the prioritized test suite to reveal faults. Executing the test cases in priority has the higher chance of revealing faults early in the system.

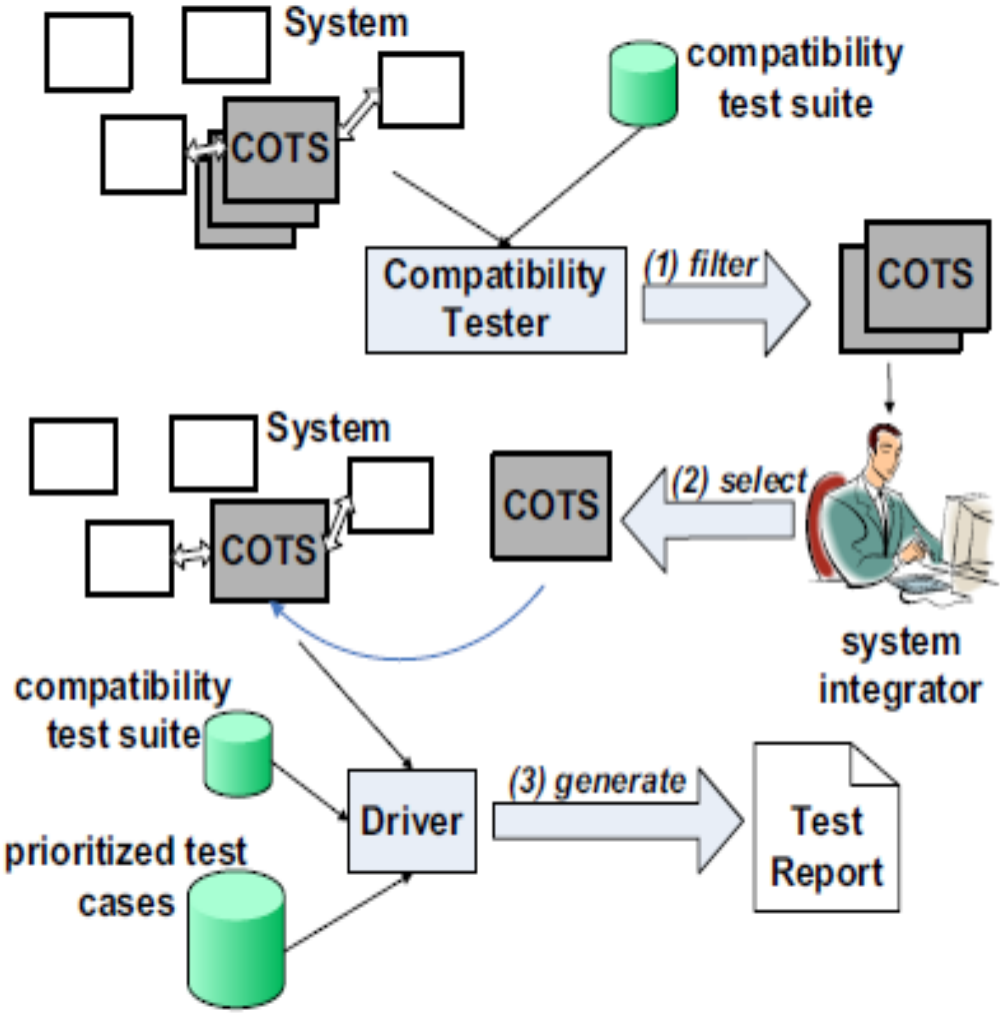


Fig. 2.5 Compatibility and regression test suites in action [10].

Compatibility and regression test suites are generated from behavior models of component and are generated using the following process:

A. Component monitoring
B. Behavior model generation
C. Test case selection

A. Component monitoring:

Components are monitored with the help of suitable infrastructure so that we can trace interactions between components and the system.

B. Behavior model generation:

Behavior models are generated using Behavior Capture and Test technology (BCT). BCT is a dynamic analysis technique that can automate the synthesis of behavior models from execution traces [25]. BCT automatically infers input output and interaction models. Input output models are Boolean expression and interaction models summarizes the interaction sequence that can occur whenever a particular method is executed. This technology produces behavior model that explains two aspects of interaction between components:

1. Sequence of invocation
2. Property of data exchanged between components

C. Test case selection:

Behavior model helps us to group test cases according to the interaction of software component with other system and can be used to select small subset of test cases that represent interaction of system with components. Long time interactions are more likely to expose faults, as small interaction faults are already covered by unit testing [10].

2.2.1 Compatibility test suite

The test cases for compatibility test suite must be minimized to reduce excessive overheads, but should exercise useful test cases that cover interaction between component of interest and its interaction with rest of system. In order to minimize compatibility test suite, I/O model and interaction mode are quite effective. The I/O model and interaction model associated with a component are quite small and can be covered with a small subset of test cases. Fig. 2.6 shows the applicability of model that can be used to cover all the all interaction between components and the rest of system. The incoming interactions are the request from system and output request are from component to the system.

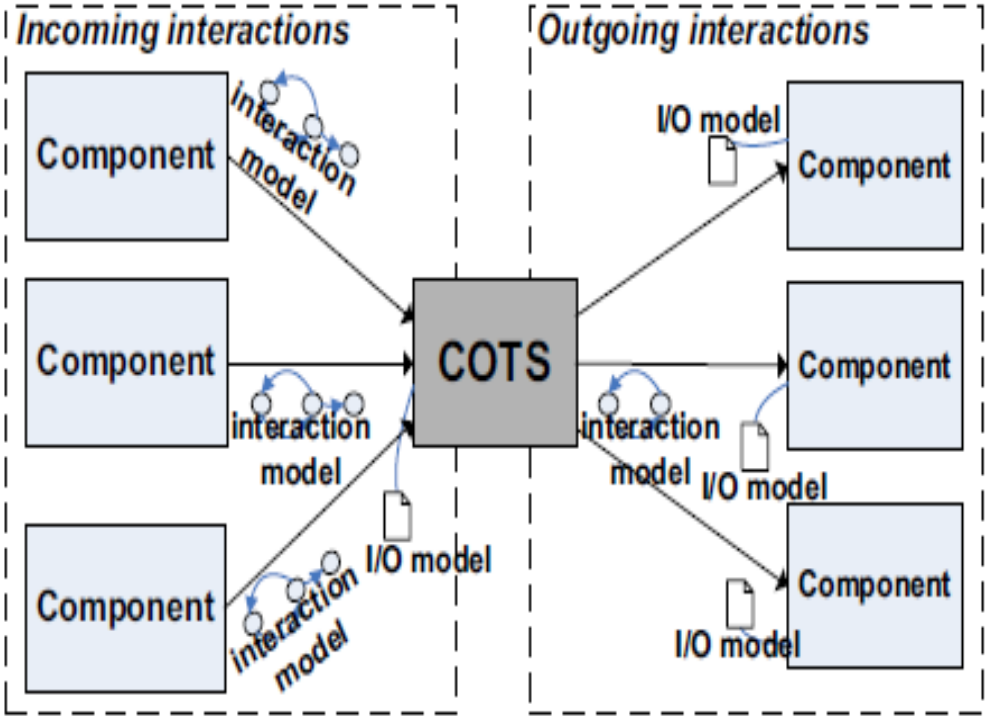
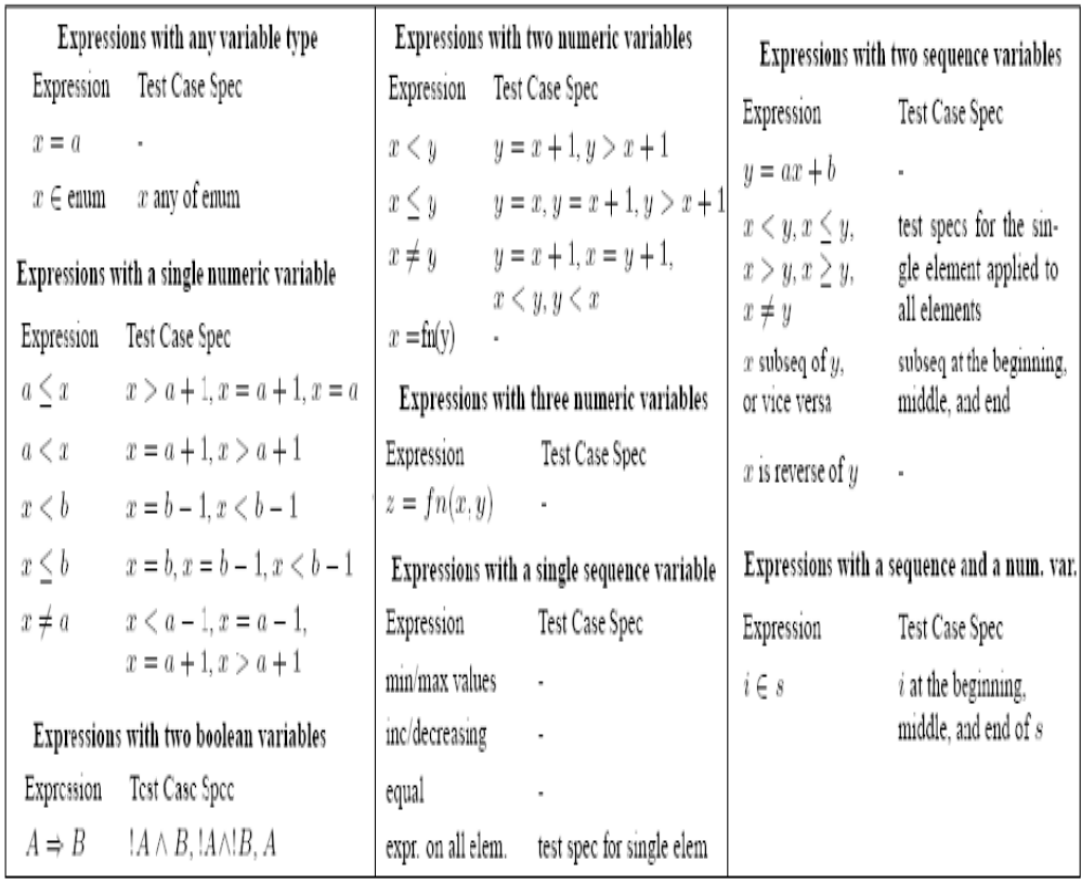


Fig.2.6 Behavior model used for generation of compatibility test suite [10]

Incoming interaction are associated with interaction model of the component and outgoing interaction are associated with outgoing I/O model of the component.

2.2.2 Prioritization of test cases

Test cases should be sorted according to their ability of finding faults. Unfortunately this can be possible by execution of test cases which is not of practical use. So test cases can be prioritized according to interaction with the system. Longer the interactions, more is the priority of the test case. The simple interactions are weighted less. While prioritizing the test cases a number of distinct interaction with the system are also considered.



Expressions with any variable type

Expression	Test Case Spec
$x = a$	-
$x \in$ enum	x any of enum

Expressions with a single numeric variable

Expression	Test Case Spec
$a \leq x$	$x > a+1, x = a+1, x = a$
$a < x$	$x = a+1, x > a+1$
$x < b$	$x = b-1, x < b-1$
$x \leq b$	$x = b, x = b-1, x < b-1$
$x \neq a$	$x < a-1, x = a-1,$ $x = a+1, x > a+1$

Expressions with two boolean variables

Expression	Test Case Spec
$A \Rightarrow B$	$!A \wedge B, !A \wedge !B, A$

Expressions with two numeric variables

Expression	Test Case Spec
$x < y$	$y = x+1, y > x+1$
$x \leq y$	$y = x, y = x+1, y > x+1$
$x \neq y$	$y = x+1, x = y+1,$ $x < y, y < x$
$x =$fn(y)	-

Expressions with three numeric variables

Expression	Test Case Spec
$z = fn(x, y)$	-

Expressions with a single sequence variable

Expression	Test Case Spec
min/max values	-
inc/decreasing	-
equal	-
expr. on all elem.	test spec for single elem

Expressions with two sequence variables

Expression	Test Case Spec
$y = ax + b$	-
$x < y, x \leq y,$ $x > y, x \geq y,$ $x \neq y$	test specs for the single element applied to all elements
x subseq of y, or vice versa	subseq at the beginning, middle, and end
x is reverse of y	-

Expressions with a sequence and a num. var.

Expression	Test Case Spec
$i \in s$	i at the beginning, middle, and end of s

legend: x, y and z indicate names of variables or sequences; a, b and c indicate constants; fn indicates any function; A and B indicate boolean expressions, and – indicates that any test case that covers the variables in the expression covers also the expression.

Fig. 2.7 Coverage of main operand in I/O (Input Output model [10].

2.3 Contract based mutation testing

With the advent of component based software engineering a lot of software are being developed by assembling number of reusable software components. Research has shown that by using reusable component the organization can reduce the development cost and can increase the productivity [27]. The interface is a collection of functions by which application can interact with each other and the system. The strongly typed contracts between software components provide useful set of semantically related operations (methods). These contract are intersting point of analyis for COTS behavior, as the COTS components provides the services through these contracts.

Features of interface: According to Microsoft specification of COM (Component Object Model), interface has the following features [28]:

- is not a class
- is not a component object
- denotes behavior only, not state
- is strongly typed

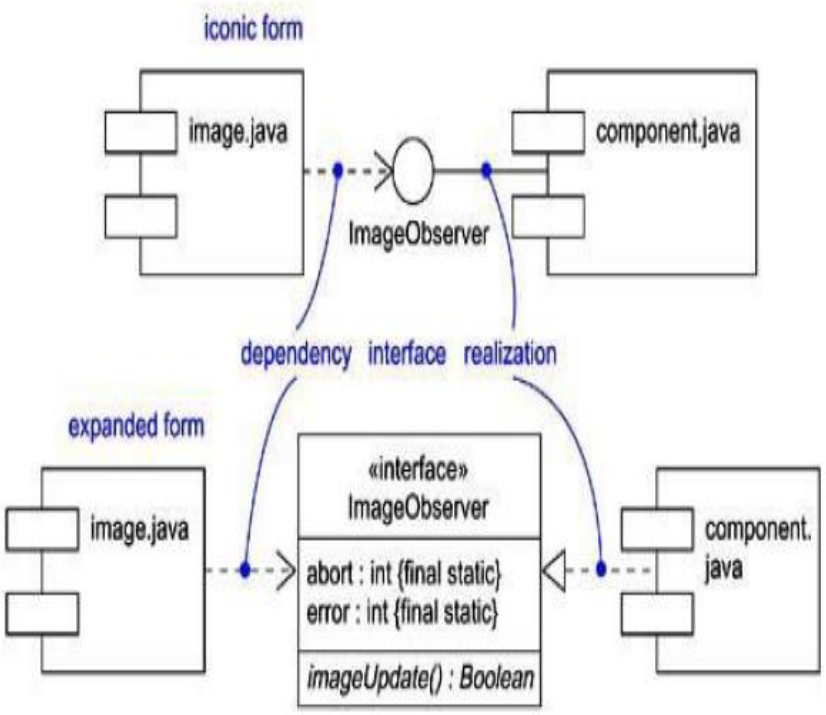


Fig.2.8 Component and interfaces [29]

The basic idea that is associated with this testing technique is to apply mutation testing to the contracts that are provided with the component. As the contracts can be supplied without source code, performing the mutation on the contracts does not depend on the source code of components and there is no dependence on vendor. Design by contract is a method that emphasizes on increasing software testability. Increasing testability means that the software is capable of revealing their faults themselves. The basic idea of design by contract is to establish contract between user of COTS component and the provider. Since the contract describes the behavioral features of COTS, whenever the contracts are violated can be known whenever the COTS component is running. Typically contract

includes the pre-conditions; post conditions, class invariants, loop variants, and loop invariants, etc. The fig. 2.9 shows the typical structure of contracts.

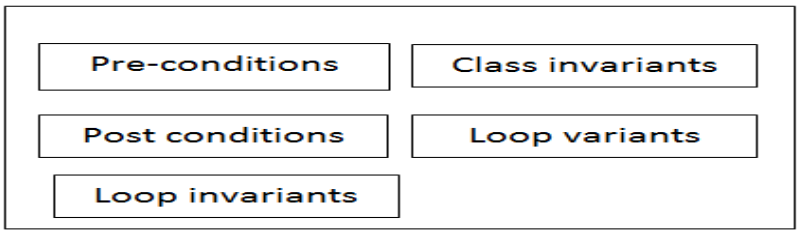


Fig.2.9 Typical structure of contracts

The COTS component provides services through interfaces that can be categorized as:

- *Provided interface:* Through this interface, the component provides the functionality to other components or sub-systems.
- *Required interface:* through this interface, the component takes services from other component or sub-systems.

Provided interface and required interfaces are shown in fig. 2.10.

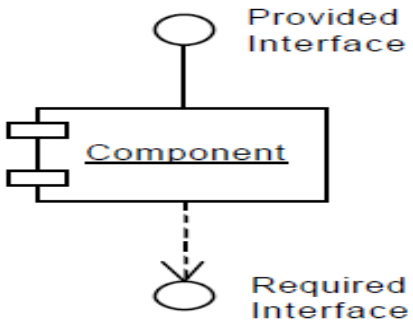


Fig. 2.10 Provided and required interfaces

2.3.1 Contract mutation

The approach of contract based mutations aims at mutating component contracts. It employs a set of high level mutation operators, whose main purpose is to generate fewer mutants without any information loss. A typical structure of interface contract java beans component has been shown in fig.2.11.

```
component ::=
        Component comp-name
                interface-section
        End Component.
interface-section ::=
        Interface
                method-list
        End Interface.
method-list ::=
        method { ; method } ;
method ::=
        method-name [ ( para-list ) ] returns type_specifier
                contract-spec
contract-spec ::=
        /*
        {PreconditionSection}
        {PostconditionSection}
        */
PreconditionSection ::=
        @pre [pre-name:] ContractExpression ;
PostconditionSection ::=
        @post [post-name:] ContractExpression ;
```

Fig. 2.11 Typical structure of contracts in java bean components [11]

A precondition expresses the conditions under which the component interface will function properly if valid inputs are provided. A post condition expresses properties of the results when a component interface has been executed correctly. The mutation operator that can be generated by this approach has been shown in table 2.1.

Table 2.1 Contract mutation operator [11]

Name	**Meaning**
CN	Contract negation
CE	Contract exchange
PW	Precondition weakening
PS	Post condition strengthening
CS	Contract stuck out

Initially test suite for each component can be generated according to the predicates of the interface contracts, using techniques like equivalence class partitioning and boundary value analysis. Secondly, we execute every mutant of the example component with the

initial test suite, and select effective test cases. The experimental procedure of contract mutation has been shown fig. 2.12.

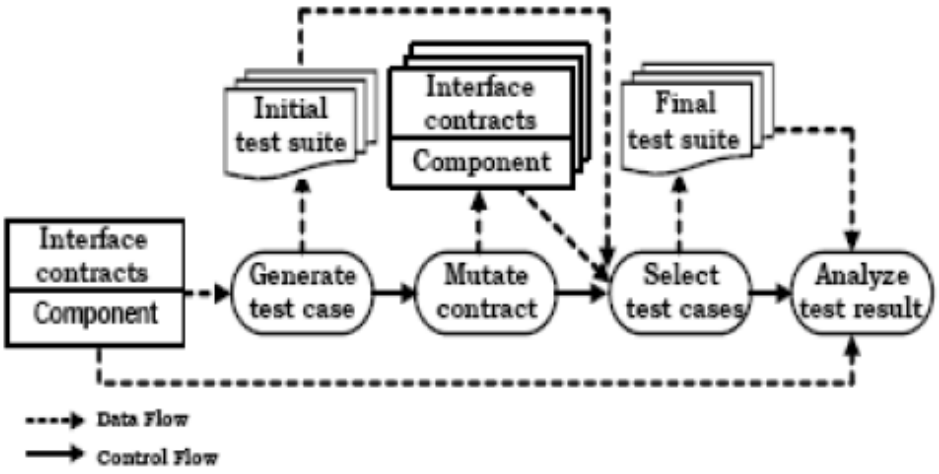


Fig.2.12 Experimental procedure of build-in test components [11]

2.4 Build-in Regression testing for Component- based Software System

Build-in test design is an effective way to improve the testability of Components as components are capable of self-revealing their faults. In this technique the motivating idea was to validate the change and impact of COTS based software, which need the mutual collaboration between software vendor and COTS user. During evolution of Component Based Software Engineering, a portion of software may be replaced, updated with its life cycle, but whenever a software part is updated, it is necessary to retest the whole software again, so there is greater need of regression testing.

2.4.1 Build-in test design

Orso et al. firstly presented the motivating idea of concept of meta-data for component based software engineering (CBSE) [30], and then utilized this meta-data to represent the information exchange between the software components, associated code change and the test cases associated with the software components [31].

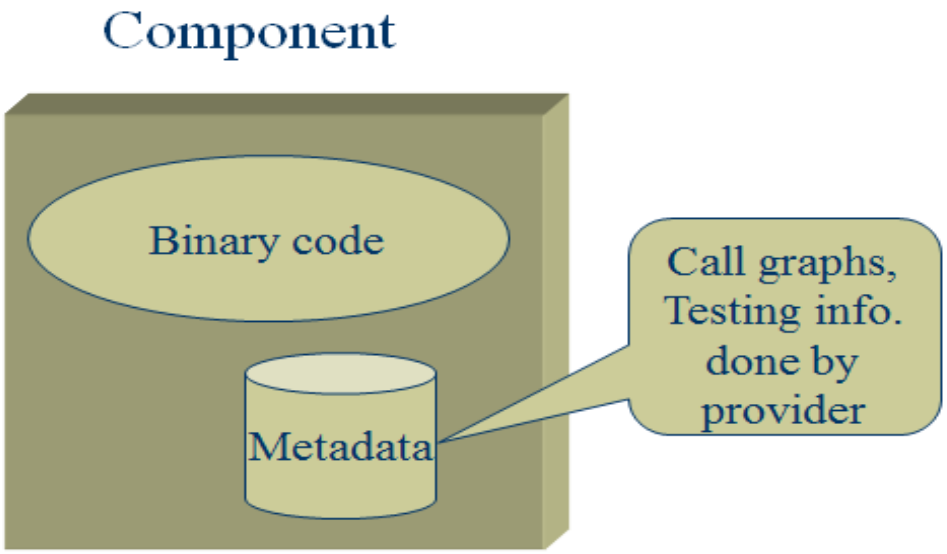


Fig. 2.13 Orso's technique of placing metadata information in software components [30]

Though their method was originally based on build-in test design, but their method used too much detailed information about software components such as their call graphs and testing information. There are also a number of techniques proposed like into-/retro-inspection approach, wrapped based approach [15], for improving the component testability. The built in test design is the most popular because it can be used for single component or component- based system. The fig. 2.14 shows the simple build-in test component.

```
Class BIT_INT_Stack{
   int stack[N];
   int stack_index;
   BIT_INT_Stack();  //The constructor
   ~ BIT_INT_Stack(); //The destructor
   //Member functions in normal mode
   int BITs_Stack_Empty();
   int Pop();
   int Push(int element);
   //Built-in test functions (maintenance mode)
   void BIT_INT_Stack_Test1(...){...};
   void BIT_INT_Stack_Test2(...){...};
}; //A component of integer stack
```

Fig. 2.14 Simple build-in test component [33]

The basic idea of build-in component design is to place test script in the component. In general test scripts include the test case information and possess the facilities for generating the test cases that can be used for testing criteria's. The build-in design component can operate in two modes [32]:

- **Normal mode:** In this mode the component behaves normally like any simple component and provides the functionality that it is intended to provide.
- **Maintenance mode:** In this mode the component is capable of testing itself. The tester case generator produces the test cases so that can it can test the component that can be used for exposing the functionality to the user, or revealing faults.

The modes in which build in design component can operate has been shown in fig .2.15

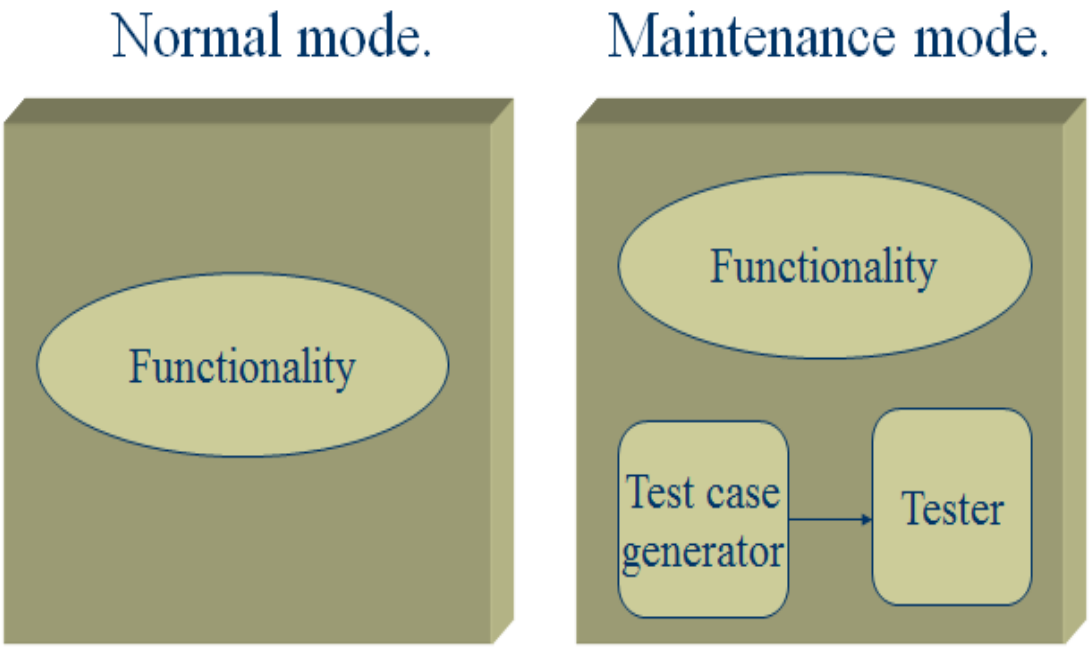


Fig. 2.15 **b**uild-in test design components operation modes [33]

2.4.2 Regression testing process

The implementation process has been shown in fig. 2.16. The need of regression testing arises when component is updated, so the process must work for both system developer and vendor. To facilitate the test case selection, component developer will build testing

interface in the new component. The testing- interface information can be provided with the help of XML files, which directs how to operate the testing interface.

For the system develop, they should place some probes in specific part of system to record the information such as precondition of each published methods. Then they execute the testing interface method through these instrumentation records to select subset of test cases that are related with the change points.

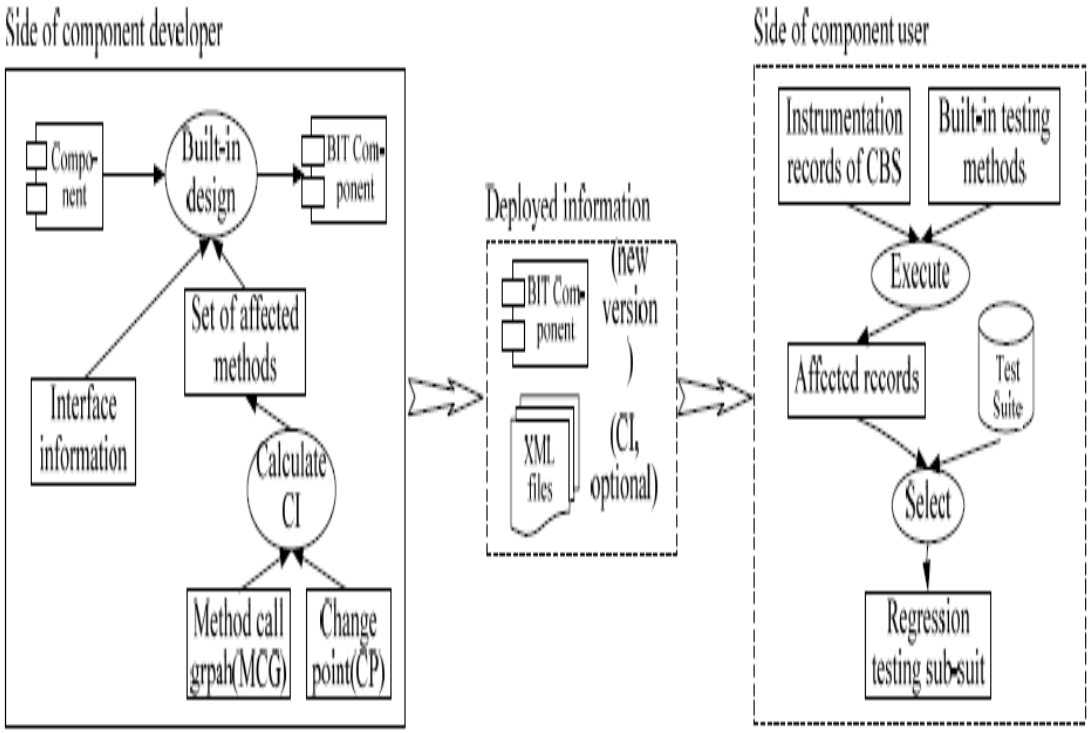


Fig.2.16 Regression testing process based on build in design [33]

2.5 Coupling Based Criteria for Integration Testing

Offbut et al. have proposed an effective and automatable technique for testing of connections between components during integration of software system. A program unit is a contiguous statement having a name that can be invoked by other part of software. A module is a collection of units, files, packages [34]. Module or unit testing is considered as testing of the units as these are mostly independent units. The idea of integration testing is to test interfaces between units, modules to assume that they have consistent

assumption, all information that is transferred between them is correct and integration is correct [35].

Unit testing can also be applied to the integration testing, but they suffer from two problems:

- First it is too expensive to apply unit testing for integration testing
- Secondly unit testing cannot reveal some faults in integration testing because these errors are at the interfaces between modules.

Coupling between two modules measures the dependency between two modules by reflecting interconnection between two modules [36]. Coupling also describes design of the software and its structure. Good software is characterized by high cohesion and low coupling. Greater is the coupling means more interdependence between modules. So in highly coupled system there is greater possibility that the defect in one module will cause fault in other module. Fig. 2.17 shows the coupling and cohesion between units.

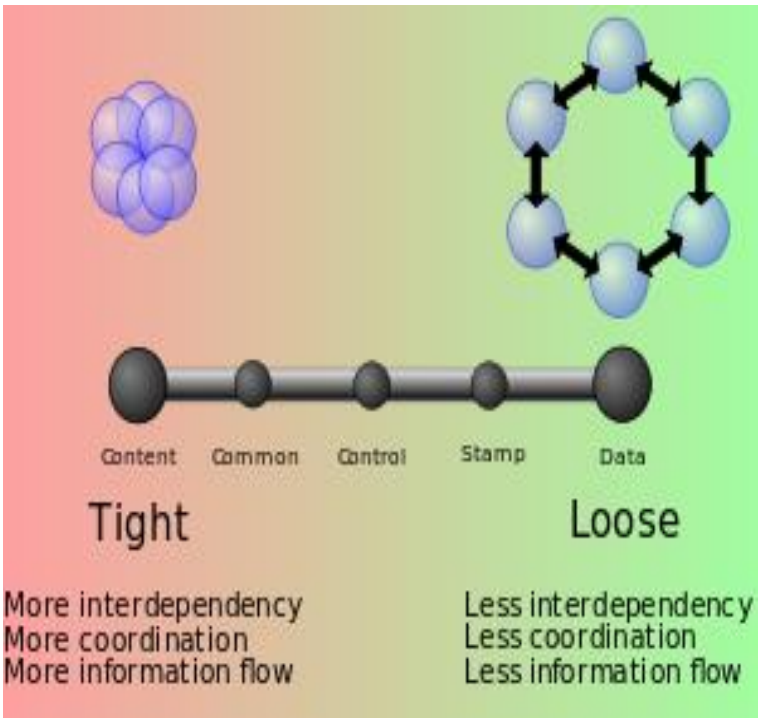


Fig. 2.17 Coupling and cohesion

Offbut et al. have presented 12 levels of coupling that can be categorized into following unordered types:

- *Call coupling:* for module (A) calls module (B) or module (B) calls module (A), but there is no parameter sharing or reference between A and B.
- *Parameter coupling:* This refers to parameter passing. This type includes scalar stamp data coupling, scalar data coupling, stamp control coupling, scalar data/control coupling, scalar control coupling, stamp data/control coupling and tramp coupling
- *Shared data coupling:* this type includes global coupling and non-local coupling.
- *External device coupling:* these include procedures that can access the same external medium.

2.5.1 Basic terminology

Some of the basic terminologies that are used with software testing are listed below:

- Basic block: It is usually a sequence of statement such that any one statement if executed, then all the statements of the block is executed.
- Control flow graph (CFG): It is a directed graph that is associated with the structure of the program. CFG is show in fig. 18.
- Definition (def): It is a location where program is declared in memory
- Computation use (C-use): it is a node where variable in used in computation like arithmetic computations.
- Predicate use (p-use): It is an edge that is used for decision like in testing condition like (if, if else).
- Definition clear path: It is a path for a variable from one node to another where it is not declared again.

Fig. 2.18 shows the control flow graph for a triangle program.

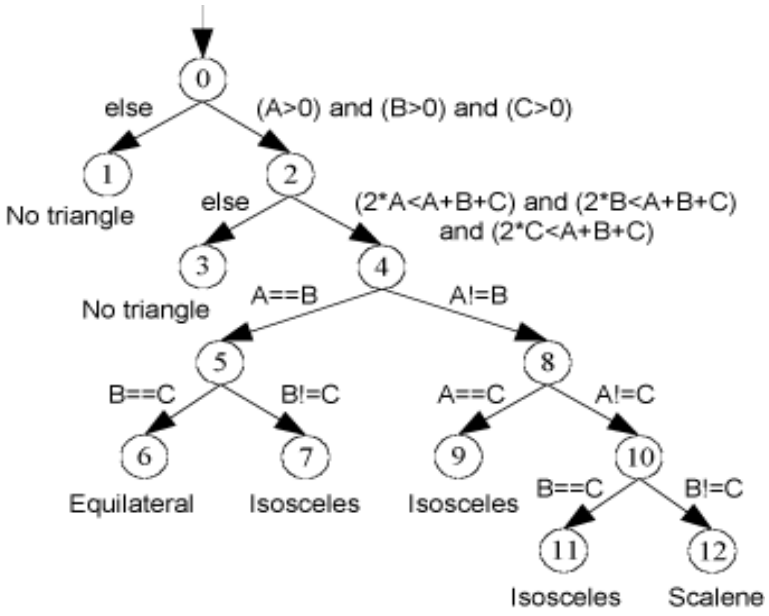


Fig.2.18 Control flow graph for a triangle program

2.5.2 Coupling-based testing definitions

Several definitions are needed to define the coupling-based testing criteria. P is an arbitrary unit in the system, and definitions refer to the units P1, P2, P3 and P4 units.

- P1 calls P2,
- P3 and P4 are units that do not call each other, but they share the global/non-local variable g, and they both refer to the same external device f.

These units and their call relationships are shown in Figure 2.19.

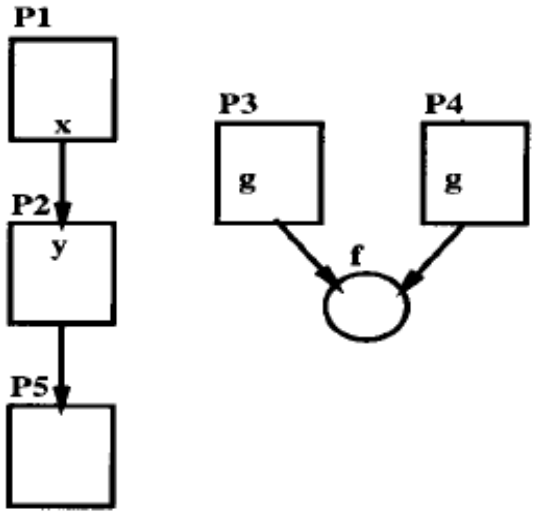


Fig. 2.19 Call graphs for coupling between units [14]

2.6 Software Wrapping Technique

This technique can be used for understanding third party software component or commercial-off-the-shelf (COTS) using notion of software wrapping. Using this technique we can predict how the COTS component will behave with in a particular system. Software wrapping allows data flow in and out at the public interfaces, so that many testing techniques such as fault injection, data collection and assumption checking can be performed whenever the source code is not available.

The use of COTS components in software industry has increased to a great extent as they promise lesser development cost and time. But COTS are not developed for a specific application, so it may not fulfill all the requirements. Further source code of COTS is barely available, still if it is available how the COTS will behave while it is integrated in the system is unpredictable [37]. Software wrapping technique helps us to understand how the COTS component will behave when it is integrated with in the software system.

Some of the problem associated with usage of COTS component is listed below [38], [39]:

- The source code of COTS component is not available, so they can be analyzed as black box only.
- Updates and evolution of COTS components are provided by the software vendor. The new functionality that can be added is also under software vendor
- If the vendor goes out of the business, then then there is no reliability of using their COTS components
- Maintenance can become an issue, as all the features and functionalities are provided by the software vendor. Developers in the organization are forced to make changes to it, which is quite difficult.

2.6.1 High Level Overview of Software Wrapping Technique

Conceptually this approach needs a layer called software wrapper that encapsulate the COTS component. The wrapper is responsible for intercepting the input that is provided to the component and output that is provided by the component. Once the input and output information is recorded variety of testing operation can be applied to COTS

component to provide an improved understanding of the COTS component with the rest of system. The high level overview of software wrapping is shown in fig. 2.20.

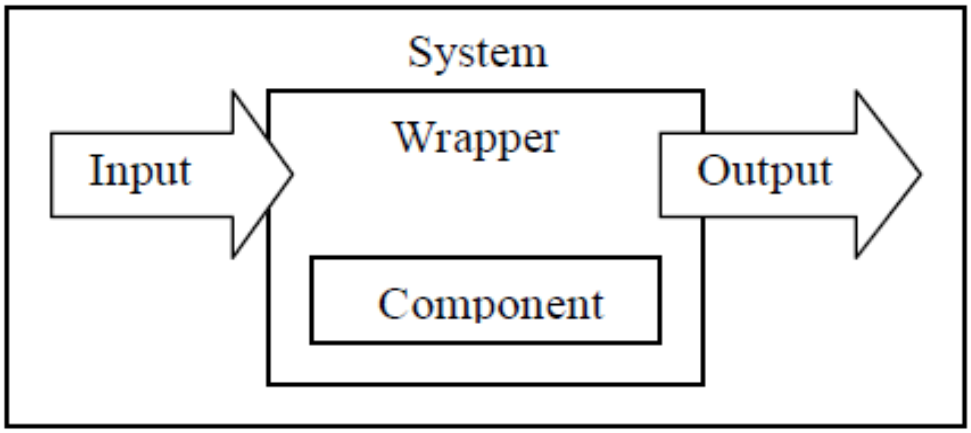


Fig.2.20 Conceptual view of software wrapping [15]

This technique can also be used to predict behavior in a scenario when multiple COTS components are interacting with each other. So using this technique the tester can have an overview of what's going on. Basically fault injection, data collection and assumption checking can be applied using this technique. By fault injection, we mean passing the faulty data to the software component and checking it how it handled the faulty data [40].

2.6.2 Methodology of applying software wrapping

Even if we gain complete understanding of COTS component, still there is value of how the COTS component will behave with buyer system and what type of data will be interchange with system. So there is greater need of understanding the public interfaces that are provided with the COTS. Thus for building the software system using COTS component, we must have idea what type of information will the system provide to the COTS component and what type of information does the system expect to receive form the COTS component. The overview of applying software wrapping is shown in fig. 2.21.

The aim of this approach is to identify the scenarios where the interaction between COTS component and software system causes failure to the system. So first point that a developer must take is to define assertions that how the COTS component will behave with the system that they are part of. During the execution, fault injection can be applied

to the COTS component interfaces in order to understand under what circumstances the COTS components fails.

Fault injection can be applied at two points of interaction of COTS component:

- *Input:* at input the fault injection can be applied to understand the behavior of COTS component that how it behaves when a faulty input is provided to it. So usually we are trying to determine what type of input can cause the COTS component fail.
- *Output:* the fault injection can also be applied at output of COTS component. Using this we try understand what should be the behavior of the system, when COTS component provides us a faulty output.

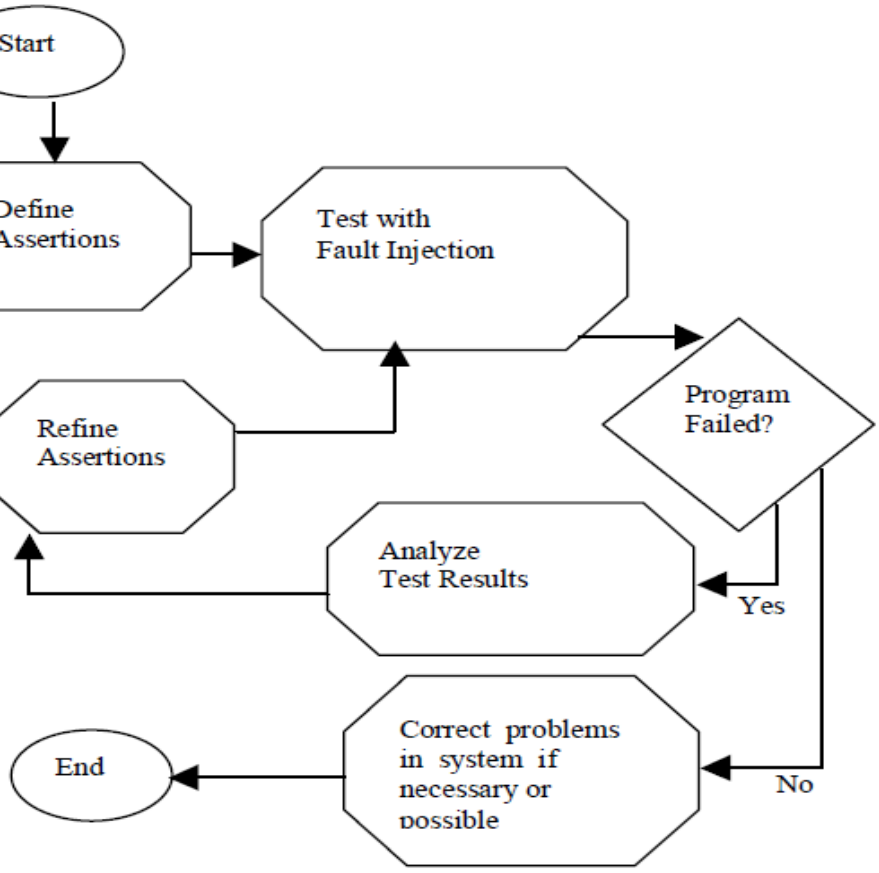


Fig. 2.21 Overview of methodology for software wrapping [15]

Data collection can also be applied during COTS component interaction with the software system, in order to understand the state information of the system and COTS component. By creating assertion and applying fault injection, the developer can gain understanding of how the COTS component behaves with application.

2.6.3 High level overview of wrapping system

The main experimental procedure that has been presented by Haddox et. al is based on java technology [15]. The byte code instrumentation can be used access and modify the bytecode of a java class that is required to be wrapped. This technique uses third party package JavaClass to aid in the byte code instrumentation. The high level overview of this technology is shown in fig. 2.22.

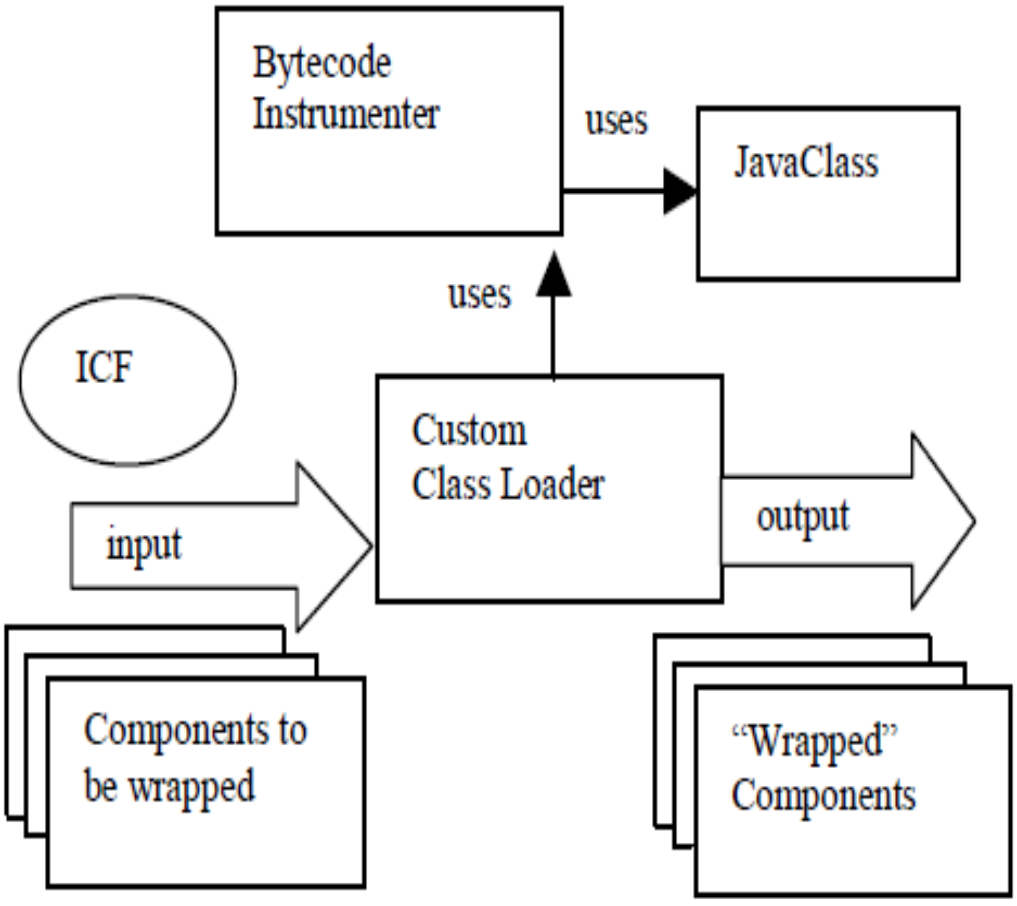


Fig. 2.22 High level overview of wrapper system [15]

2.7 Comparison of Testing Techniques

Black box testing techniques are applied when source code of COTS is not available, i.e. when the COTS products are reused. White box testing techniques are applied when the source code of COTS products is available i.e. when while developing the COTS product. A comparison of various testing techniques with their key advantages and their limitations has been given in table 2.2 which also suggests the testing technique suitable for a particular scenario.

Table 2.2 Comparison of COTS testing techniques

Testing Technique	**Best Suitable for**	**Advantages**	**Limitations**
Software wrapping technique	Blackbox testing of COTS components	No more dependence on vendor for testing of COTS component.	It can record only behaviour of software component based on input given and output received.
Compatibility and regression testing	Regression testing of COTS components	It makes selection of software components by potentially eliminating software components that are not compatible with the specifications.	Regression test suites are time consuming and resource consuming.
Boundary value and equivalence class testing	Best suited when source code of COTS components are available	Boundary value testing helps to identify potential problems that occur at boundaries of data types (like integer ranges 0 to 32767) and equivalence class helps in reducing test case as exhaustive testing is not possible.	Cannot address all the software components testing issues because most of the times source code is not available.
Contract based mutation testing	Integration testing used when no source code of	Capable of testing interfaces of COTS that follow standardized	Just testing the interfaces through which software

	COTS components is available	component models (COM,DCOM,CORBA) through which COTS components communicate and provide services.	components communicates and provides the services does not solve the issue related with using software component when there is no source code available
Interface mutation testing	Integration testing when source code of COTS components is available	Capable of finding faults that can occur when components communicate with each other and the system. It addresses the entire integration problem that might exist when software component is integrated into the system.	This technique addresses the integration issues of software components but software components are still not tested. This technique is dependent on input output models and interaction between components.
Build-in test design	Self-testable components that is capable of self-revealing faults.	Test script in embedded inside the component, so the user of build-in components can see the behavior by using the test cases.	Static nature and does not ensures that the test cases are generated according to the need of user.
Coupling based technique	When there is need of testing at integration level, and units are dependent on each other	Best for testing interconnection between units at integration level	Does not cover every aspect of the system, only suitable for testing integration of system

Chapter 3

Problem statement

The use of COTS components in software industry has increased to a great extent as they promise lesser development cost and time. But COTS are not developed for a specific application, so it may not fulfill all the requirements. Further source code of COTS is barely available, still if it is available, how the COTS will behave while it is integrated in the system is unpredictable. The problem that is generally faced by developers is they cannot identify right candidates from multiple candidates. So the basic task is to identify the problems that are faced by the developer while selecting COTS components when there are multiple candidates.

In order to do so, our first requirement was to provide an interface that can be easily used by the organization for effective selection of COTS component. In this interface, the developer can add test cases they want to run and can perform testing by selecting multiple components with similar behavior. The entire test cases can be prioritized into low, medium and high priorities. The time taken by each test cases run by each component is also recorded to provide a better overview of best component.

The public interfaces provided by COTS component are of great interest. By using these interfaces, one can predict behavior of COTS component without any need of source code. By doing this the developer can have better overview, of how the COTS component handle input and can predict the quality of COTS component.

There are multiple testing techniques that are presented by various authors for testing COTS component without any dependence on vendor. Our approach helps the user to effective select COTS components when there are multiple candidates by running multiple test cases automatically. Thus based on best score, the right candidate can be selected.

Based on the following problems we have set the following objective:

- Identify the problems that are faced by developer when selecting best candidate of COTS component when there are multiple candidates having same requirements.
- Using the public interfaces of COTS component, to expose their functionality to the user when the source code is not available.
- Providing input according to the user and recording output, so that multiple candidates can be compared.
- To provide an effective and easy interface that can be easily used by user.
- Comparing multiple candidate components according to the score based on the test cases that are provided by the user.

Chapter 4

Design and Implementation of an Interface Testing Technique for Better Selection of COTS Components

In this chapter the design and implementation of the interface testing technique is covered that can be easily used by the organization for selection of better COTS component when there are multiple candidates. The interface is written in ASP.NET and we have used SQL Server 2008 as its back end. This chapter covers the feature, architectural design, and implementation of the interface for better selection of COTS component. At the end of this chapter we have shown the working of the system along with some screenshots.

4.1 Features of technique for better selection of COTS components

The main features of proposed system are:

- It should be capable adding the software components in its directory.
- The component should be plugged at run time, whenever a developer selects the component for testing.
- The user should be capable of testing multiple COTS components automatically.
- User can add test cases delete it or modify the test cases at any time according to the requirements.
- Security and authentication of user is an important part of any software, so the interface should be accessible only to those users that are authenticated for using it.
- The interface should be easy to use and quick navigation should be provided.
- The interface should be capable of recording execution time that each test case takes to complete for each component.
- The system should also generate test reports according to the user queries.
- A separate administration unit is also provided so that can delete user, manage the software component storage.

4.2 Architectural design of interface testing technique for better selection of COTS components

In the proposed technique the user can run multiple test cases simultaneously on multiple components. This technique helps the user in selecting best candidates when there are multiple candidates for the same job. The architecture design is shown in fig.4.1.

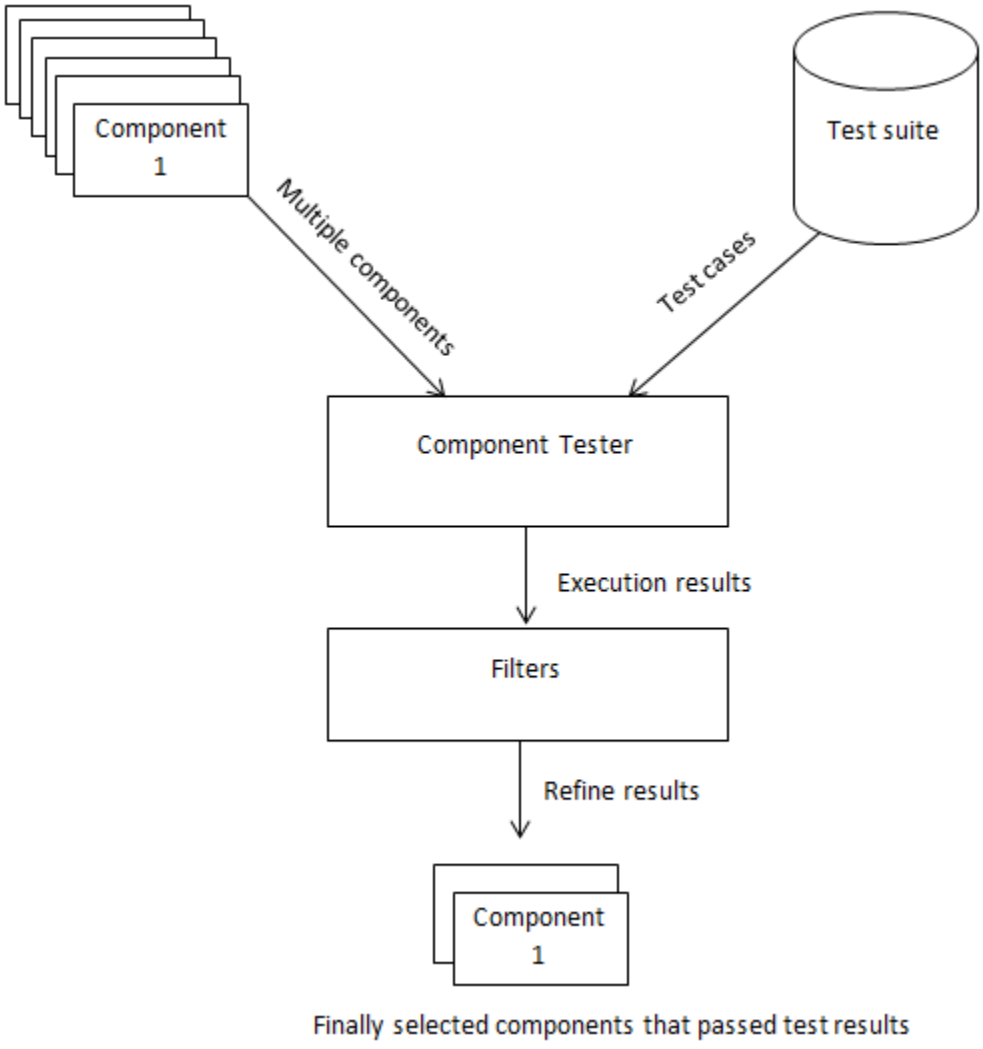


Fig. 4.1 Architectural designs of interface testing for better selection of COTS

The user first selects the software components that fulfill their requirements by query the system using keywords. After the system display that software components that are matched to the query. The user then can select multiple components accordingly. After selection of software components or COTS components, the user is proceeded to add the

test cases according to his requirement. The test cases can be added or deleted at this point. The test cases can also be prioritized accordingly high, medium or low. After the test cases are entered by the user, the system will store the test cases in database. So next phase is component tester in which the system executes each test case on each component and store the result obtain in the database. After getting the results, the user sorts the test cases according to pass and fail, by analysis the output and expected output. So in the filter phase, the user is capable of selecting the best candidate according to the score of each software component.

4.3 Behavioral aspects of interface testing technique for better selection of COTS

The user has to follow the following steps to perform testing of COTS:

Step 1: First gets a session ID from the system before proceeding to Component testing.

Step 2: Enter the query regarding the components present in the system by entering keywords in the search textbox.

Step 3: If component is present in the system, the system will show the component. The user can select single or multiple components for testing and after selecting proceed to step 4. If no component is present, proceed to step 2.

Step 4: Add the test cases according to requirement. Test cases can be deleted or added. The test cases can be prioritized as high, medium and low. If addition of test cases is completed, proceed to step 5 else continue.

Step 5: Execute the test cases to each component and record the output to database.

Step 6: If the test case pool is empty proceed to step7, else proceed to step 5.

Step 7: On the basis of result, create a test report which contains information regarding pass and failure of test cases, and on the basis of the test report select the best candidate among COTS components.

Step 8: If you want to test more components proceed to step 1.

The activity diagram that shows the behavioral aspect of the system is shown in fig. 4.2.

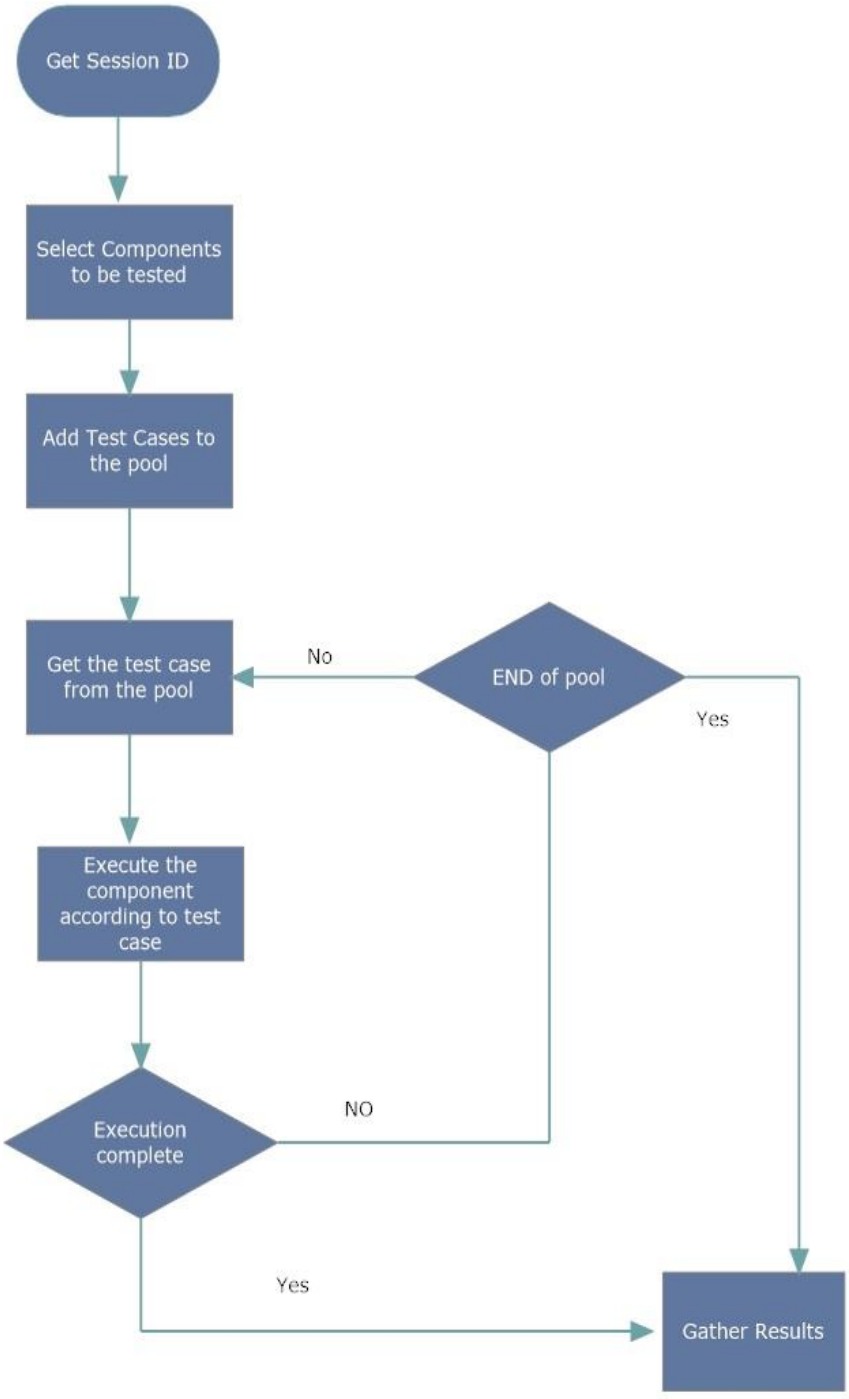


Fig. 4.2 Activity Diagram of Interface Testing Technique

4.4 Database diagram

We have used Microsoft SQL server 2008 as the back end of selection of better COTS. This Database server is very reliable and data can be stored in this securely. For the system we have considered the following tables:

- **Comp**: This table contains information regarding component name, component ID, language and domain. Component ID uniquely identifies each component.
- **Compdetail**: Detail information regarding components is entered in compdetail.
- **Paramlist**: It contains information regarding interface of component. Prototype of public functions is listed in this table.
- **Storedloc**: All the information regarding storage of component in various directories is stored in storedloc.
- **Sessionid**: It is used for session id that is generated automatically by system and it is always unique. This session id uniquely identifies each test suite.
- **Testsuite:** This table contains information regarding test cases that are entered by user. It contains input to be provided and expected output.
- **Results:** It contains the information regarding actual results obtained by executing the test cases against the selected components.
- **Timetaken:** Time taken by each test case over each component is kept in the fields of table timetaken.
- **Report:** The information regarding the failed and passed test cases is stored in table report.
- **User:**The user related information like their username and password that are used for security purposes are kept in this table.

The database diagram of the tables used for development of better selection of COTS system is shown in fig. 2.3.

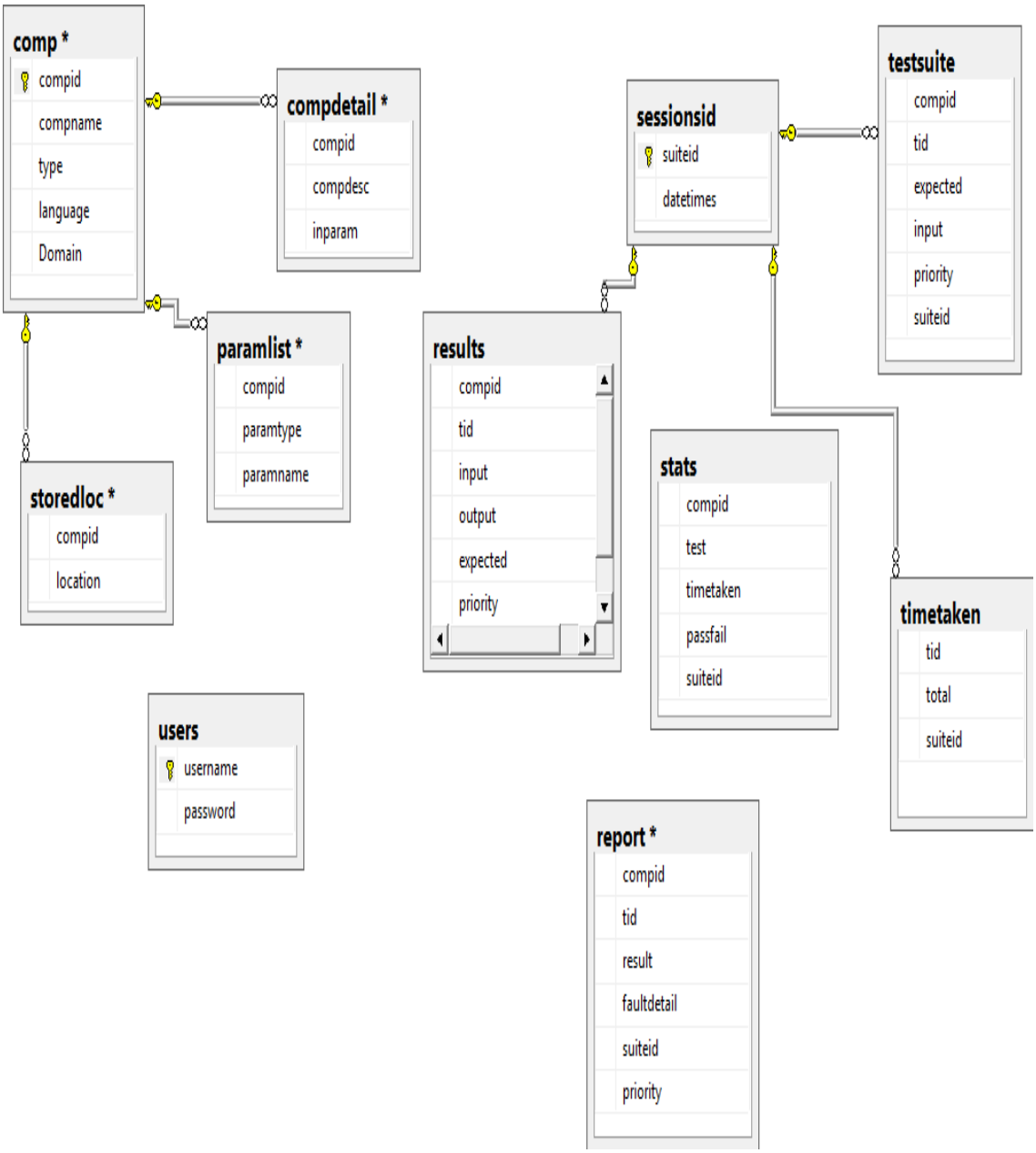


Fig. 4.3 Database diagram for better selection of COTS system.

4.5 Storage of components

Software Components are usually categorized as black box and white box components.

- **Black box components:** These are the components that can be used without carrying out any modification. Usually they are hard to use and take allot of integration effort. But as they are tested and reliable components, they promise of

reduced development cost and time. COTS components are characterized as black box component.

- **White box components:** These are the components that are provided with their source codes. So making changes to them is easy, but in order to do so, one should have greater inner detail of components.

The components that can be stored in the system as black box as well as white box components. Bu we have only considered only black box components which are basically executable and dynamic link library (DLL's) .We have stored the software components in the repository according to the following fields:

- **Component ID:** This field is assigned to each component to be stored in the directory. The component ID assigned to each component is unique.
- **Component name:** In this field the name of component is defined.
- **Component type:** This indicates whether a component is executable, source code or dynamic link library (DLL) component.
- **Component language:** This field represents the language that's used for writing component i.e. C, C++, and Visual basic.
- **Category:** The category contains information regarding the component domain, i.e. Artificial intelligence, Network etc.
- **Component description:** This field contains the detail description of the component that is stored in the directory.
- **Parameter type:** Parameter type contains data types that are used in the interface of the component. This is the most important as according to the parameter test cases are executed on the basis of interface provided information.
- **Parameter name:** This contains the prototype information of the parameter and order highly matters.

The component storage is shown in fig. 4.4.

Fig.4.4 Storage of COTS components

4.6 Working of Better selection of COTS system

The system is developed as web application and its full features, various units and working is explained in this section. The back end of this software is developed using Microsoft SQL server 2008 and front end is ASP.NET. The software is visual studio 2010 and framework 4.0 was used for its development.

4.6.1. Security and Authentication

The feature of any software to prevent it from unauthorized access is an essential feature of good software. So any user that wants to access this software needs to authenticate itself using its username and password. If the username and password does not match as according to the credentials, error message is shown by the system. Fig. 4.5 shows user authentication.

Fig. 4.5 User authentication

4.6.2 Session ID

Session ID is generated automatically by system and it is always unique and used to identify each test suite. Fig. 4.6 shows the session ID generation by the system.

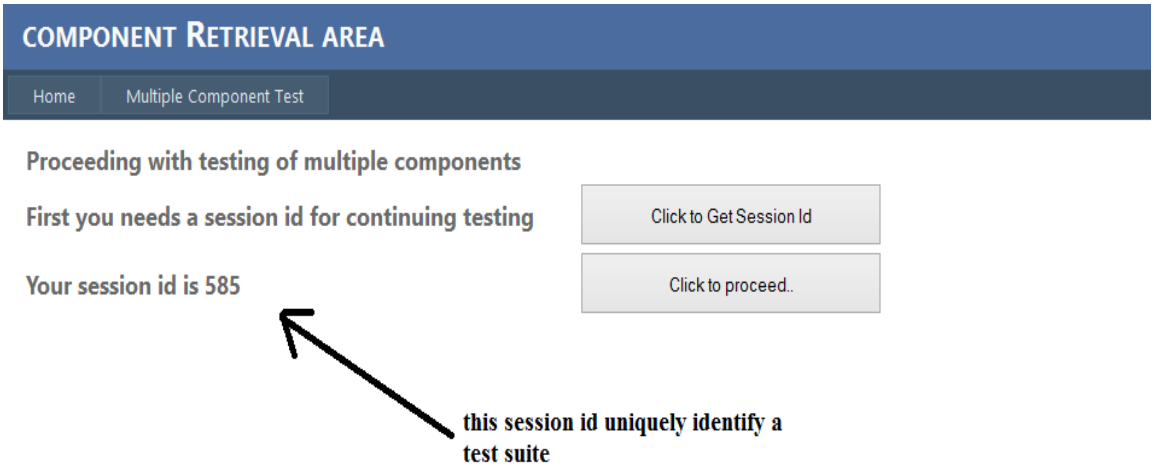


Fig. 4.6 Session ID for unique identification of Test suite

4.6.3 Selecting multiple components for testing

In this phase user can query the system for specific component. To help the user in the selection of component, the system provides keyword based search for components. If no component is present, the system flags error and shows that no such component is present. Currently we are using component name for searching. If multiple components are present, it is shown by the system. Multiple check boxes are shown with each component show that quick selection is possible.

The user has to follow the following step to use this feature

Step 1: Query the system about the component that is required using the query box. Keyword based search currently supported in this system, so the user can use keyword for searching of components.

Step 2: Select the components using the check boxes that are listed with each software component the system has retrieved. If no component is selected and the user select proceed the system will flag error code- No selection made.

Step 3: After selecting single or multiple component, click on add selected component and system will add the components to the pool for testing.

Step 4: Click on proceed to test cases to add test cases according to the requirements.

After doing this properly, the selected components are added to the pool for testing. Care must be taken to add only components that fulfill same requirements and have same prototype to ease testing. This feature can also be used for testing of single component. The working of this feature is shown in fig. 4.7.

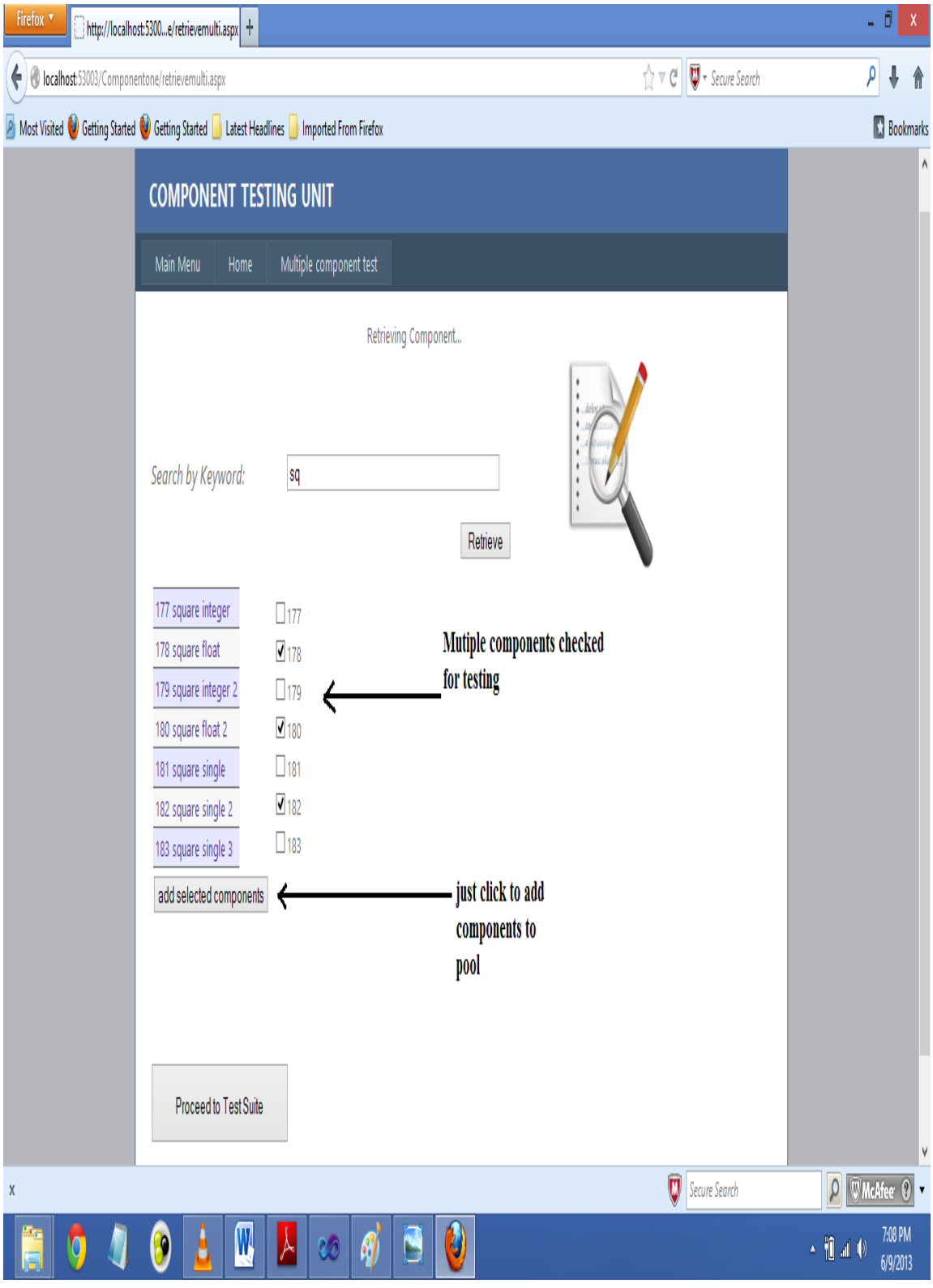


Fig. 4.7 COTS Selection for Testing

4.6.4 Adding test cases

This is the main part of the software. This feature of the software helps the user to add test cases according to its requirement. The test cases can be added, or deleted if needed. The user also inputs the expected output that is expected by the COTS component. The fig. 4.8 shows the working of this feature.

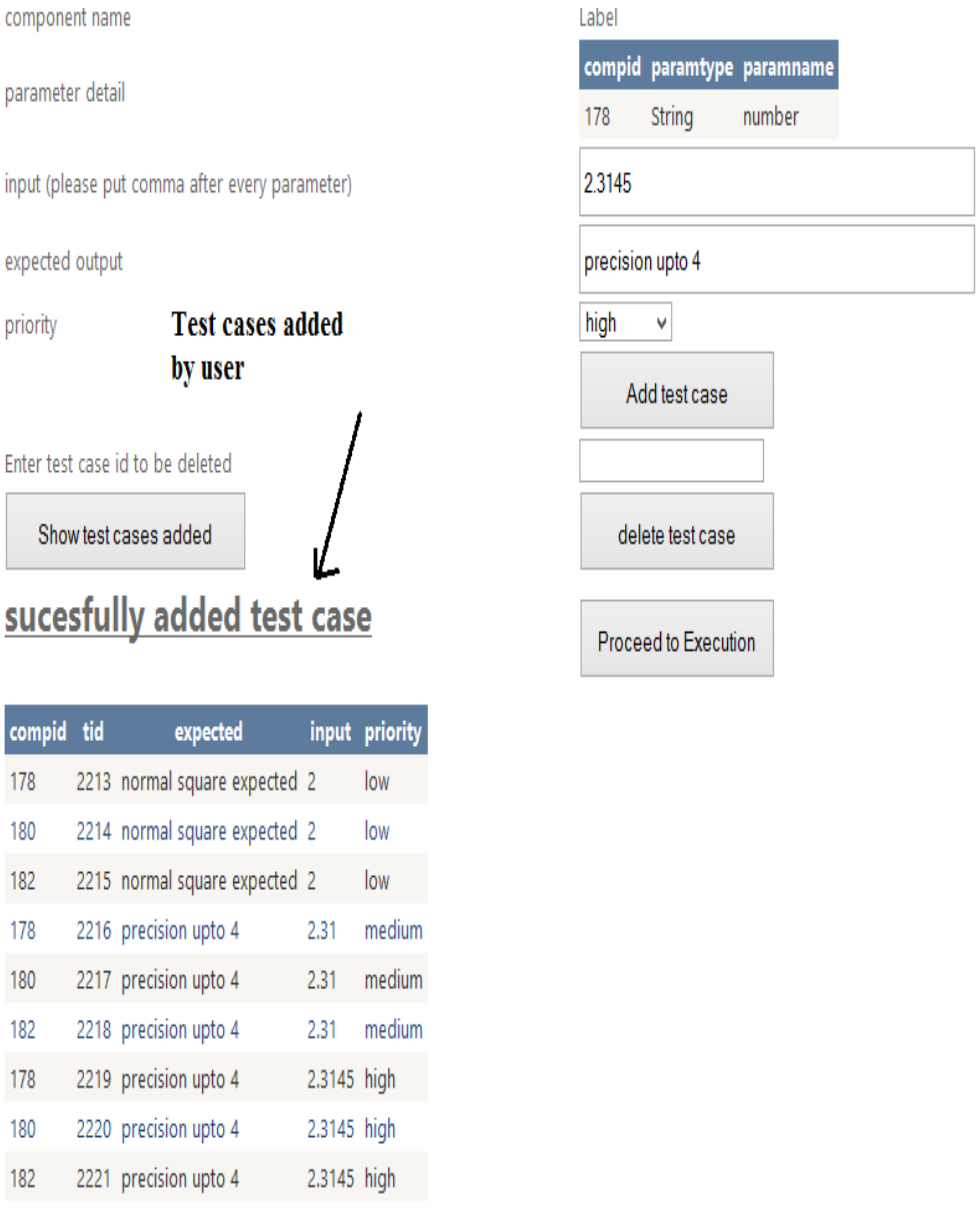


Fig. 4.8 Adding Test Cases

4.6.5 Execution phase

All the components that are added to the pool of testing are executed to each test case that is added by the user. The system usually follows the following step in this phase of execution.

Step 1: Select component from buffer and execute each test case on the component. Proceed to step 2.

Step 2: If the pool of test cases is empty, remove the component that has completed the execution and add the next component to the buffer and proceed to step 1. If all the components have completed the testing phase, proceed to step 3.

Step 3: Add the result to the database of each test case that is associated with each component.

The execution feature of system is shown in fig. 4.9.

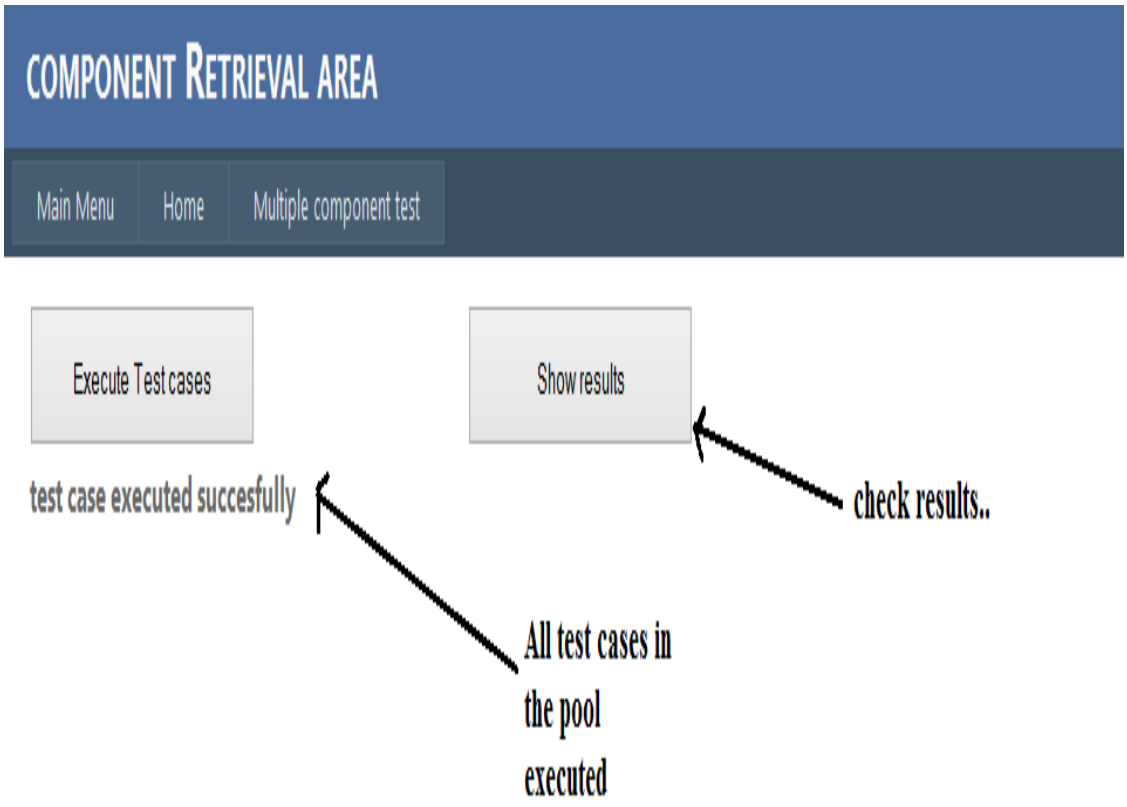
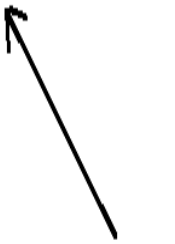


Fig.4.9 Execution of COTS Components

4.6.6 Execution results of COTS testing

After the execution is complete, the system stores the result in database. All the outputs obtained by executing the component from provided test cases are used to create further reports. The results obtained are shown in fig.4.10.

compid	tid	input	output	expected	priority
177	2222	2	4	normal square expected	low
178	2223	2	4	normal square expected	low
179	2224	2	4	normal square expected	low
180	2225	2	4	normal square expected	low
182	2226	2	4	normal square expected	low
177	2227	2.31	Input string was not in a correct format.	normal square expected	medium
178	2228	2.31	5.3361	normal square expected	medium
179	2229	2.31	Input string was not in a correct format.	normal square expected	medium
180	2230	2.31	5	normal square expected	medium
182	2231	2.31	5.33609973564148	normal square expected	medium
177	2232	2.3145	Input string was not in a correct format.	precision upto 8	high
178	2233	2.3145	5.35691025	precision upto 8	high
179	2234	2.3145	Input string was not in a correct format.	precision upto 8	high
180	2235	2.3145	5	precision upto 8	high
182	2236	2.3145	5.35691068262673	precision upto 8	high

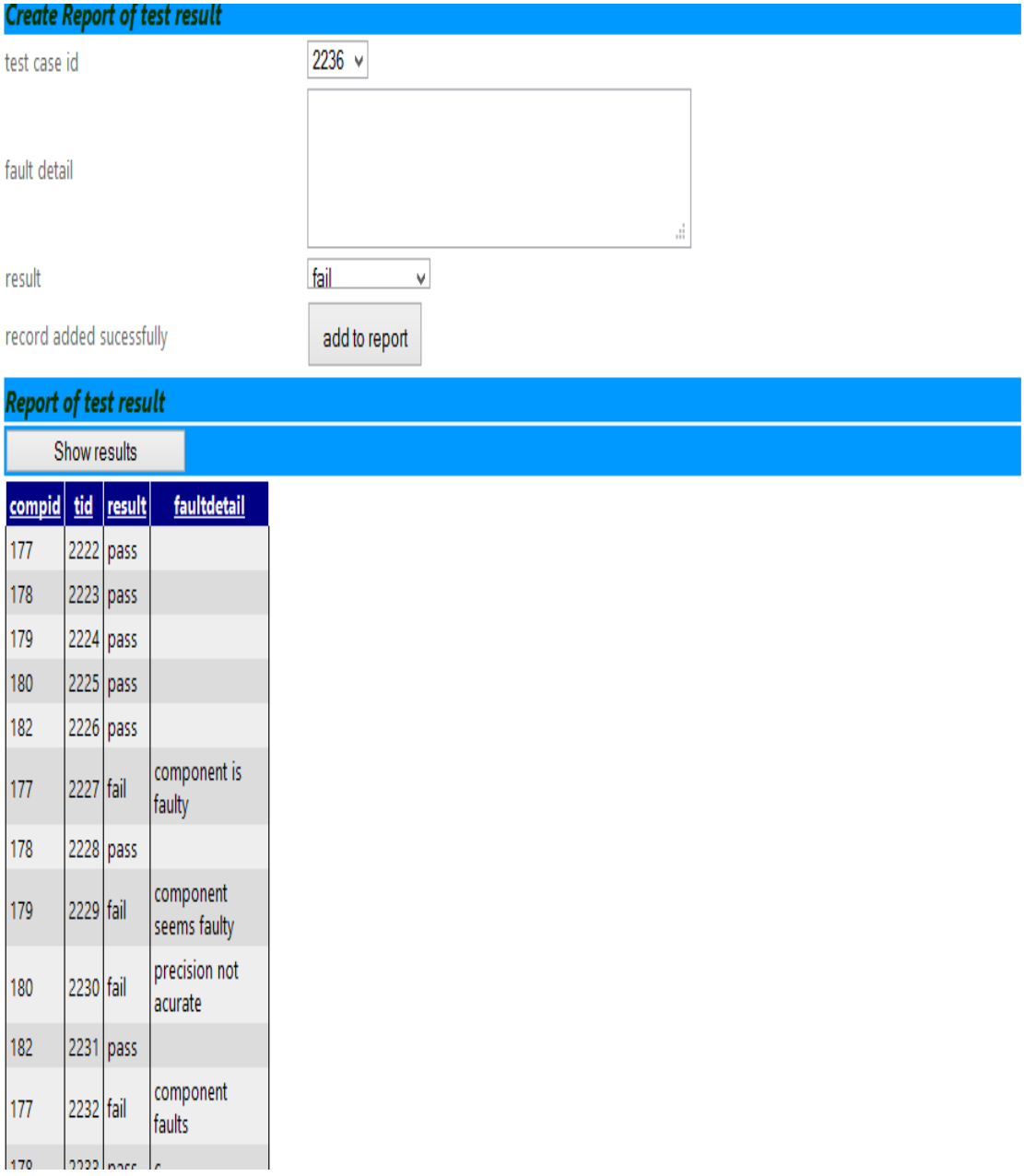


Execution result of testing

Fig.4.10 Results Obtained from Execution Phase

4.6.7 Report creation

Now the user can create the report by analysis of the results that are obtained from the system. The user has to select test case id (tid) from the dropdown list and select pass or fail from second dropdown list. Fault detail can also be provided by the user in the provided textbox. The report creation is shown in fig. 4.11.



compid	tid	result	faultdetail
177	2222	pass	
178	2223	pass	
179	2224	pass	
180	2225	pass	
182	2226	pass	
177	2227	fail	component is faulty
178	2228	pass	
179	2229	fail	component seems faulty
180	2230	fail	precision not acurate
182	2231	pass	
177	2232	fail	component faults

Fig. 4.11 Report Creation

4.6.8. Data analysis of COTS testing

The system record the time interval taken by each test case executed on each component. The total time taken by the whole process is also shown. The report that is created can now further be used for COTS selection. The data analysis is shown in fig. 4.12.

tid	Time_Taken_in_Milliseconds
2222	73
2223	109
2224	131
2225	148
2226	166
2227	239
2228	261
2229	300
2230	310
2231	322
2232	359
2233	380
2234	435
2235	453
2236	472

compid	tid	result	faultdetail	priority
177	2222	pass		low
178	2223	pass		low
179	2224	pass		low
180	2225	pass		low
182	2226	pass		low
177	2227	fail	component is faulty	medium
178	2228	pass		medium
179	2229	fail	component seems faulty	medium
180	2230	fail	precision not acurate	medium
182	2231	pass		medium
177	2232	fail	component faults	high
178	2233	pass	c	high
179	2234	fail		high
180	2235	fail		high
182	2236	fail		high

Fig. 4.12 Data analysis obtained from interface testing of COTS Components.

Chapter 5

Experimental Results

5.1 Test results

In our test process, we selected five components for testing and applied a series of test cases on them. As the test cases were run on components, the time intervals were also calculated for each test cases associated with each component. The following results were obtained shown in table 5.1.

Table 5.1 Execution results

Compid	Tid	Result	Fault detail	Priority
177	2222	Pass	Nil	Low
178	2223	Pass	Nil	Low
179	2224	Pass	Nil	Low
180	2225	Pass	Nil	Low
182	2226	Pass	Nil	Low
177	2227	Fail	Component error	Medium
178	2228	Pass	Nil	Medium
179	2229	Fail	Component error	Medium
180	2230	Fail	Precision not accurate	Medium
182	2231	Pass	Nil	Medium
177	2232	Fail	Component error	High
178	2233	Pass	Nil	High
179	2234	Fail	Component error	High
180	2235	Fail	Precision not accurate	High
182	2236	Fail	Precision not accurate	High

On the basis of above data we found that the component with ID 178 has passed maximum number of test cases, so it was the best candidate among the others selected components. The graph shown in fig. 5.1 shows the results

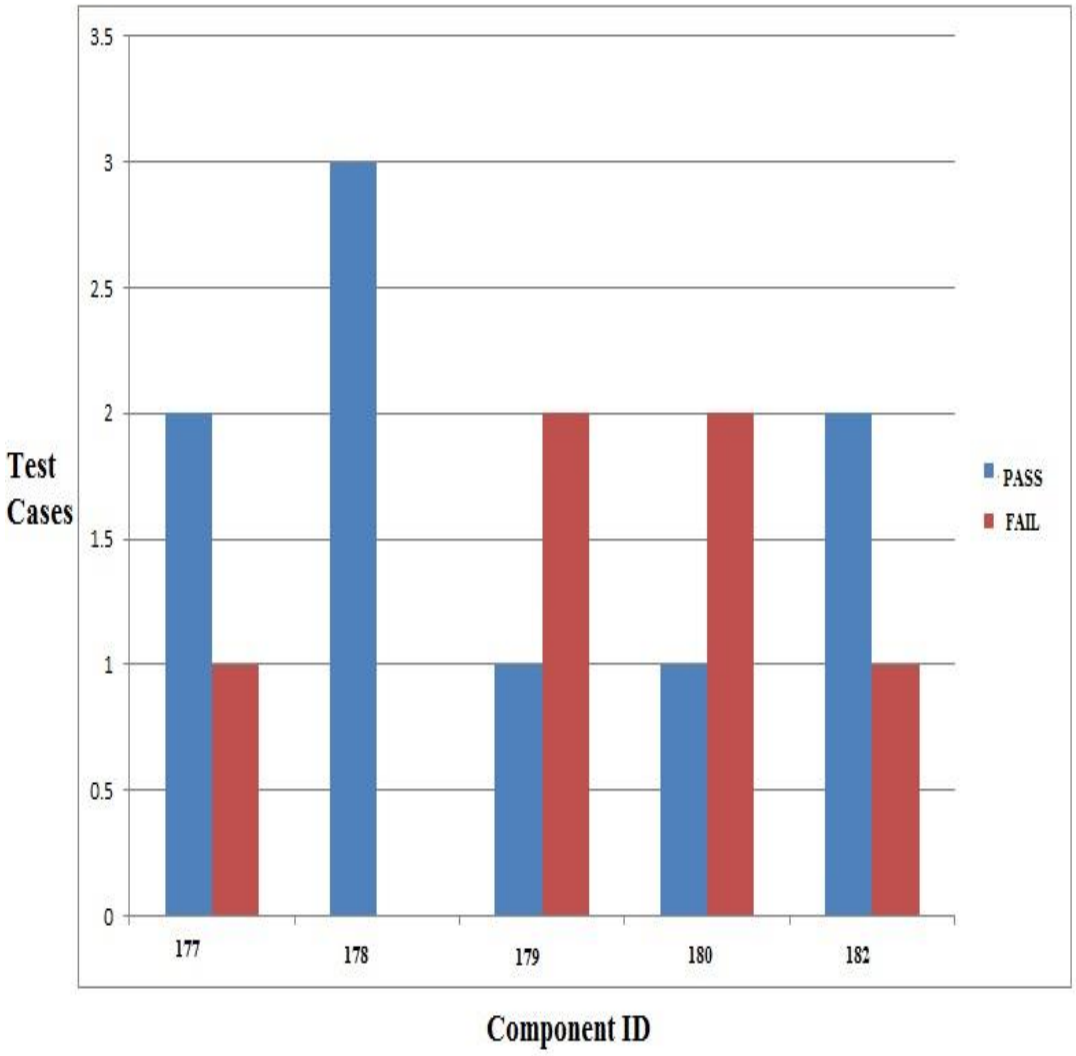


Fig. 5.1 Results of selected components

The test cases were prioritized as high, medium and low. So according to the priorities the scores were calculated for each component. Low priority is assigned with multiplier 1, medium with multiplier 2 and high with multiplier 3.

Table 5.2 Scores of each component

Compid	Test cases having low priority	Test cases having Medium priority	Test cases with High priority	Score
177	1	1	0	3
178	1	1	1	6
179	1	0	0	1
180	1	0	0	1
182	1	1	0	3

So from the test result it is found that component with ID 178 got the highest score and is the best candidate among other components. The fig. 5.2 shows the score of each component.

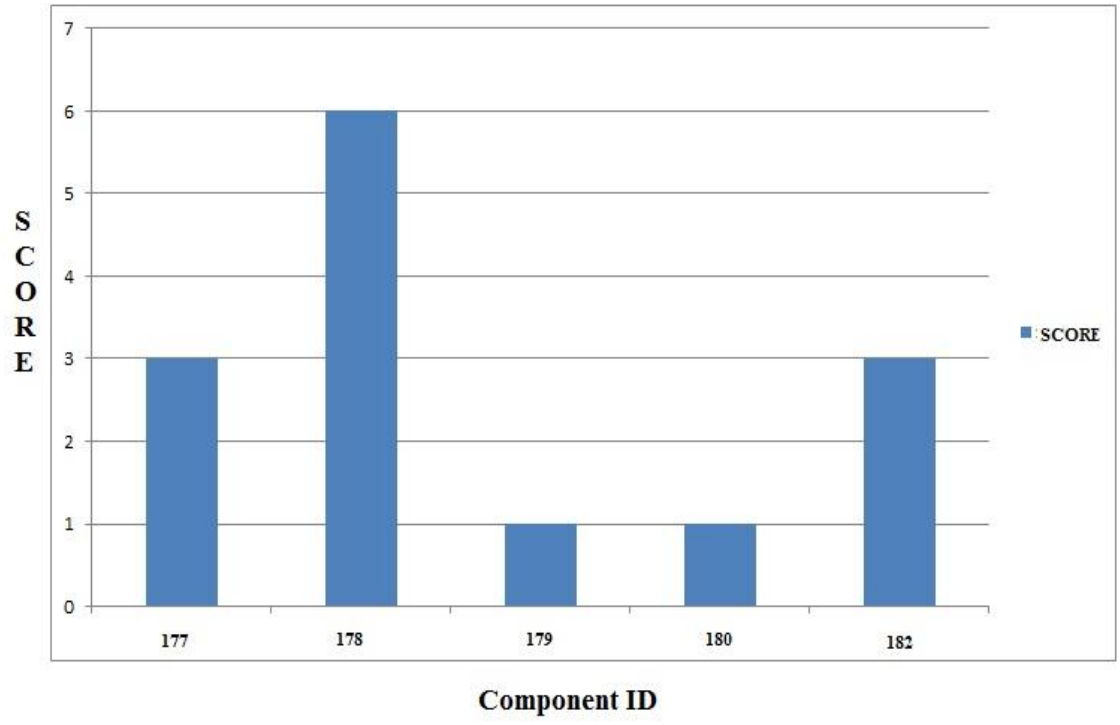


Fig. 5.2 Score of Each Component

In our test process the system recorded each interval between executions of test cases on the selected components. So from the experimental procedure, the following data was recorded.

Table 5.3 Time intervals for executed test cases

Test case ID	**Time (in milliseconds)**
2222	73
2223	109
2224	131
2225	148
2226	166
2227	239
2228	261
2229	300
2230	310
2231	322
2232	359
2233	380
2234	435
2235	453
2236	472
Total time taken	4158(ms)

The total time taken by all the test cases is recorded by system in mili seconds. Total time taken by all the test cases is 4158 (ms). The time taken by each test case is shown in fig.5.3.

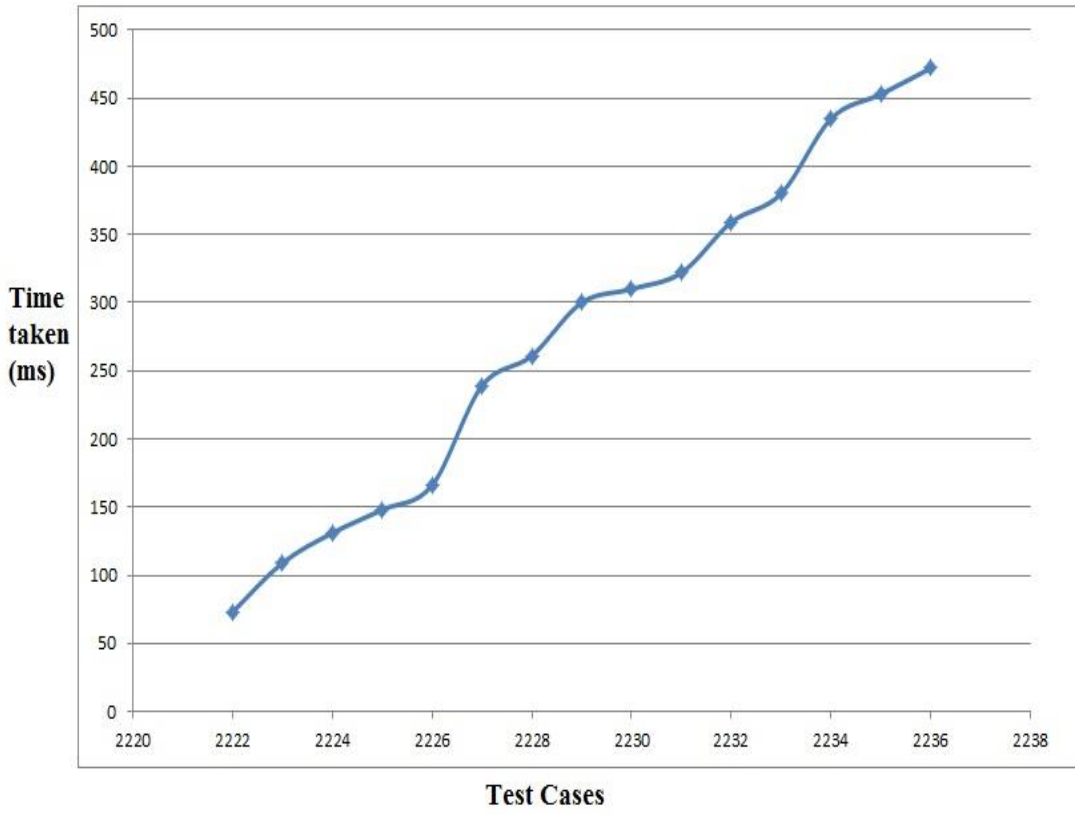


Fig. 5.3 Time taken for execution of each test case

In the test process five components were selected for testing and a series of test cases were applied on them. As the test cases were run on components, the time intervals were also calculated for each test cases associated with each component. It was found that the component with ID 178 has passed maximum number of test cases, so it was the best candidate among the others selected components.

Chapter 6

Conclusion and future scope

6.1 Conclusion

The interface testing technique for selection of better COTS is capable of easing the selection of COTS when there are multiple candidates. The user can select multiple components that fulfill the same requirement and can execute multiple test cases automatically. According the best score, best candidates among these candidate components, the selection for best candidate can be made. Currently only components that support COM were considered. Only Dynamic link library (DLL's) was considered for testing. Support for more component types like EXE can be added later. By using the public interfaces of COTS components, source code was not required for testing.

6.2 Future Scope

Currently this technique is capable of testing single components interaction only. This technique can be expanded more to support multiple component interaction, which can further be used for integration testing. Currently our approach support COM based components only, but it can also be used for testing variety of components that uses other standards like CORBA, JAVABEANS. Number of testing techniques like fault injections can be applied to public interfaces to understand the behavior of COTS components. Monitors can also be used to record information flow into the COTS components from the system and from COTS components to the software system.

References

1. D. McKinney, "Impact of Commercial Off-The-Shelf (COTS) Software on the Interface between Systems and Software Engineering" , In Proceedings of the 1999 International Conference on Software Engineering, 22 May 1999: pp. 627-628.
2. M. R. Vigder, J. C. Dean, "Building maintainable COTS based systems", In Proceedings of International Conference on Software Maintenance, Nov 1998: pp.132-138.
3. V. G. Sami Beydeda, "Testing Commercial-off-the-Shelf Components and Systems", Springer, (Jun-2005).
4. R. S. Pressman, "Software Engineering - A Practitionar's Approach", 5th Edition, McGraw-Hill International Edition, (2001).
5. V.R. Basili and B. Boehm, "COTS-based systems top 10 list", Computer, Vol.34, No.5, (2001): pp. 91-95.
6. D. J. Carney and A. O. Patricia, "The commandments of COTS: Still in search of the promised land", Crosstalk Vol., 10, No. 5, (1997): pp. 25-30.
7. M. Morisio et al. "COTS-based software development: Processes and open issues", Journal of Systems and Software, Vol. 61, No. 3, (2002): pp. 189-199.
8. Offutt, A. Jefferson, and Roland H. Untch. "Mutation 2000: Uniting the orthogonal", Mutation testing for the new century. Springer US, (2001): pp. 34-44.
9. M. E. Delamaro, J. C. Maidonado and A. P. Mathur, "Interface mutation: An approach for integration testing", IEEE Transactions on Software Engineering, Vol. 27, No. 3, (2001): pp. 228-247.
10. L. Mariani, S. Papagiannakis, and M. Pezze, "Compatibility and Regression Testing of COTS- Component-Based Software", In Proceedings of the 29th international conference on Software Engineering (ICSE 2007), IEEE Computer Society, Washington, DC, USA, pp. 85-95.

11. Y. Jiang, S. Hou, J. Shan, L. Zhang, B. Xie, “Contract-Based Mutation for Testing Components”, International Conference on Software Maintenance (ICSM), (2005): pp. 483-492.
12. Y. Wang, G. King, & H. Wickburg, “A method for built-in tests in component-based software maintenance”, In Proceedings of the IEEE Third European Conference on Software Maintenance and Reengineering, (1999): (pp. 186-189).
13. H. Gross and M. Nikolas, "Built-in contract testing in component integration testing", Electronic Notes in Theoretical Computer Science, Vol. 82, No. 6 (2003): pp.22-32.
14. Z. Jin and A.J. Offut, “Coupling Based Criteria for Integration Testing”, Software Testing Verification and Reliability, Vol. 8, (1998): pp. 133-154.
15. J.M. Haddox, G.M. Kapfhammer, and C. C.Michael, "An approach for understanding and testing third party Software components", In Proceedings of the Annual Reliability and Maintainability Symposium, Seattle, WA, USA, (2002): pp. 293-299.
16. S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information", IEEE Transactions on Software Engineering, Vol. 11, No. 4, Apr. 1985, pp. 367-375.
17. R. A. DeMillo, R. J. Lipton, & F. G. Sayward, “Hints on test data selection: Help for the practicing programmer”, Computer, Vol. 11, No. 4, (1978): pp. 34-41.
18. R. G. Hamlet, "Testing programs with the aid of a compiler", IEEE Transactions on Software Engineering, Vol. 4, (1977): pp. 279-290.
19. G. J. Myers, S. Corey and T. Badgett, “The art of software testing”, Wiley, 2011.
20. S. A. Vilkomir and P. B. Jonathan, "Formalization of control-flow criteria of software testing", In Proceedings of *COMPSAC*, 2001.
21. A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf, “An Experimental Determination of Sufficient Mutant Operators”, ACM Transactions on Software Engineering Methodology, Vol. 5, No. 2, (1996): pp. 99-118.
22. A.J. Offutt, G. Rothermel, and C. Zapf, “An Experimental Evaluation of Selective Mutation”, In Proceeding of 15th International Conference Software Engineering, (May 1993): pp. 100-107.

23. A. Haley and S. Zweben, “Development and Application of a White Box Approach to Integration Testing”, Journal of Systems and Software, Vol. 4, (1984): pp. 309-315.
24. U. Linnenkugel and M. MuÈllerburg, “Test Data Selection Criteria for (Software) Integration Testing”, In Proceeding of First International Conference Systems Integration, (Apr. 1990): pp. 709-717.
25. L. Mariani, P. Mariani, "Behavior capture and test: Automated analysis of component integration." In Engineering of Complex Computer Systems, ICECCS 2005. In Proceedings of 10th IEEE International Conference on, (2005): pp. 292-301.
26. L. Mariani, M. Pezz`e, and D. Willmor. “Generation of Selftest Components”, In proceedings of the International Workshop on Integration of Testing Methodologies, volume 3236 of LNCS, Springer, (2004): pp. 337–350.
27. J.E. Gaffney, and T.A. Durek, "Software reuse-key to enhanced productivity: some quantitative models", Information and Software Technology, Vol. 31, No. 5, (1989): pp. 258-267.
28. Microsoft Corporation and Digital Equipment Corporation. (October 24, 1995),” The Component Object Model Specification (Draft Version 0.9)”, Microsoft Corporation.
29. J. Rumbaugh, I. Jacobson, and G. Booch (Eds.),”The Unified Modeling Language Reference Manual”, Addison-Wesley Longman Ltd., Essex, UK (1998).
30. A. Orso, M. J. Harrold, D.Rosenblum, “Component Metadata for Software Engineering Tasks”, In Proceeding of The 2nd International Workshop on Engineering Distributed Objects (EDO 2000), LNCS 1999, (2001): pp. 126-140.
31. A. Orso, M. J. Harrold, D. Rosenblum, and et al., “Using Component Meta content to Support the Regression Testing of Component-based Software”, In Proceeding of ICSM’01, IEEE Press, (2001): pp. 716-725.
32. Y. Wang, G. King, H. Wickburg, “A Method for Built-in Tests in Component-based Software Maintenance”, In Proceedings of CSMR’99, IEEE Press, (1999): pp. 186-189.

33. M. Chengying, "Built-in Regression Testing for Component-based Software Systems," 31st Annual International Computer Software and Applications Conference, (July 2007): pp.723-728.
34. I. Sommerville, "Software Engineering", 4th edition, Addison-Wesley, Wokingham, U.K. (1992).
35. B. Beizer, "Black-box testing: techniques for functional testing of software and systems", John Wiley & Sons, Inc., (1995).
36. Constantine, L. L. and Yourdon, E, "Structured Design", Prentice-Hall, Englewood Cliffs, NJ, U.S.A. (1979).
37. J.C. Knight, R. W. Lubinsky, J. McHugh, and K.J. Sullivan, "Architectural approaches to information survivability," Technical Report" CS-97-27, University of Virginia, (Sept. 1997).
38. C. L. Braun, "A lifecycle process for the effective reuse of commercial off-the-shelf (COTS) software", *In Proceedings of the 1999 ACM symposium on Software reusability*, (1999): pp. 29-36
39. M. Vigder, J. Dean, "An architectural approach to building systems from COTS software components", Technical Report 40221, National Research Council, (1997).
40. J. Voas, G. McGraw, "Software Fault Injection: Inoculating Programs Against Error", John Wiley and Sons (1998).

LIST OF PUBLICATIONS

Published

1. Jagdeep Singh, Shivani Goel, "COTS and OSS: A Comparative View", International Journal of Computer Science & Applications (TIJCSA), Vol.1, No.12, 2013: pp. 55-59.

2. Jagdeep Singh, Shivani Goel, “Testing techniques for COTS”, International Journal of Advanced Research in Computer Science and Software Engineering *(*IJARCSSE*)*”, Vol.3, No.4, 2013: pp. 829-833.