Probe Mechanism for Object-Oriented Software Testing

Anita Goel[1], S.C. Gupta[2], and S.K. Wasan[3]

[1] University of Delhi, Dyal Singh College
New Delhi-110003, India
aagoel@vsnl.com
[2] National Informatics Center, NIC, A Block
New Delhi-110003, India
scgupta@hub.nic.in
[3] Jamia Millia Islamia, Department of Mathematics
New Delhi-110025, India
skwasan@yahoo.com

Abstract. This paper presents a probe-based testing technique that facilitates observing internal details of execution at different levels of abstraction-unit, integration and system levels, during testing of object-oriented software. Our technique adapts probe, an observability measure, to suit the testing needs of object-oriented software. It uses source-code instrumentation, which requires probes to be pre-determined and pre-built in the software during the development phase. Test coverage reports are generated from the information gathered by the executed probes. It includes coverage of probes at probe, method, class, inheritance, regression and dynamic binding levels. During regression testing, our technique helps in the selection of test cases that must be re-executed. Furthermore, the log generated by active probes can be used for post-analysis.

1 Introduction

The unique architecture and features of object-oriented software introduce new kind of errors in the software. As a result, some issues involved in the testing of object-oriented software are different from the conventional software testing issues [1, 2]. In order to handle the unique testing issues of object-oriented software, the conventional software testing techniques require improvisation or new ones need to be developed.

During testing, a correct output result does not always ensure correctness of processing. An incorrect state may not be reflected in the output, but it governs the future behavior of methods. An incorrect output will have to be diagnosed in terms of the various execution steps. This requires access to the internal behavior [20].

Object-oriented software is tested at unit, integration and system levels. Class is the basic unit of testing. Class is composed of data structure and a set of methods. Objects are run time instances of the class. The data structure defines the state of the object that is modified by the methods defined in the class. The encapsulation feature of object-oriented software hides the data structure, posing difficulty in accessing the

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 310–324, 2003.
© Springer-Verlag Berlin Heidelberg 2003

state of the object, which is essential for verifying the correctness of processing and for error diagnosis. The already tested units integrated via relationships like inheritance and aggregation are tested during integration testing. The focus is on the interaction among the units. System testing at input/output level requires information about the interaction among the integrated units.

We see that the focus of testing shifts as we move from unit to system testing. Correspondingly, the focus of internal execution details observed during testing must also shift as we move from unit to system testing. Several techniques exist [5, 12, 14, 15, 16] that verify/display the state of the object. Tools exist that provide method-level or statement-level trace [7, 11] of execution details or allow specific values to be observed [6, 9, 13, 18] during testing. But, none of them address the issue of observing internal execution details at different levels of abstraction during testing,

In this paper, we focus on observing internal execution details at different levels of abstraction- unit, integration and system levels, during testing of object-oriented software. During unit testing, the input/output of the methods and the impact of method execution on the state of the object are observed. The sequence of execution of classes and input/output of the class is observed during integration testing. The input/output of the integrated units is observed during system testing.

Here, we present a probe-based testing technique that adapts *probe-an observability measure* for object-oriented software testing. It uses source-code instrumentation. Probes are pre-determined and pre-built in the software during design and coding phases, for observability needed at different levels of abstraction. During testing, probes are externally activated/deactivated, facilitating visual display of execution details at unit, integration and system levels. Probes can be turned off when internal execution detail is not needed. Probes left embedded in software results in creation of *testable software* for further modifications and easy *corrective maintenance*.

The coverage of inheritance hierarchy and dynamic binding relationship is needed to ensure adequate testing of these relationships. Our technique uses the internal execution details to generate test coverage of probes at probe, method, class, *inheritance and dynamic binding* levels. Due to the iterative and incremental nature of object-oriented software, regression techniques are needed during development phase as well as during maintenance. Our technique helps to identify the test cases to be re-executed during *regression testing.* It also generates coverage of the changed unit.

A probe-based testing tool based on the probe-based testing technique has been developed. It is implemented in Java and designed using use case driven object-oriented approach. The probe-based testing technique has been applied to test UIS-erver-a translator software that translates UIML (User Interface Markup Language) document to WML (Wireless Markup Language) or CHTML (Compact- HyperText Markup Language) document. UIServer has been developed using Java and XML.

In this paper, Section 2 discusses probe- an observability measure. Section 3 describes the probe-based testing technique. In section 4, the probe-based testing technique during testing phase is discussed. Section 5 describes in brief, the experience in testing of UIServer software using our technique. Section 6 describes related work. Section 7 states the conclusion.

2 Probe - An Observability Measure

Observability measures are provisions in the software that facilitate observation of internal and external behavior of the software, to the required degree of detail [20]. The need to build observability in object-oriented software has been stressed by Binder [17]. According to Binder, "if you cannot observe the output of a component under test, you cannot be sure how a given input has been processed".

Traditionally, print statements and debuggers have been used, to get access to internal information. Print statements require frequent commenting, uncommenting and recompilation of code each time changes are made. As we move from unit testing to system testing, deciding what to comment/uncomment becomes difficult. It requires intimate knowledge of the software. Moreover, print statements are not structured, posing hindrance in the analysis of internal information details. Debuggers give access to all the information, at a point in execution. But what is important to observe becomes harder and harder to decide, as the size of the software increases.

The concept of probe is an effective observability measure for observing internal details of the software. Probe is a method invocation having a structured probe identifier and a message [20]. The syntax of probe method invocation is *probe(probe_id, probe_message)* where, *probe* is a method name, *probe_id* is a unique structured identifier identifying the location of *probe_message* and *probe_message* contains state-related attributes, parameter values, temporary variables or a message, relevant at the location of probe in the code. Probes are inserted in the software at locations where information relevant at that point is needed. During execution of software, probe can be activated/deactivated and probe breakpoints can be set, externally. An active probe on execution generates a message carrying the internal information along with its identity. Probes can be turned on/off as and when required. Probes do not interfere with the logic of the software and can be left embedded in the software.

3 Probe-Based Testing Technique

The probe-based testing technique facilitates observation of internal execution details during testing of object-oriented software. The execution details consist of the-

Class being executed
Method of the class being executed
Value of parameters or messages at method entry/exit or within the 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 idea behind our testing technique is quite simple. Our technique uses probe, an observability measure, to observe the internal details of execution during testing. Probes are inserted in the source-code during software development. During testing, probes are controlled externally– activate/deactivate, to display execution details at unit, integration and system levels. A point to be noted here is that our technique gathers internal details of execution from the probes inserted in the software. Thus, the content

of the probe must be carefully decided and probes must be inserted at proper locations. Our technique is defined in three phases -

(1) *Probe Structure* defines the structure of *probe_id* and *probe_message*.
(2) *Probe Insertion* defines locations in the software where probes are to be inserted.
(3) *Probe Subsystem* defines the different subsystems that operate on the software embedded with probes, during testing.

Using our technique requires- defining the probes, inserting probes in the software and using probe subsystem during testing. The software developer should have knowledge of our technique during the design and coding phases.

We illustrate our technique with an example of link list, written in Java, shown in Program code 1(relevant details shown). For the time being, we do not focus on the "Log" statements. The program is one integrated unit consisting of three classes (1) UserInterface (2) LinkList (3) Node. UserInterface accepts input from the user to add/delete to link list. LinkList has methods to create list, add/delete nodes from the list. Node has methods to create node with string data, get and set data etc. LinkList creates a link list, where each node is of type class Node. We extend this example to create heterogeneous link list, to show coverage at inheritance and dynamic binding levels (code not shown). Class IntNode is derived from the class Node. IntNode creates node with integer data. A node in link list is of type IntNode or Node. IntNode inherits getNext() and setNext() from Node and defines its own methods getData(), setData(), printData() etc. The method printData() is dynamically bound.

Program code 1. Example of Link List (relevant details shown)

```
class UserInterface{
public static void main(String[] args) {
1.      Log d = new Log("UserInterface");
2.      Log.penter("L1/1", "msg:start main");
        LinkList lst = new LinkList();   // Create a list
             :
             :
    // get string from user to be inserted in the list
    // lst.addNodeAtHead(new String(str));
    // get string to be deleted from the list
    // lst.deleteNode(new Node(str));
             :
        lst.printList();
3.      Log.pexit("L1/5", "msg:end main");
4.      Log.close();  } }
class LinkList{
        private Node head;
public LinkList(){...}                          // head = null
public Node getHead(){...}                      //returns head
public void addNodeAtHead(String s){...}
                              //adds node with data s at head
public void printList(){...}             //prints link list
public void deleteNode(Node n){
        Log.penter("L2/9", "delete node=" + n);
        if (head == null) {
            Log.pexit("L2/10","msg:list empty");
```

```
            return;}
        if (n.equals(head)) {
            head = head.getNext();
5.          Log.pmsg("L3/11","msg:node deleted at head");}
        else { Node p = head, q = null;
              while((q = p.getNext()) != null) {
                if (n.equals(q)) {
                   p.setNext(q.getNext());
                   Log.pmsg("L3/12","deleted node=" + q);}
                else p = q;  }}
        Log.pexit("L2/13","msg:end delete node");
        return;}
class Node  {
        private String data;
        private Node next;
public Node(String wrd){...}     //initialises data = wrd
protected Node getNext(){...}        // returns next node
protected void setNext(Node n){...}      //sets next to n
public String getData(){...}        //returns string data
public void setData(String s){...}       //sets data to s
public void printData(){...}               //prints data
public boolean equals(Object o){
        Log.penter("L2/13", "compare=" + o);
        boolean bool = false;
        if (this == o)  {
             Log.pmsg("L3/14", "msg:object equal");
             bool = true; }
        else if (!(o instanceof Node)) {
             Log.pmsg("L3/15", "msg:object not equal");
             bool = false; }
        else if ((((Node)o).data).equals (this.data)) {
             Log.pmsg("L3/16", "msg:data equal");
             bool = true; }
        else { bool = false;
               Log.pmsg("L3/17", "msg:not equal"); }
        Log.pexit("L2/18", "comparison was= " + bool);
        return bool;  } }
```

3.1 Probe Structure

Our technique associates level number with the probe, to display execution details at different levels of abstraction. Also, each probe must be uniquely identifiable, to locate probe in the code. Probe structure defines *probe_id* as- "*level_number/probe_number"* where, *level_number* indicates the testing level. The *level_number* is L1, L2 and L3 for probes that display interaction among the integrated units (system level testing), interaction among the classes of integrated unit (integration level testing) and code level behavior of a class (unit level testing) respectively. The *probe_number* uniquely identifies a probe in a class. For each class, it starts from 1 for the first probe and is incremented for every other probe in the class. It may be in any order within the class. E.g. "L2/2" represents an integration level probe having probe number 2.

The structure of *probe_message* is defined as *"variable1:val variable2:val ..variableN:val* msg*:string"* where, val is the value of a variable, and, msg displays a message. E.g. "msg:start main" is interpreted as, *start main* is a message.

The designer and code developer decide the *level_number* and *probe_message* respectively, during the design and coding phases of software development.

3.2 Probe Insertion

Our technique requires probes to be inserted at the beginning, end (before return statement) and anywhere between begin and end of the method (if needed) like in if-statement or loop statements, as shown in Program code 1, line 2, 3, 5 respectively. Probe insertion is the responsibility of the developer.

The methods are of three kinds in object-oriented software- public, protected and private. Public methods are invoked from outside the class. Private methods are invoked by public methods. Protected methods are private to the class but can be invoked from the derived classes. The *level_number* in *probe_id* is decided based on the kind of method and the location of probe in the method. Probes defined at the beginning and end of a method have *level_number* L1 in the public methods of classes that interact with other integrated units, L2 in the public methods of rest of the classes, and, L3 in private and protected methods of a class. Probes defined anywhere in between the beginning and end of any method has *level_number* L3.

To display the hierarchy of execution of classes and the parameter values at method entry/exit, probe insertion defines static methods-- *penter, pexit and pmsg,* for probes inserted in the beginning, end and anywhere in between a method respectively. Probes are inserted as *Log.penter, Log.pexit, Log.pmsg*, as shown in Program code 1, line 2, 3, 5 respectively. The class "*Log*" defined in our technique interacts with the software embedded with probes during testing. To start and stop recording information from probe, line 1 and 4 in Program code 1 are needed respectively.

3.3 Probe Subsystem

The probe subsystem work on the software embedded with probes, during testing. It has four components –Preprocessor, OnLineTest, OffLineTest, and Report. ***Preprocessor*** works on the compiled software before its execution. Its functions are- (1) store details of inheritance hierarchy- classes, declared and inherited methods of each class, and class to which inherited method belongs. (2) Store details of dynamic binding relationship- classes, the dynamically bound methods and method from where the dynamically bound methods are invoked. (3) Store "*level_number/ probe_number/class_name/tag/method_name*" for each probe. It is needed to display the execution details during testing. The *level_number* and *probe_number* are defined in probe_id. *Tag* is probe method *penter, pexit* and *pmsg*. The *class_name* and *method_name* is the class and method respectively, in which the probe is defined. To get class executed during testing, Preprocessor inserts "*class_name*" in probe_id of probes. (4) Insert "*getClass()*" in *probe_message* of probes of inherited methods, to find the class invoking the inherited method. The function *getClass()* must be sup-

ported by the programming language. It is supported in Java. (5) Identify modified classes and help in selection of test cases to be re-executed, during regression testing. ***OnLineTest*** is invoked on execution of the software, for testing. It defines probe settings for probe activation and probe breakpoint to facilitate observation of internal execution details at unit, integration and system levels. Output of active probes is stored in *log file*. Output of all executed probes is also stored in a file. ***OffLineTest*** is invoked after testing, to analyze the log file and to get details about the software-classes, methods etc. ***Report*** is invoked after testing to generate coverage report.

4 Using Probe Subsystem

The components of probe subsystem facilitate observation of internal execution details at unit, integration and system levels, selection of test cases to be re-executed during regression testing, post-analysis of log file and report generation.

4.1 Probe Settings

Probe settings allow the tester to externally control the probes during online and off-line testing. **Probe Activation** defines commands to selectively activate/deactivate probes, externally, during testing. Output of only active probes is displayed on the screen and stored in the log file. Probes are referred to in a generic style so that a group of probes can be addressed through a single command. The format of probe activation command to activate and deactivate probe is

A: *class_name/method_name/level_number/probe_number* (1)

D: *class_name/method_name/level_number/probe_number* (2)

respectively. The *level_number* in (1) and (2) results in activation of probes at the specified *level_number* and at lower *level_number*, and, deactivates probes at the specified *level_number* respectively. E.g. "A:*/*/L2/*" activates all probes having *level_number* L1 and L2 in all classes of all methods. The command "D:Node/*/L3/*" deactivates all probes having *level_number* L3 in all methods of class Node. A '*' used for *level_number, class_name* or *method_name* matches with all *level_number, class_name* and *method_name* respectively.

Probe BreakPoint allows breakpoints to be set on selected probes. On occurrence of break, execution of the software pauses. The tester can observe the already displayed probes and change the probe settings to observe rest of the execution details. Probe breakpoint can be set as follows:

class_name/method_name/level_number/probe_number (3)

A String (4)

In (3), break occurs when the *level_number, class_name, method_name* and *probe_number* are true in the probe being executed. In (4) break occurs when output of a probe contains the specified string. E.g. string "start deleting" results in a break when it is encountered in the probe output.

Probe settings are made during testing based on what is to be observed and at what level of detail. All probe settings are stored in a file and can be modified during testing. All commands in the file are applied sequentially, to find active probes.

4.2 OnLine Testing

During testing, activating/deactivating the *level_number* in probe activation command facilitates observation of internal execution details at different levels of abstraction. The *class_name, method_name* and *probe_number* can be activated or deactivated to "fine-tune" the execution details to be observed during testing. Probe breakpoint is set to observe the already displayed probes.

Table 1. Internal execution details at unit level (delete node from list) of Program code 1 (->enter, <-exit, --between)

Class Name	Method Name	LevelNo./Probe No.
ListClient	main(String[])	L3/3 msg: Start deleting
|→Node |←Node	Node(String)	L2/1->msg: create string node L2/2<-data=stringB
|→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 was=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

Unit Testing: Probes at *level_number* L3 are activated during unit testing. It results in activation of probes at *level_number* L1, L2 and L3. The internal execution detail of deleting a node from the link list, at unit level, is shown in Table 1. It displays code level behavior of the program- the executed public, private and protected (*getNext()*) methods, the part of if-condition executed (*msg:data equal*) and a message in between the method (*msg:node deleted at head*), at begin and end.

Integration Testing: During integration testing, probes at *level_number* L2 are activated. It results in activation of probes at *level_number* L1and L2. The internal execution detail of link list at integration level is shown in Table 2. It displays sequence of execution of classes and their input/output. E.g. for node deletion it displays that the node was compared, found to be equal and deleted. It does not display the internal execution details of the class as shown during unit testing in Table 1.

Table 2. Internal execution details at integration level of Program code 1

Class Name	Method Name	LevelNo./Probe No.
ListClient	main(String[])	L1/1->msg: start main
|→List |←List	List()	L2/1->msg: List() L2/2<-head=null

|→List	addNodeAtHead(String)	L2/5->add node=stringA
|→Node |←Node	Node(String)	L2/1->msg: create string node L2/2<-data=stringA
|←List	addNodeAtHead(String)	L2/6<-added at head=Node@e96ff324
|→List	addNodeAtHead(String)	L2/5->add node=stringB
|→Node |←Node	Node(String)	L2/1->msg: create string node L2/2<-data=stringB
|←List	addNodeAtHead(String)	L2/6<-added at head=Node@2debf324
|→Node |←Node	Node(String)	L2/1 ->msg: create string node L2/2 <-data=stringB
|→List	deleteNode(Node)	L2/9->delete node=Node@f133f325
|→Node |←Node	equals(Object)	L2/13->compare=Node@2debf324 L2/18<-comparison was=true
|←List	deleteNode(Node)	L2/13<-msg: end delete node
|→List	printList()	L2/14->msg: print list
|→Node |←Node	printData()	L2/11->msg: print string data L2/12<-data=stringA
|←List	printList()	L2/15<-msg: list printed
ListClient	main(String[])	L1/5<-end main

System Testing: Probes at *level_number* L1 are activated to observe interaction among the integrated units of the system.

Corrective Maintenance: It involves fixing reported errors in released and in-use software. For this, there is a need to understand the software to identify the component containing the error and to locate erroneous code in it. Since maintenance team is generally not same as development team, understanding the software is difficult due to its large size and complexity. The erroneous code cannot be isolated using the limited information available at user interface.

Our technique helps in the corrective maintenance activity. The software is re-executed with probes on, with input that resulted in error. Observing the internal execution details at system level, integration level and unit level helps to identify the integrated unit, the class of the integrated unit, and, the method and probe of the class displaying incorrect value, respectively. The *probe_number* in the probe displaying incorrect value is used to locate the probe in the class. Code near the probe is examined to isolate erroneous piece of code.

4.3 Regression Test Cases

Regression testing is the process of validating modified software to detect whether new errors have been introduced into previously tested code and to provide confidence that modifications are correct [8]. An overview of regression testing techniques is given in [8][21]. Here, we focus on selection of regression test cases. Having known the unit (method, class etc.) of change (add, delete, modify), there is a need to select the test cases to be re-executed, from the already executed test cases.

Our technique identifies class as the unit of change. It includes classes in the hierarchy, if changed class is part of inheritance hierarchy. Preprocessor identifies the test cases to be re-executed in two steps. First, it identifies the test cases containing the

changed class name, from the file storing all executed probes. It includes classes impacted by changed class, since the impacted class invokes it or is invoked by it. Next, from this subset, test cases containing unique sequence of *probe_number, class_name* (invoking the changed class - of the changed class - invoked by changed class) is identified. The resulting test cases are to be re-executed.

4.4 OffLine Testing

OffLineTest facilitates analysis of log file using probe setting defined in OnLineTest. Probe settings allow the tester to "play" with the contents of log file. The behavior of a class or a method can be observed/analyzed. For very large software, execution details of a portion of the software can be observed.

Table 3. Information about the Program code 1

Class Name	Method Name	LevelNo./Probe No
ListClient	main(String[] args)	L1/1 L3/2
	:	
	:	
Total Classes=3	Total methods=13	Total Probes=36

OffLineTest also provides details of the software- (1) classes, methods and probes. (2) Inheritance hierarchy, inheritance category, and, declared and inherited methods of each class, and (3) Dynamic binding relationship. It also provides count of these relationships, as shown in Table 3. It retrieves these details from Preprocessor.

4.5 Report Generation

All probes traversed (active/inactive) are logged, which, help to generate coverage of probes at- probe, method, class, inheritance, regression and dynamic binding levels. The logged probes can also help to identify covered path sequences, for test data adequacy. Coverage reports can be viewed as *%coverage* and *in the form of list*, for- *a single, set* and *all test cases combined*. The *list of uncovered probes* generated at these levels helps to locate untested code. To calculate coverage, Report gathers details of the executed and total probes from OnLineTest and Preprocessor respectively.

Probe, Method and Class Coverage: The coverage of probes at probe, method and class levels is displayed as (*class_name,probe_number1,…,*), (*class_name, method_name1,…*) and (*class_name*) respectively, as shown in Table 4.

Table 4. Coverage at probe, method and class levels of Program code 1

Uncovered Probes	Uncovered Methods	Uncovered Classes
(Node, 7, 8, 9, 10, 14, 15, 17) (List, 10, 12,)	(Node, getData(), setData(String))	None
%coverage: 75.0%	% coverage: 84.61539%	%coverage:100.0%

Inheritance Coverage: The creation or modification of an inheritance hierarchy falls in four categories [1][2]-

(A) Creation/Modification of superclass
(B) Creation/Modification of subclass
(C) Modification of superclass data structure from subclass, and,
(D) Pure extension of superclass.

Testing for category

(A) and (C) necessitates retesting of superclass, all subclasses and units dependent on it

(D) requires testing of subclass only, and,

(B) requires testing of subclass and retesting of inherited methods of superclass. Retesting is needed because inheritance provides new context for the inherited methods. The inherited methods, thus, must be executed from the class in which it is declared and from the class inheriting it.

Our technique calculates coverage at inheritance level based on inheritance category. Inheritance coverage is displayed as (*probe_number/class_name/method_name /class invoking the inherited method*). The *class invoking the inherited method* gets value from *getClass()* (inserted by Preprocessor). It is blank if method is executed from the class in which it is declared. As shown in Table 5, getNext(), setNext() defined in class Node are invoked from Node itself, statement (1), (3) and from IntNode (inheriting class), statement (2), (4).

Table 5. Coverage at inheritance level (of inherited methods)

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

Dynamic Binding Coverage: Testing a dynamic binding relationship requires testing of all possible methods that can get bind at runtime for a single method invocation. Coverage at dynamic level is displayed as (*probe_number/class_name/ method_name/method_name from where invoked*). The "*method_name from where invoked*" is the method enclosing call to the dynamically bound method. As shown in Table 6, printData() defined in Node and IntNode, is invoked from method printList().

Table 6. Coverage at dynamic binding level (as list of covered probes)

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

Regression Coverage: Coverage at regression level is displayed as coverage at probe level of the modified unit. It includes coverage of probes at inheritance level if modified class is part of the inheritance hierarchy.

5 A Case Study - UIServer

The probe-based testing technique has been used during testing of an XML based software-UIServer, written in Java. UIServer operates in the web environment. It translates a UIML (User Interface Markup Language) document to WML (Wireless Markup Language) or CHTML (Compact-HyperText Markup Language) document.

UIServer consists of three integrated units- Validate, ParseTree and GenerateTML. *Validate* receives URL of the UIML document and the http request-UAData (UserAgentData) from the web engine. It fetches the UIML document, checks its validity according to UIML 2.0a DTD (Document Type Definition- it defines the structure of UIML document. It specifies the tags, attributes of these tags and their arrangement with each other. UIML document can contain 28 tags.). Validate returns valid/invalid UIML document. If document is valid, *ParseTree* is invoked. It parses the UIML document and builds a tree based on the tag hierarchy defined in UIML2.0a DTD. It stores the tags defined in UIML document as nodes of the tree. If no error found, GenerateTML is invoked. *GenerateTML* uses the tree to generate WML/CHTML document as the output. In case of error, an error message is output.

Our technique was used to observe the internal execution details at different levels of abstraction during the testing of UIServer. We discuss an instance, where without our technique, locating the erroneous code was a difficult task. During system testing, for a UIML document as input, the WML document generated was incorrect. No error message was displayed. Observing the internal execution details at system level displayed error in the output of the integrated unit- ParseTree. Next, the execution details observed at integration level, displayed error in the output of class ParseUIML, of ParseTree. While observing the internal execution details of ParseUIML at unit level, it was found that the code developer had not taken any action for the tag <attribute> defined in the tag <call> of UIML document. The error was notified.

Table 7. Characteristics and test execution result of UIServer

#of lines of code	#of probe inserted	Execution time with probes on	Execution time with probes off	Coverage at probe level	Coverage at method level	Coverage at class level
8994	705	5.85s	5.60s	84%	88%	100%

Table 7 shows characteristics of UIServer and test execution result using our technique. It includes number of lines of code and probes in the software, execution time with probes on and off, and coverage of probes at probe, method and class level. The uncovered probes helped to locate the untested portion of code.

6 Related Work

This work is related to observability measures in object-oriented software and testing of object-oriented software.

6.1 Observability Measures

Previous research focuses on observing state of the object during testing of object-oriented software. Assertions, [5][12] are injected at key locations in the source code and state-space monitored. Assertion monitoring does not display internal execution details as long as assertions evaluate to true. It only verifies state of the object. False assertion evaluation triggers an exception. McGregor and Korson [14] emphasize functional testing and provide observer methods to check externally observable states. Murphy et al. [16] provide state-reporting methods with every class. Dorman [15] uses friend declaration to observe private data of C++ program. To check method call sequence, it inserts callback functions before and after the call.

Probes are used in tools to trace execution details or observe values of specific variables during testing. Compaq's JTREK, JOIE [6] and BIT [9] use byte-code instrumentation for troubleshooting Java applications. It requires watch points to be inserted in the software, to observe values of selected parameters, return values etc. The selective instrumentation, based on what needs to be observed is a limitation for testing. It requires understanding of the internal behavior of the software. JIE [11] and instr [7] use 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.

Probes are also used for profiling in compiler optimization feedback and performance tools. Hundt [18] describes HP Caliper, which uses dynamic instrumentation to provide a framework for building tools for performance analysis, coverage analysis and correctness checking. DeRose et al. [13] describe- Dynamic Probe Class Library (DPCL) an infrastructure for developing instrumentation for performance tools.

Tools like Panorama JavaTest, CodeWork JCover and TestWorks' TCAT instrument the code to generate test coverage report at class, method, statement and branch levels. They also display the executed and the unexecuted part of the code.

6.2 Testing Object-Oriented Software

Unit Testing: Most of the research work in class testing is based on generation of method sequences as test cases, execution and result validation. Souter et al. [3] develop a code-based testing and analysis tool– TATOO to generate test tuples based on object manipulation. Chen et al. [10] develop a tool TACCLE that selects fundamental pairs of equivalent ground terms and checks the observational equivalence of resulting objects. Turner et al. [5] use automata to model internal representations of the class, generate test cases using state-based test suites and verify the final state.

Integration testing: According to Chen et al. [10], little study has been made on cluster-level testing or its relationship with class-level testing. They perform static cluster testing on horizontal interactions among classes using individual message-passing rule. Jin and Offutt [22] generate test cases using coupling based criteria.

Dynamic binding: Alexander and Offutt [19] extend coupling-based testing to detect faults from polymorphic relationships among components. Orso et al. [4] extend traditional data flow test selection criterion. They define def-use sets to select

execution paths that can reveal failures due to incorrect combination of polymorphic calls. Payne et al. [12] implement inheritance hierarchy such that pre and post conditions made in base class are not violated by polymorphic methods.

Inheritance Testing: Dorman [15] uses call-interface instrumentation to determine coverage of inherited methods of base class in context of their usage. Payne et al. [12] use down-call technique to restrict inheritance to be a sub-typing relationship. Retesting of base class in context of the derived class is not needed.

7 Conclusion

In this paper, we present a probe-based testing technique that adapts probe-an observability measure for object-oriented software testing. It facilitates observation of internal execution details at different levels of abstraction- unit, integration and system levels, during testing. Our technique requires probes to be pre-determined and pre-built in the software during design and code phases of software development. During testing, probes are externally activated/deactivated facilitating observation of internal details of execution at unit, integration and system levels. Coverage of inheritance hierarchy and dynamic binding relationship generated using our technique helps in determining the adequacy of testing these relationships. Our technique also aids in the selection of regression test cases. Probes can be turned off when the internal execution details are not needed. Probes left permanently embedded in the software results in creation of testable software for further modifications and easy corrective maintenance.

Acknowledgement

We are grateful to Dr. Mukul Sinha, Director, Expert Software Consultants Limited, for his valuable comments and suggestions in the development of the testing technique and application of our technique to UIServer project. We wish to thank Tanmoy, Pawan and Ramakant for their help in the implementation of the concepts discussed in this paper.

References

1. A. Goel, S.C. Gupta, K. D. Sharma: Object Oriented Testing: An Overview. In: Proceedings of International Conference on Software Engineering And Its Application, 91-99, Hyderabad, India, Jan (1997)
2. A. Goel, S.C. Gupta, S. K. Wasan: Object Oriented Software Testing: A Survey Report. In: Proceedings of 2nd Annual International Software Testing Conference, QAI, Bangalore, India, Jan (2001)
3. A. L. Souter, T. Wong, S. Shindo, and Lori L. Pollock: TATOO: Testing and Analysis Tool Object-Oriented Software. In: Proceedings of 7th International Conference, Tools

and Algorithms for the Construction and Analysis of Systems, held as part of ETAPS, April (2001)
4. A. Orso, M. Pezze: Integration Testing of Procedural Object-Oriented Programs with Polymorphism. In: Proceedings of 16th International Conference on Testing Computer Software, Washington D.C., June (1999)
5. 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, 106-112, (1995)
6. G. A. Cohen, J. S. Chase, D. L. Kaminsky: Automatic Program Transformation with JOIE. In: USENIX Annual Technical Symposium, 167-178, (1998)
7. Glen Mc Clunskey & Associates LLC: Java source code instrumentation. http://www.glenmccl.com/instr
8. Graves, Harrold, Kim, Porter, Rothermel: An Empirical Study of Regression Test Selection Techniques. In: ACM Transactions on Software Engineering and Methodology, Vol 10, No. 2, 184-208, April (2001)
9. H. B. Lee, B. G. Zorn: BIT: A Tool For Instrumenting Java Bytecodes. In: Proceedings of USENIX Symposium on Internet Technologies and Systems, 73-82, Monterey, California, Dec (1997)
10. H. Y. Chen, T. H. Tse, T. Y. Chen: TACCLE: A Methodology for Object-Oriented Software Testing at the Class and Cluster Levels. ACM Transactions Of Software Engineering And Methodology, Vol. 10, No. 4, 56-109, Jan (2001)
11. Java Instrumentation Engine: http://dl.tromer.org/jie
12. 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, 34-43, (1997)
13. Luiz DeRose, Ted Hoover Jr., J. K. Hollingsworth: The Dynamic Probe Class Library – An Infrastructure for Developing Instrumentation for Performance Tools. In: Proceedings of International Parallel and Distributed Processing Symposium (IPDPS), April (2001)
14. McGregor, Timothy D. Korson: Integrated Object oriented Testing and Development Processes. Communication of ACM, 59-77, Sept (1994)
15. Misha Dorman: C++ "It's Testing Jim, But Not As we Know It. In: Proceedings of EuroSTAR, Edinburg, Scotland, Nov (1997)
16. Murphy, Paul Townsend, Pok Sze Wong: Experiences with Cluster and Class Testing. Communication of ACM, 39-47, Sept (1994)
17. Robert V. Binder: Design for Testability in Object Oriented Systems. Communication of ACM, 87-101, Sept (1994)
18. Robert Hundt: HP Caliper- An Architecture for Performance Analysis Tools. In: Proceedings of WIESS , San Diego, California, USA, Oct (2000)
19. R. T. Alexander, A. J. Offutt: Analysis Techniques for Testing Polymorphic Relationships. In: Proceedings of TOOLS, Santa Barbara, California, USA, August (1999)
20. S. C. Gupta, M. K. Sinha: Improving Software Testability by Observability and Controllability Measures. 13th World Computer Congress, IFIP, 94, Vol 1, 147-154, (1994)
21. Yuejian Li, N J Wahl: An Overview of Regression Testing. Software Engineering Notes, ACM SIGSOFT, 69-73, Jan (1999)
22. Z. Jin and A. J. Offutt: Coupling-based Criteria for Integration Testing. Journal of Software Testing, Verification and Reliability, Vol. 8, No. 3, 133-154, Sept. (1998)