International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:12, 2008

COTT – A Testability Framework for Object-Oriented Software Testing

A. Goel, S.C. Gupta and S.K.Wasan

***Abstract*—**Testable software has two inherent properties – observability and controllability. Observability facilitates observation of internal behavior of software to required degree of detail. Controllability allows creation of difficult-to-achieve states prior to execution of various tests. In this paper, we describe COTT, a Controllability and Observability Testing Tool, to create testable object-oriented software. COTT provides a framework that helps the user to instrument object-oriented software to build the required controllability and observability. During testing, the tool facilitates creation of difficult-to-achieve states required for testing of difficult-to-test conditions and observation of internal details of execution at unit, integration and system levels. The execution observations are logged in a test log file, which are used for post analysis and to generate test coverage reports.

***Keywords*—**Controllability, Observability, Test Coverage and Testing Tool.

I. Introduction

TESTABLE software is one that can be tested easily, systematically and without following any ad-hoc measures [22][24]. Testable software need to possess two characteristics i.e. observability and controllability. During testing, there is a need to observe internal details of execution to ensure the correctness of processing and to diagnose errors. Provisions have to be made in software so that the internal behavior of software can be observed during testing. *Observable software* makes it feasible for tester to observe the internal behavior of software, to the required degree of detail. Also, during testing, some of the tests are difficult to conduct as state of software required to be created before these tests can be executed are difficult to create using the commands available at user interface. Some provisions have to be made in software so that the difficult-to-create states can be created easily during testing, without using any ad-hoc mechanism. *Controllable software* makes it possible to initialize software to desired states, prior to execution of various tests.

In this paper, we present COTT (Controllability and Observability Testing Tool), which is a set of Java classes that uses our probe-based observability mechanism [1] and controllability mechanism [2], for creation of testable software. Our tool provides an environment for testing, which helps to create testable object-oriented software by providing (1) interactive interface for instrumentation of source-code with probes and control commands, (2) interface to control the inserted probes and control commands, externally, to display internal execution details at unit, integration and system levels, and, to create difficult-to-achieve states required for testing of difficult-to-test conditions, respectively, (3) test coverage report of probes at probe, method, class, inheritance and dynamic binding levels and coverage of control command, and (4) test log file for post analysis.

COTT is composed of two subsystems – instrumentation and testing. The instrumentation subsystem provides an interactive interface for instrumentation and preprocessing, to produce a preprocessed program for the testing subsystem. The testing subsystem displays test output. Additionally, the testing subsystem provides test coverage reports and allows analysis of the generated test log file.

COTT consists of several components –

- an instrumentation program that helps in insertion of probes and control commands,
- a test preprocessor that collects details of instrumented user program,
- a testability interface to make settings for probe and control commands,
- a test display output interface that displays test output in a hierarchical and tabular form,
- test coverage reporter which provides coverage information at different levels based on probes and control commands, and
- test file analyzer that supports post analysis of the generated test log file.

The major contribution of this work is to provide a systematic framework for testability viz. observability and controllability. The observability framework focuses on handling the perennial problem of large log output data. Unlike previous work, the mechanism addresses observing the internal execution details at unit, integration and system levels. During unit testing, input/output of methods and impact of method execution on state of object are observed. The sequence of execution of classes and input/output of class is observed during integration testing. The input/output of integrated units is observed during system testing.

The controllability framework is based on a testing mechanism that focuses on creation of difficult-to-create states required for testing of difficult-to-test conditions. Unlike previous work that requires fault-injection tools, each for

A.Goel is with Department of Computer Science, University of Delhi, Dyal Singh College, New Delhi-110003, India (phone: +91-120-4245356; e-mail: agoel@ dsc.du.ac.in).

S. C Gupta is Senior Technical Director in National Informatics Centre (NIC), New Delhi, India (e-mail: scgupta@nic.in).

S. K. Wasan is with Department of Mathematics, Jamia Millia Islamia, New Delhi, India (e-mail: skwasan@gmail.com).

specific kind of faults, dynamically during runtime, our mechanism is generic and systematic that requires provisions to be made during software development phases.

In this paper, section II discusses testability of object-oriented software. An overview of COTT is given in Section III. Section IV and V describe the details of COTT subsystem. In section VI we present implementation and test results of our tool. Section VII describes related work. Finally, we state the conclusion.

II. TESTABILITY OF OBJECT-ORIENTED SOFTWARE

The unique architecture and features like inheritance and dynamic binding, has resulted in some issues involved in testing of object-oriented software to be different from the testing issues of conventional software [3][9]. In order to handle the testing issues of object-oriented software, conventional software testing tools have to be adapted or new tools need to be developed.

Binder [1][23], Freedman [22] and Pettichord [5] stress the need to design object-oriented software for testability. According to Binder [23], "To test a component you must be able to control its input and observe its output". According to Freedman [22], "observability refers to ease of determining if specified inputs affect the outputs; controllability refers to the ease of producing a specified output from a specified input".

***Observability*:** Our research has developed a *probe-based observability mechanism* [1] that adapts probe for object-oriented software testing. Probe is a method invocation [24] having the syntax

probe(probe_id, probe_message)

where, *probe* is a method name, *probe_id* is a unique structured identifier identifying location of *probe_message*, and, *probe_message* contains state-related attributes or message, relevant at probe's location.

The idea behind our mechanism is quite simple. Our mechanism defines the structure of *probe_id* and *probe_message* of probe as follows -

probe_id - "level_number/probe_number"

probe_message-"variable1:val1..variableN:valn msg:string"

where, *level_number* is L1, L2 and L3 for system level, integration level and unit level testing respectively. The *probe_number* is a unique integer in a class. val1, valn is values of variables and string is a message.

Probes are inserted in source-code as *Log.penter, Log.pexit* and *Log.pmsg* at method entry, exit and anywhere in between, respectively. For example, Log.penter("L1/1", "msg:start main");. The class "Log" defined in our tool based on probe-based observability mechanism interacts with software embedded with probes during testing.

During testing, probes are controlled externally– activate/deactivate, to display internal execution details at unit, integration and system levels.

Controllability: We define *control condition* as the condition that is difficult-to-test, as state of software required for its execution is difficult-to-create from user interface of software. The state that is difficult-to-create from user interface of software is defined as *control state*. Our research has proposed and developed a controllability mechanism [2] that facilitates creating control states, required for testing of control conditions in object-oriented software. We identify and classify control conditions [2] in two categories as-

(1) Special conditions, exception conditions and error conditions that arise from software environment or from within the software. The errors from software environment may be due to resource exhaustion or failure like disk-full, out of memory, disk sector bad, network failure, frame lost etc. The errors from within software may be due to internal resource limit conditions, initial state setting and inter-module failure like table full, timeout, read empty file, initialization, assignment etc., and

(2) Specialized tests that require setting up of test environment or additional code to perform the testing activity, like, full-scale test, stress test, create large input data, initialize software etc.

The controllability mechanism requires identification of control state, creation of control environment for each identified control state, and insertion of control commands in source-code. To create control environment, the action to be taken to create a control state is decided. For example, table full condition may require - filling table with dummy data, resource exhaustion/failure - throw exception, communication software - block frame, create large input - generate data. Next, the decided action is implemented and inserted as a *control method* in user program. This may require assignment, insertion of additional method in an existing class or a new class (a class is defined that acts as a control pipe between receiver and sender, facilitating testing of frame drop, frame out of sequence etc.). In some cases, an exception is thrown; for example, signaling disk full results in execution of code handling disk full condition.

Our mechanism defines control command as a method invocation having a structured *control_id*, as - *C.c(control_id)*. The *control_id* is *"control_number/ control_name"* where, *control_number* is a unique integer and *control_name* is a string representing the condition to be tested. 'C' and 'c' are class name and method name respectively. Control command invokes the control method. The control command is inserted in two formats-

Format 1. if (control command)
{Control Method}

Format 2. if (Control State || control command)
{Control Method}

The first format is used when an additional if-statement is required to be inserted. The second format is used when the control state to be achieved is already being detected by an if-condition in user program. For example,

Format 1. if (C.c("1/Initialize")) sr.initialize()

Format 2. if (!(siteCommit_T=1)||(C.c("1/abortCase"))) {};

The class 'C' defined in our tool interacts with user program embedded with control command during testing. During testing, control commands are externally activate/ deactivate and control breakpoint set. An active control command activates control environment, resulting in creation of Control State.

III. COTT ARCHITECTURE

COTT provides an environment for testing based on probe-based observability mechanism and controllability mechanism. COTT helps tester in instrumentation of user program with probes and control commands. The tool provides tester with a testability interface, for accessing and controlling control commands and probes externally during testing, for the desired controllability and observability, respectively. Additionally, the tool provides test coverage of probes and control commands for a single, set and all test cases combined. COTT also provides information about probes that have not got executed, so that test cases can be designed accordingly. The tool also stores internal execution details in *test log file* which can be viewed offline using testability interface.

The tool architecture is composed of two main subcomponents, namely, instrumentation subsystem and testing subsystem, as shown in Fig. 1. The purpose of instrumentation subsystem is to insert probes and control commands, and to create control environment in user program. The subsystem also preprocesses user program to store information about inserted probes and control commands, required by testing subsystem.

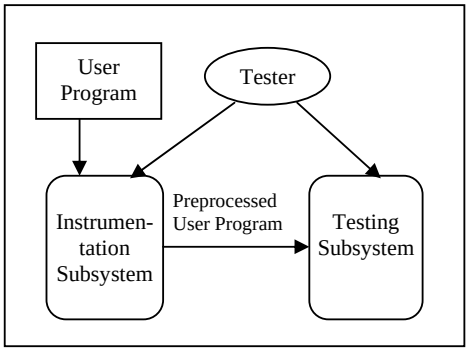

Fig. 1 COTT overall architecture

The re-compiled preprocessed user program is input to testing system. The testing subcomponent offers an environment to execute, monitor and analyze the test. It supports four features - (1) testability interface that allows settings to be made for probes and control commands, (2) output display in a tabular form, (3) visualization of test coverage reports, and (4) analysis of test log file. The primary purpose of settings made via testability interface is to facilitate display of internal execution details at unit, integration and system level, and creation of difficult-to-create states.

IV. INSTRUMENTATION SUBSYSTEM

Fig. 2 shows subcomponents of instrumentation subsystem. A Java user program is input to instrumentation program. The instrumentation program inserts probes and control commands in source code of user program. The instrumented user program produced as a result of instrumentation is preprocessed using *test preprocessor*, to store information about probes, control commands, inheritance hierarchy and dynamic binding relationship. The final output of instrumentation subsystem is the preprocessed user program and files storing information gathered during preprocessing.

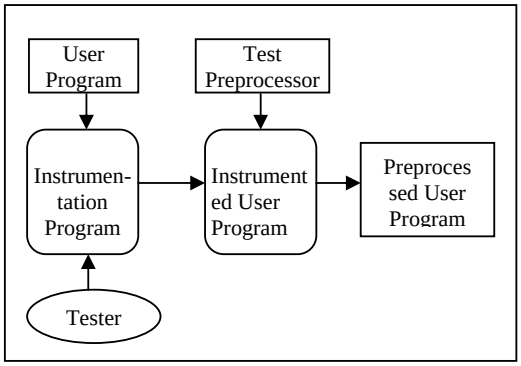

Fig. 2 Instrumentation Subsystem

Instrumentation Program provides an interactive interface to tester, for instrumentation. The insertion of probes by instrumentation program combines the information about integrated units, which tester provides from this interface, with the information of classes in user program, for creating an instrumented code.

For identifying integrated units in user program, the instrumentation program allows tester to interactively select classes that represent the integrated units. Once identified, probes are inserted by instrumentation program at beginning and end of each method. The *level_number* of probe is L1 in public methods of classes that interact with other integrated units, *level_number* is L2 in public methods of rest of classes and *level_number* is L3 in private and protected methods of class. For inheritance coverage, instrumentation program inserts '*getClass()*' in *probe_message* of probes of inherited methods, to find class invoking the inherited method. Additionally, instrumentation program also provides an interface to selectively insert probes having *level_number* L3, anywhere in between beginning and end of a method.

To insert control commands in user program, instrumentation program provides interface to tester (1) to write control command and to identify the location where control command is to be inserted, and, (2) to write control method and to identify the location where control method is to be inserted. The instrumentation program uses this information to insert control commands and control method in user program.

The user program embedded with probes and control program is the instrumented user program, ready for preprocessing by Test Preprocessor. Table I shows probes at statement (3) and (4), control method *largeInput()* at statement (2), and control command at statement (1) which invokes control method *largeInput()* to create large input for B+ tree software.

Test Preprocessor takes compiled instrumented user program as input and collects information about classes, methods, probes, inheritance hierarchy and dynamic binding relationship, and, control commands and stores them in a file.

TABLE I
PROBE, CONTROL COMMAND AND CONTROL METHOD IN B+ TREE SOFTWARE TO CREATE LARGE INPUT

```
class UserInterface {
 public static void main(String args[]) throws IOException
{    :
    if (C.c("1/LargeInput")) largeInput();          ----(1)
                    : }
 public static void largeInput() throws IOException { -(2)
  int noOfKeys = 0;
  int key;
  Log.penter("L2/9", "enter large input")          -----(3)
  System.out.println("Enter number of keys(int)");
  noOfKeys = acceptKey();
  for (int i =0; i < noOfKeys; i++) {
     key=(int)java.lang.Math.round
                    (java.lang.Math.random()*100);
     insert(key);}
   Log.pexit("L2/11", "exit largeinput"); }    ----(4)
```

V. TESTING SUBSYSTEM

The testing subsystem works during execution of preprocessed user program for testing, as shown in Fig. 3. The following subsections describe the components of testing subsystem, namely, Testability Interface, Test Display Output, Test Coverage Reporter, and Log File Analyzer.

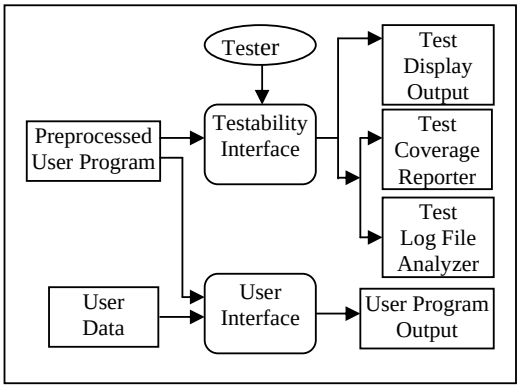

Fig. 3 Testing Subsystem

A. Testability Interface

Probes and control commands that have been inserted in user program are externally controlled using probe setting and control setting made via testability interface, during testing. The testability interface displays current probe setting i.e. probe activation/deactivation and probe breakpoint setting, and, current control setting, i.e., control activation and control breakpoint setting.

To activate and deactivate probes, tester uses the following commands:

A: class_name/method_name/level_number/probe_number
D:class_name/method_name/level_number/probe_number

respectively. Output of only active probes is displayed on screen and stored in test log file. Probes can also be referred to in a generic style using a "*". For unit testing, probes at *level_number* L3 are activated. It results in activation of probes at *level_number* L1, L2 and L3. Probes at *level_number* L2 are activated during integration testing. It results in activation of probes at *level_number* L1 and L2. During system testing, probes at *level_number* L1 are activated. For example, "A:Node/*/L2/*" activates all probes at level number L1 and L2 in all methods of class Node.

The tester makes probe breakpoint setting as follows:

class_name/method_name/level_number/probe_number
A String

Probe breakpoint allows breakpoints to be set on selected probes. On occurrence of break, execution of user program pauses. The tester can observe already displayed probes and change probe settings to observe rest of execution details.

For control setting, the control commands in user program are displayed to tester. Tester selects a single or a group of control commands to be activated. The tester also selects control command to set control breakpoint.

B. Test Display Output

Test Display Output consists of two parts – Check Setting and Test Output. Test settings made from Testability Interface, and probe and control command are input to Test Display Output. Fig. 4 shows interaction between Testability Interface and Test Display Output.

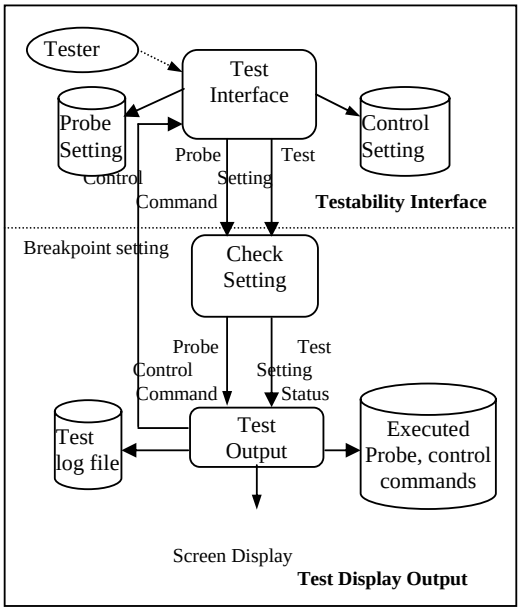

Fig. 4 Interaction between Testability Interface and Test Display Output

Check Setting checks activation and breakpoint status of probe and control commands. For this, it performs two steps - (1) it gets all probes and control commands from Test Preprocessor and stores them in a buffer. It gets test settings and marks probes and control commands in buffer as active/deactive. (2) On receiving a probe or control command that is being executed, it checks status of the received command with marked settings in buffer and returns status of activation and breakpoint setting (0-false, 1-true).

Test Output checks status of breakpoint setting for the received command. If the received command is a probe, and breakpoint is true, it allows tester to make new probe setting

from Testability Interface. If false, it (1) stores all executed and active probes and control commands in Test log file, (2) displays probes on screen, and (3) stores all executed probes and control commands (active and deactive).

Execution details displayed during testing consist of -

- Class being executed
- Method of the class being executed
- Value of parameters or messages at method entry/exit or within method
- Hierarchy of execution of classes
- Probe identification to locate probe displaying parameter/message, in the code

These execution details are displayed at *unit, integration and system levels*. The internal execution detail of deleting a node from link list, at unit level and integration level is shown in Table II and Table III respectively. At unit level, code-level behavior of user program is displayed. At integration level, sequence of execution of classes and their input/output is displayed. It does not display internal execution details of class as shown during unit testing.

TABLE II
INTERNAL EXECUTION DETAILS AT UNIT LEVEL (DELETE NODE FROM LINKLIST) (FROM [1]) (->ENTER, <-EXIT, --BETWEEN)

Class Name	Method Name	LevelNo./Probe No.
|→List	deleteNode(Node)	L2/9->delete node =Node@f133f325
|→Node	equals(Object)	L2/13-> compare =Node@2debf324
|--Node	equals(Object)	L3/16 msg: data equal
|←Node		L2/18<-comparison =true
|→Node |←Node	getNext()	L3/3->msg: get next node L3/4-> next =Node@e96ff324
|--List	deleteNode(Node)	L3/11 msg: node deleted at head
|←List		L2/13<-msg: end delete node

TABLE III
INTERNAL EXECUTION DETAILS AT INTEGRATION LEVEL (DELETE NODE FROM LINKLIST) (FROM [1])

Class Name	Method Name	LevelNo./Probe No.
|→List	deleteNode(Node)	L2/9->delete node=Node@f133f325
|→Node |←Node	equals(Object)	L2/13->compare =Node@2debf324 L2/18<-comparison =true
|←List	deleteNode(Node)	L2/13<-msg: end delete node

If the received command is a control command, and is true, the if-condition enclosing the control command becomes true. A true if-condition, results in execution of the control method defined in if-condition, facilitating creation of a control state, required for the testing of a control condition. For example, if activation status of control command, *(C.c("1/LargeInput")) largeInput()*, is returned true, it results in execution of method *largeInput()*, that creates large input for B+ tree software, as shown in Table I.

C. Test Coverage Reporter

The test coverage reporter provides test coverage of probes at probe, method, class, inheritance and dynamic binding levels, and, coverage of control commands. Coverage reports are provided as percentage coverage, or, list of covered or uncovered probes.

During testing of inheritance hierarchy, there is a need to retest the inherited methods of superclass because inheritance provides new context for inherited methods. The inherited methods, thus, must be executed from class in which they are declared and from class inheriting it. COTT displays inheritance coverage as *(class_name/method_name/class invoking inherited method)*, where *class_name* is class in which inherited method is declared and *class invoking inherited method* is the inheriting class. As shown in Table IV, methods getNext(), setNext() in class Node are invoked from class Node, at statement (1), (3) and from inheriting class IntNode, statement at (2), (4).

In a dynamic binding relationship, there is a need to test all possible methods that can get bound at runtime for a single method invocation. COTT displays coverage at dynamic level as *(class_name/method_name/ method_name from where invoked)*, where *method_name* is dynamically bound method and *method_name from where invoked* is method enclosing call to dynamically bound method. As shown in Table V, printData() defined in Node and IntNode, is invoked from method printList().

TABLE IV
COVERAGE AT INHERITANCE LEVEL (OF INHERITED METHODS) (FROM [1])

3/Node/getNext()	(1)
3/Node/getNext()/IntNode	(2)
5/Node/setNext()	(3)
5/Node/setNext()/IntNode	(4)

TABLE V
COVERAGE AT DYNAMIC BINDING LEVEL (FROM [1])

11/Node/printData()/printList()
8/IntNode/printData()/printList()

D. Log File Analyzer

Log file analyzer accepts Test log file generated during testing as an input and allows the tester to analyze this file, offline. The tool allows the tester to use testability interface to make probe activation and probe breakpoint settings to view Test log file.

COTT Interfaces: The interfaces of COTT are: (1) testability interface that allows tester to set probe activation setting and probe breakpoint setting, (2) probe output display, (3) testability interface which allows tester to set control activation and control breakpoint settings, and (4) test coverage report.

VI. IMPLEMENTATION AND TEST RESULTS

COTT is implemented in Java 1.2. COTT consists of 33 Java classes. All user interfaces have been created using Java Swing. The code size of tool is 5k lines which takes 130KB of disk space.

COTT has been applied during the testing of some large and complex object-oriented software systems - (1) UIServer – It translates UIML document to WML/CHTML document. UIServer is developed using Java and XML. (2) SMS Java is developed in Java, for sending SMS from web or any application using any SMS server. (3) Netram is a software product, to be used as a service. The users of service can view/watch any local site (physical location) remotely as snapshots of images. (4) CIC-Drishya is client-server software and has been installed at NIC, Delhi, to monitor the 500 Community Information Centers (CIC) connected via satellite. The CIC are located in northeastern part of India. The operators at different CIC's send their attendance, image and a message, if necessary, everyday.

Our concern in developing this tool is the overhead in terms of instrumentation program size and execution time, which we found is within reasonable limits.

TABLE VI
CODE SIZE OF INSTRUMENTED USER PROGRAM

User Program	Un-instrumented (# lines)	Instrumented (# lines)	% increase over un-instrumented
UIServer	8994	9699	8%
SMS Java	4000	4256	7%
Netram	6494	7258	12%
CIC-Drishya	6007	6727	12%

Instrumentation using COTT resulted in increase in the size of code of user program, where the increase ranged from 7% to 12% as shown in Table VI. This increase is due to instrumentation of source code of user program with probes and control commands. The number of lines of instrumentation code varies with number of methods in user program. The instrumented code increases with the increase in number of methods in user program. UIServer and SMS Java user programs had less number of long methods. However, Netram and CIC-Drishya had more number of small methods that resulted in more overhead.

TABLE VII
EXECUTION TIME OF INSTRUMENTED USER PROGRAM

User Program	Execution time of un-instrumented programs	Execution time of instrumented programs	% increase in execution time over un-instrumented
UIServer	5.60s	5.85s	4%
SMS Java	3.07s	3.24s	6%
Netram	2.84s	3.15s	11%
CIC-Drishya	2.65s	3.10s	17%

Table VII shows execution time of instrumented user programs, using our tool. The increase in execution time ranges from 4% to 17%. This execution time does not include the time taken to make settings for probe and control commands. We found that the increase in execution time of user programs is proportional to the increase in size of instrumented user program. Fig. 6 represents % increase in execution time and size of user programs, graphically, for the four user programs.

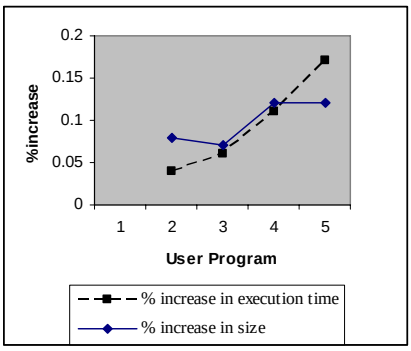

Fig. 5 Percentage increase in execution time and size of user program

Table VIII shows test coverage report based on information collected from probes inserted in user program. The coverage at probe, method and class level is displayed as percentage coverage. Test coverage report also generates the list of uncovered probes at these levels, which helps to locate untested code.

TABLE VIII
TEST COVERAGE REPORT

User Program	Coverage at probe level	Coverage at method level	Coverage at class level
UIServer	84%	88%	100%
SMS Java	69%	72%	100%
Netram	82%	55%	90%
CIC-Drishya	85%	75%	100%

We cite some specific instances of control conditions that required used of controllability mechanism – (1) In Netram software, a circular queue was required to be maintained to store the snapshots. Our mechanism was used to test the circular queue when it is full, (2) In CIC_Drishya, for testing, the mechanism was used to create 500 CIC centre that could send their messages, (3) in SMS-Java, Netram and CIC-Drishya software, to test for frame lost, frame out of sequence, timeout, duplicate frame, and (4) in SMS-Java, Netram and CIC-Drishya software to send large number of frames in a specific time.

Due to interactive nature of our tool, it is best fitted to be used for testing for errors, understanding the execution of software, and for generating test coverage report at inheritance and dynamic binding level. The testability interface allows user to "play" with internal execution details.

VII. RELATED WORK

This section reviews the work done in observability, test coverage and controllability in object-oriented software.

Observability: Traditionally, print statements and debuggers are used, to get access to internal information. However, print statements require frequent commenting and uncommenting each time changes are made. Debugging tools like gdb, give access to all information at a point in execution, but what is important to observe, becomes harder to decide as size of software increases.

There are several tools that use probes to instrument software for testing. Probes are used in tools to trace execution details or observe values of specific variables during testing. Tools like Compaq's JTREK [8], JOIE [10] and BIT [13] use probes for byte-code instrumentation in Java applications. They require insertion of watch points in software, to observe value of selected parameters, return values etc. These tools require selective instrumentation of code, based on what needs to be observed. However, selective instrumentation requires understanding of internal behavior of software, which is a limitation for testing.

Tools like JIE [17], Aprobe [4] and instr [12] use probes for source-code instrumentation of Java programs, for method-level tracing, test coverage, execution logs, debugging etc. But, the trace needed to understand the behavior of software itself needs to be understood owing to its large size.

There are tools that verify the state of object. Payne et al. [15], Turner et al. [6] inject assertions in source code and monitor the state-space. But, true assertion only verifies the state of object. It does not display internal execution details. False assertion evaluation triggers an exception. McGregor et al.[14], Tse et al. [26] emphasize state-based testing and provide observer methods to check externally observable states. Murphy et al. [11] provide state-reporting methods with every class. But, the tools for state verification do not provide internal execution details of source code.

There are limitations of these tools since they either allow selective instrumentation or provide large internal execution details that are difficult to handle. COTT handles large probe output using the simple technique of selective activation and deactivation of probes. COTT addresses the issue of observing internal execution details at unit, integration and system levels, required during testing of object-oriented software.

Test coverage: Tools like Panorama JavaTest [20], CodeWork JCover [7] and TestWorks' TCAT [27] instrument the code to generate test coverage report at class, method, statement and branch levels. Lingampally et al. [21] describe a Java-based tool JavaCodeCoverage for test coverage reporting. It records test coverage for various code-elements and updates coverage information when the code being tested is modified.

These tools provide coverage at different levels like class, method, statement, branch level, but do not address coverage at inheritance and dynamic level for object-oriented software. COTT generates coverage of probes at inheritance and dynamic binding levels, in addition to coverage of probes at class and method levels.

Controllability: There are fault injection tools that create or fake faults during testing of object-oriented software, to create difficult-to-test conditions. The FIG tool [18] focuses on testing of software against failures in underlying system environment. Manaseer et al. [25] [25]propose a special fault injection technique for memory faults. Houlihan [19] describes a targeted software fault insertion subsystem, to target common failures and errors during testing of distributed file system software. Bieman et al. [16] extend C-Patrol assertion insertion system for injecting application-level faults like initialization, assignment and function fault to increase coverage.

There are limitations to these tools and techniques as they are designed to address specific kinds of faults. A single tool is not capable of handling creation of different difficult-to-achieve states required for testing of difficult-to-test conditions. COTT allows tester to create different difficult-to-achieve states required during testing of object-oriented software.

VIII. CONCLUSION

Testability of software is defined as ease of performing testing. Observability and Controllability are the two key facets of testability. COTT is a framework that helps to design testable software. COTT is an interactive tool and is well-suited for observing internal execution details at unit, integration and system levels for errors, understanding design of software and for generating test coverage reports at inheritance and dynamic binding levels.

In the future, we plan on extending this tool. We plan to create library of control methods to help during instrumentation with control commands. We also plan to design generic software dashboard with standard controls, to observe internal execution details of user program.

ACKNOWLEDGMENT

We are thankful to Dr Mukul Sinha, Director, Expert Software Consultants Limited, to allow application of the testing tool to different software projects – UIServer, SMS Java, Netram and CIC-Drishya. All the four software are operational. UIServer and SMS-Java were projects done for other companies. UIServer software was developed for US-based client. Netram and CIC-Drishya are products developed by the company. CIC-Drishya is installed in NIC, Delhi. We wish to thank Tanmoy, Lily, Pawan and Ramakant for their help in implementation of testing tool.

REFERENCES

[1] A. Goel, S. C. Gupta, S. K. Wasan: "Probe Mechanism for Object-Oriented Software Testing", In Mauro Pezze, editor, *Proceedings of Fundamental Approaches to Software Engineering (FASE 2003), Lecture Notes in Computer Science, LNCS 2621, Springer*, Warsaw, Poland, April 2003, pp. 310-324

[2] A. Goel, S. C. Gupta, S. K. Wasan: "Controllability Mechanism for Object-Oriented Software Testing", *In Proceedings of Asia-Pacific Software Engineering Conference (APSEC), IEEE Computer Society Press*, Chiang Mai, Thailand, Dec. 2003, pp. 98-107

[3] A. Orso, S. Silva, "Open Issues and Research Directions in Object-Oriented Testing", *Proceedings of 4th International Conference on*

Achieving Quality in Software, Software Quality in the Communication Society (AQUIS), Venice, Italy, February1998
[4] Aprobe Available:http://www.ocsystems.com
[5] B. Pettichord: "Design for Testability", *In Proceedings of STARWEST* Anaheim, California, Nov 2002
[6] C. D. Turner, D. J. Robson: "A State-Based Approach To The Testing Of Class-Based Programs", *Software Concepts and Tools*, vol. 16, no. 3, 1995, pp. 106-112
[7] Codework JCover. Available: http://www.codework.com/JCover/
[8] Compaq Jtrek. Available: http://www.compaq.com/java/download/jtrk
[9] D. E. Perry, G. E. Kaiser, "Adequate Testing and Object-Oriented Programming" *Journal of Object-Oriented Programming*, Vol 2, No. 1, 1990, pp. 13-19
[10] G. A. Cohen, J. S. Chase, D. L. Kaminsky: "Automatic Program Transformation with JOIE", *In USENIX Annual Technical Symposium*, 1998, pp. 167-178
[11] G. C. Murphy, P. Townsend, P. S. Wong: "Experiences with Cluster and Class Testing", *Communication of ACM*, Sept 1994, pp. 39-47
[12] Glen Mc Clunskey & Associates LLC: Java source code instrumentation. Available: http://www.glenmccl.com/instr
[13] H. B. Lee, B. G. Zorn: "BIT: A Tool For Instrumenting Java Bytecodes", *In Proceedings of USENIX Symposium on Internet Technologies and Systems*, Monterey, California, Dec 1997, pp. 73-82
[14] J. D. McGregor, T. D. Korson: "Integrated Object oriented Testing and Development Processes", *Communication of ACM*, Sept 1994, pp. 59-77
[15] J. E. Payne, R. T. Alexander, C. H. Hutchinson: "Design-for-Testability for Object Oriented Software", *Object Magazine, SIGS Publications Inc.*, vol. 7, no.5, 1997, pp. 34-43
[16] J. M. Bieman, D. Dreilinger, L. Lin: "Using Fault Injection to Increase Software Test Coverage", *In Proceedings of International Symposium on Software Reliability (ISSRE)*, White Plains, New York, Oct 1996, pp.166-174
[17] Java Instrumentation Engine. Available: http://dl.tromer.org/jie
[18] P. Broadwell, N. Sastry, J. Traupman. FIG: "A Prototype Tool for Online Verification of Recovery Mechanisms", *In Proceedings of Workshop on Self-Healing, Adaptive and Self-MANaged Systems (SHAMAN)*, New York, June 2002
[19] P. J. Houlihan: "Targeted Software Fault Insertion", *In Proceedings of STAREAST* , Orlando, Florida, May 2001
[20] Panorama. Available: http://kb.panorama.com/JavaTest/
[21] R. Lingampally, A. Gupta, P. Jalote, "A Multipurpose Code Coverage Tool for Java, *In HICSS. 40th Annual Hawaii International Conference on System Sciences*, 2007.
[22] R. S. Freedman, "Testability of Software Components", *IEEE Transactions on Software Engineering*, vol. 17, No. 6, June 1991, pp. 553-563
[23] R. V. Binder: "Design for Testability in Object-oriented Systems", *Communication of ACM*, Sept 1994, pp. 87-101
[24] S. C. Gupta, M. K. Sinha: "Improving Software Testability by Observability and Controllability Measures", *13th World Computer Congress,IFIP*, vol. 1, 1994, pp. 147-154
[25] S. Manaseer, F. A. Masoud, A. A. Sharieh, "Testing Loaded Programs Using Fault Injection Technique", *Proceedings of World Academy of Science and Engineering and Technology (WASET)*, Vol. 3 Dec. 2004.
[26] T. H. Tse, Z. Xu, "A Formal framework for Improving Object oriented Software Testing", *13th International Conference on Testing Computer Software*, Washington DC, 1996
[27] Testwork's TCAT. Available: http://www.soft.com/TestWorks/

Dr. Anita Goel has got her Ph.D. in Computer Science from Jamia Millia Islamia, Delhi, India in 2005. She did her Masters in Computer Applications from Department of Computer Science, University of Delhi, India. Her research interests include software testing, object-oriented software, aspect-oriented software, distributed systems, web services, service-oriented architecture, software testability, web testing, software maintenance, and operating system.

She is reader in Department of Computer Science, Dyal Singh College, University of Delhi, India. She has 20 years of experience in teaching and research. She has about 10 national and international publications, including papers in LNCS, Springer, and IEEE Computer Society Press.

Dr. Goel is a life member of Computer Society of India. One of her co-authored paper received the "Best Paper Award" in conference in Delhi.

Dr. S.C. Gupta received B.Tech (Electrical Engg.) degree from Indian Institute of Technology, New Delhi in 1975 and PhD degree (Software Engg) from Birla Institute of Technology and Science, Pilani in 1993. His research interests include software testability, distributed databases, grid computing and information security.

He is currently Senior Technical Director in National Informatics Centre (NIC), CGO complex, New Delhi. He has over 30 years of experience in research and development and has guided over 100 B.Tech and M.Tech. projects. He has over 15 national and international publications, including papers in IFIP Congress, LNCS Springer, and IEEE Computer Society Press.

Dr Gupta has visited Sydney University, Australia, on UNDP fellowship in 1983.

Dr..S. K. Wasan is PhD from Delhi, He did his M.S. from Oregon. His field of interest includes coding theory, data mining and software testing.

He is Professor in Department of Mathematics, Jamia Millia Islamia, Former Registrar University of Delhi (1992-1996) and Dean Faculty of Natural Science, Jamia Millia Islamia (1998-2000). He has taught at Ramjas College (University of Delhi), University of Oregon and University of Aden. He has published research papers in international journals on coding theory, data mining, application of data mining to healthcare and software testing.

Prof. Wasan is Fellow IETE, Member of Society of Mathematical Sciences and Visitor's nominee to first Academic Council of Assam University. He visited Thammasat University (Bangkok) and University of Philippines on a Regional Fellowship of Third World Academy of Sciences.