

INTRODUCING TESTABILITY CONSIDERATIONS IN SOFTWARE DEVELOPMENT PHASES

THESIS

**Submitted in partial fulfilment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY**

By

SURESH C. GUPTA

**Under the Supervision of
DR. MUKUL K. SINHA**

**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN) INDIA**

1993

Acknowledgements

I am grateful to Dr. Mukul K. Sinha, my thesis supervisor, for his valuable guidance and encouragement throughout the course of this study. I thank him for the great amount of time he has devoted to innumerable discussions and reviews of the drafts of this thesis, often taking time out from his consultancy work.

I am highly indebted to Dr. N. Seshagiri, Special Secretary & Director General, National Informatics Centre (NIC), Planning Commission, Government of India, for encouragement and permission to continue this work, and for organizational support. I thank him for the valuable advice which helped in finishing successfully and with great satisfaction, the otherwise difficult task of writing this thesis. I gratefully acknowledge his grant of leave which enabled me to concentrate exclusively on writing.

I began this work while I was at Expert Software Consultants (ESC) Ltd. I am grateful to Mr. L.N. Rajaram, Director, ESC for the valuable technical discussions I had with him during the initial period of this research.

I am also grateful for the opportunity to work on the TIFACLINE Host Software Project, a large and complex distributed database software project of the Technology Information Forecasting and Assessment Council (TIFAC), Department of Science and Technology, Govt. of India. It was this project which provided the challenge, and the fertile ground for experimenting with some of the ideas in practice as well as for judging the usefulness of other concepts evolved during this research.

The contribution of some of the students from IIT, Delhi, JNU, and BITS, specially Ms. Radhika Ramnath and Ms. Meenu Gupta, in implementing some of the concepts presented in the thesis, earlier at ESC and later at NIC, is gratefully acknowledged.

I am grateful to Mr. M. Moni, Technical Director, National Informatics Centre (NIC), for the final reading of the thesis, which has been helpful in improving the flow of the text. Assistance of my colleagues, Mr. P. Venkatesan and Mr. Chandra Kumar, in taking innumerable printouts during the course of the preparation of this thesis, is gratefully acknowledged.

I am grateful to my parents, my wife, Amita, and my sons, Gaurav and Saurabh, for allowing me time to devote to this research, specially on weekends.

This acknowledgment would be incomplete without thanking my Guru Maharaji for all the Grace he bestowed upon me in this endeavor.

Signature : 

Name : Suresh C. Gupta

Designation : Technical Director

Date : Nov 14, 1993

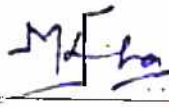
Organization: National Informatics Centre

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI RAJASTHAN

CERTIFICATE

This is to certify that the thesis entitled INTRODUCING TESTABILITY
CONSIDERATIONS IN SOFTWARE DEVELOPMENT PHASES and submitted by
Shri SURESH C Gupta ID. No. 89 PHX F802 for award of Ph.D. Degree of the
Institute, embodies original work done by him under my supervision.

Signature in full of
the Supervisor



Name in capital block
letters

DR MUKUL K SINHA

Date: Nov 18, 1995.

Designation

DIRECTOR

TABLE OF CONTENTS

Chapter 1 : Introducing Testability Considerations in Software Development Phases

1.1	Introduction	1
1.2	Testability & Maintainability of Software	3
1.3	Testability : Controllability & Observability	4
1.4	Controllability Measures	5
1.5	Observability Measures	6
1.5.1	Probe Mechanism and its Salient Features	8
1.6	Impact of Testability Measures on Software Development Process Model	8
1.7	Software Control Interface	10
1.8	Testability in Testing & Corrective Maintenance Phases	10
1.8.1	Testability in Testing Phase	11
1.8.2	Testability in Corrective Maintenance Phase	12
1.9	Testability Measures & Debuggers	12
1.10	Probe Coverage	14
1.11	Test Plans and Controllability Measures	15
1.12	Initial Research on Automatic Test Data Generation	15
1.13	Testability Measures in a Large Distributed Project	16
1.14	Contents of other chapters	17
1.15	Extended Summary of Research Contributions	19
1.15.1	Software Testability	19
1.15.2	Observability Measures	20
1.15.3	Controllability Measures	21
1.15.4	Control Interface	22
1.15.5	Corrective Maintenance	23
1.15.6	Performance Considerations	23
1.15.7	Test Data Generation & Test Driver Development	24

Chapter 2 : Current Testing Methods

2.1	Introduction	25
2.2	Programming Environment	28
2.3	Typical Testing Environment	29
2.3.1	Print Mechanism	29
2.3.2	Debugging Tools	30
2.3.3	Code Coverage	31
2.3.4	Test Execution	32
2.4	Summary	33

Chapter 3 : Observability

3.1	Introduction	35
3.2	Probe Mechanism: A Mechanism for Event Recording	39
3.2.1	Probe Identifier Structure	40
3.2.1.1	Probe Name	41
3.2.1.2	Probe Levels	41
3.2.1.3	Probe Category	43
3.2.2	Probe addressing for control commands	44
3.2.3	Event Recording	44
3.2.4	Probes and Software Execution	45
3.2.5	Test-Control Commands	46
3.2.5.1	Probe Activation/Deactivation	47
3.2.5.2	Probe Breaks	47
3.2.5.3	Event Display	48
3.2.6	Query of Event History file	48
3.3	Discipline to be followed for Design and Coding	49
3.4	Observability Measures in Testing and Maintenance of Software	50
3.4.1	Testing of Software	51
3.4.2	Maintenance of Software	52
3.5	An Example from TIFACLINE HOST Project	53
3.5.1	Brief out-line of the Project	53
3.5.2	Error example 1 : Unable to Connect	55
3.5.3	Error Example 2 : Lost Messages	56

3.6	Event Message Keywords	57
3.7	Probe Coverage	59
3.8	Observability in Software Systems	60
3.8.1	Observability in Scientific Software	60
3.8.2	Observability in Non-Scientific Software	61
3.8.3	Observability in Hardware Systems	62
3.9	Debugger and Probe Mechanism : A Comparison	63
3.9.1	Debugger : The Salient Features	63
3.9.2	Probe Mechanism : The Salient Features	66
3.10	Summary	70

Chapter 4 : Design verification

4.1	Software Design : Need for Verification and Analysis for Improvisation	72
4.2	Design Verification Using Probe Mechanism	73
4.2.1	Limitations of Existing Approaches	75
4.2.2	The Event History Approach using Probe Mechanism	75
4.2.3	Design Verification & Event History Approach	76
4.3	Design Verification	77
4.3.1	Information Required for Design Verification	78
4.3.2	Probe Mechanism: A Mechanism for Event Recording	78
4.4	The Enhanced Probe Mechanism	79
4.4.1	Probe Level & Probe Category	79
4.4.2	Examples of Design /Algorithm Probe Usage	80
4.4.3	Probe Mechanism & Design Verification	81
4.5	An Example from TIFACLINE HOST Project	81
4.5.1	User Specifications of Help Module	83
4.5.2	Design Level Specifications	83
4.5.3	Verification of TIFACLINE Host Software	85
4.5.3.1	Specifications Verification	85
4.5.3.2	Design Verification	85
4.6	Summary	87

Chapter 5 : Controllability

5.1	Introduction	88
5.2	Controllability Development	91
5.2.1	Controllability Measures	94
5.2.2	Modifications Proposed in Software Development Paradigm	98
5.3	Controllability and Testing	99
5.4	Controllability and Fault Diagnosis	101
5.5	Examples of Controllability	101
5.5.1	Table Handling	102
5.5.2	Filing System of an Operating System	102
5.5.3	Concurrent Systems : B-Tree	103
5.5.4	Messages based Distributed System	105
5.5.5	Communication Module of TIFACLINE HOST Project : A Detailed Example	106
5.5.5.1	Controllability in Communication Module	107
5.5.6	Conclusions from Examples	108
5.6	Summary	108

Chapter 6 : Testability & Corrective Maintenance

6.1	Software Maintenance	111
6.2	Corrective Maintenance	113
6.2.1	Fault Diagnosis	114
6.2.2	Improving Fault Diagnosibility	115
6.3	Observability for Fault Diagnosis	115
6.3.1	Probe Mechanism & Stepwise Focusing for Fault Diagnosis	116
6.3.2	Fault Diagnosis : Responsibilities of Software Designer and Developers	118
6.3.3	Approach for Fault Diagnosis	119
6.4	Controllability for Fault Diagnosis	120
6.4.1	Controllability Measures and Fault Diagnosis	121

6.5	Corrective Maintenance & Observability and Controllability Measures	122
6.6	Summary	123

Chapter 7 : Implementation Issues in Testability

7.1	Testability Overheads & Optimization Considerations	125
7.1.1	Attachable & Detachable Observability & Controllability	125
7.1.2	Execution time Activation Control	126
7.1.3	Active Probe Identification	126
7.2	Logging of Probe Messages	128
7.3	Display of Probe Messages	129
7.3.1	On-line Behaviour Window & Filter	129
7.3.2	Off-line Behaviour Window & Filter	130
7.4	Query Processing	131
7.5	Performance Measurement using Probe Mechanism	132
7.6	Assertion checking	133
7.6.1	Probe Mechanism & Assertion Checking	134
7.7	Application of Probe Mechanism in Real-Time Software	135

Chapter 8 : Testability Revisited.

8.1	Testability & Software Development Process	137
8.1.1	Testability Building Phases	137
8.1.2	Testability Application Phases	139
8.2	Testability & Current Testing Methods	141
8.3	Summary of Research Contributions	144
8.3.1	Software Testability	144
8.3.2	Observability Measures	145
8.3.3	Controllability Measures	146
8.3.4	Control Interface	147
8.3.5	Corrective Maintenance	148
8.3.6	Performance Considerations	148
8.3.7	Test Data Generation & Test Driver Development	149

Chapter 9 : Future Directions

9.1	Test Space for Efficient Implementation of Probe Mechanism	150
9.2	Software Dashboard For Large Software Systems	151
9.3	Compiler Support for Observability and Controllability	153

Appendix A : Test Data Generation using Equivalence Classes

A.1	Test Data Generation Using Equivalence Classes	155
A.2	HUTEST : An Automatic testing tool under HUMIS (4GL) Application Development Framework	156
A.3	Testing of Software using HUTEST	165
A.4	Further Extensions	166

Appendix B : Test Driver Development

B.1	Test Driver Development	167
B.2	Test Driver Development for 3GL Based Interactive Applications	168
B.3	Application Development Discipline Required to Facilitate Test Driver Development	169
B.4	Advantages of Design Discipline for Test Driver Development	173
B.5	Building Observability in Application Logic	174

List of References

175

List of Publication

184

Chapter 1 : Introducing Testability Considerations in Software Development Phases

1.1 Introduction

Software Testing is an important activity in software development. Better software development methodologies can only reduce the incidence of errors, but not eliminate the need for testing [Abbo86, Beiz90, Ince 85, Mosl93, Myer79, Whit87].

In specification-driven software development paradigms [Boeh81, Ieee87], the sequential phases of software development are: requirement analysis and specification, software design, detailed design, coding, testing, installation (also called implementation), and maintenance.

In testing phase, the software is subjected to different levels of testing to detect different types of faults. The testing levels are unit testing, integration testing, system testing, and acceptance testing. Test cases are the central ingredient for successful testing of any type. While the goal of unit testing is to test the internal logic of each module, that of integration, system, and acceptance testings, is to test software design, software specification, and software functioning in real environment, respectively.

For a software development project, its test-plans specify the schedule of testing, test units and associated test cases, for the entire testing process. Usually, the test planning is initiated after specification is frozen, and it runs in parallel to activities of software design, detailed design, and coding phases. Specification, design, and detailed design documents are inputs for formulating various components of test-plans, but no component of test-plans is ever used either in the design phase, or in detailed design phase or in the coding phase. Consequently, no provision exists in the developed software to ease the execution of the test-

plans.

To devise adequate test cases, based only on module (user) interface, is extremely difficult, since the software cannot be brought to desired state (for testing purpose) through module interface commands easily. Ad-hoc measures are, therefore, adopted to execute various test cases. This lack of systematic testing, reduces confidence in the reliability of the software.

We define a software as testable software if it can be tested easily, systematically, and externally at interface level (i.e., without any code modification). A software with greater testability can only be made a reliable software. We stress that to produce reliable software, testability must be an essential criterion for architectural design, detailed design and coding phases.

When a large software is installed and released for use, its corrective maintenance is the most essential post-release activity, that the developing agency has to guarantee. Again, the maintenance engineers of a software are invariably different from those who were involved in pre-release development activities of the software.

We define a software as maintainable, if a maintenance engineer can confirm, isolate, and concretize the fault easily, systematically, and externally at user (maintenance) interface level. A software with better maintainability will take less time to rectify, and will, therefore, reduce software maintenance cost. As per the above definition of maintainability, a testable software only can become maintainable .

In this thesis, we try to identify software testability criteria and stress that, to produce a reliable and maintainable software, the software testability criteria has to be taken into account in design and coding phases itself, so that the developed software is readily testable.

1.2 Testability & Maintainability of Software

Existing software development models do not have any provision for building testability in the software in earlier phases. It only asserts that appropriate parts of the test-plans should be prepared, immediately after the specification and design phases, and in parallel with coding phase, so that these test-plans are available at time of testing [Pres92, Ie87VV]. Test-plans are neither made available to the designer nor to programmers to make adequate provisions in the software, to ease execution of test-plans.

Once a software has been built without testability considerations, these cannot be injected afterwards, when their need is really felt during testing and maintenance phases. For example, it has been realized that quality cannot be injected into a product during the last phase of quality control. Quality is a result of activities in earlier phases [Hetz88, Ie87VV, Neuf93, Ould86]. Similarly, testability desired during the testing and maintenance phases cannot be injected in a software after its coding is over. It has to be planned and built right from the beginning.

Most of the research in testing has focused attention on test data selection [Good75, Rama76, Rapp85], static analysis of code [Howd81a, Ince85, Whit87], and dynamic testing techniques [Howd81b, Ince85, Whit87]. Not much attention has been paid to the study of test execution process itself.

To identify the testability criteria of a software being developed, i.e., the basic requirements for facilitating the execution of various types of tests as well as error diagnosis, the activities of software development that need to be analyzed, in detail, are

1. Unit and integration testing
2. Design verification
3. System testing and
4. Corrective maintenance

While analyzing these activities, mechanisms that could meet the above requirements were also explored.

1.3 Testability : Controllability & Observability

In order to understand the need for building testability, i.e., aspects which facilitate testing, it is important to look, first, at the testing activity itself.

For testing, we have to first decide, the granularity of the software to be tested. The software granule can be a unit, a sub-module, a module, or the whole integrated software system. Then, set of tests to be applied on each software granule, are decided. Again, for application of each test case, a software granule is initialized to a desired specific state, in isolation, for execution of the test to be meaningful. When a test is executed on a software granule, observation of its external and internal behavior is necessary to ascertain the correctness of processing and the diagnosis of errors discovered during this process.

Out of all the activities of testing described above, we focus our attention only on the two activities that are related to a software granule under testing. These two activities are : (a) to set the initial state of the software granule as required by the test cases prior to its execution, and (b) to observe the internal and external behavior of the software granule during the execution of test cases. Usually, a software is designed and developed with its functional specification as its sole goal, and testability considerations are never taken into account. As a result, on any software granule, the two above mentioned testing activities cannot be done easily, systematically, or through its interface commands. Conversely, a software is testable, if these two activities are feasible.

In other words, a software is testable if it has two inherent properties, viz., controllability and observability. A software

While analyzing these activities, mechanisms that could meet the above requirements were also explored.

1.3 Testability : Controllability & Observability

In order to understand the need for building testability, i.e., aspects which facilitate testing, it is important to look, first, at the testing activity itself.

For testing, we have to first decide, the granularity of the software to be tested. The software granule can be a unit, a sub-module, a module, or the whole integrated software system. Then, set of tests to be applied on each software granule, are decided. Again, for application of each test case, a software granule is initialized to a desired specific state, in isolation, for execution of the test to be meaningful. When a test is executed on a software granule, observation of its external and internal behavior is necessary to ascertain the correctness of processing and the diagnosis of errors discovered during this process.

Out of all the activities of testing described above, we focus our attention only on the two activities that are related to a software granule under testing. These two activities are : (a) to set the initial state of the software granule as required by the test cases prior to its execution, and (b) to observe the internal and external behavior of the software granule during the execution of test cases. Usually, a software is designed and developed with its functional specification as its sole goal, and testability considerations are never taken into account. As a result, on any software granule, the two above mentioned testing activities cannot be done easily, systematically, or through its interface commands. Conversely, a software is testable, if these two activities are feasible.

In other words, a software is testable if it has two inherent properties, viz., controllability and observability. A software

granule is controllable, if, through granule interface commands, it can be initialized to different states as required by the test being applied. Again, a software granule is observable, if its external and internal behavior can be observed on-line and /or recorded for post-analysis.

As described earlier, a software, in general, has neither the controllability property nor the observability property. Extra measures have to be built in the software during the design and coding phases, to incorporate controllability and observability properties.

Testability measures have been defined as composed of controllability and observability measures. Controllability measures are extra provisions in the software which help in the creation of difficult states required for executing various tests. Observability measures are provisions in the software, which enable the tester to observe all the desired external and internal behavior of the software, to the required degree of detail, with the purpose of ascertaining correctness of processing as well as diagnosis of discovered or reported faults.

Some of the proposed concepts have been implemented and used effectively in TIFACLINE Host project [Tifa90a, Tifa90b], a large distributed database system (over 150,000 lines of 'C' code). Several examples, which are included in the thesis to illustrate some of the concepts, are based on the various modules of this project.

1.4 Controllability Measures

Generally, test conditions can be created using various commands available at the granule interface level. All the states cannot be created using this external interface. Therefore, some additional functionality and interface commands are required to be built so that difficult states can be created easily. This enhancement of

software functionality and its external interface, for purpose of controlling the initial state of a software granule and its execution in the desired manner, is termed as controllability.

These measures can be built in the software provided the specification-based test-plans are made available to the designer. Also, an additional design criteria that the designed software should facilitate creation of difficult states, is to be provided to the design phase. It will ensure that the resulting design of the software is more controllable.

Furthermore, many decisions are made during the design phase, and these design decisions may not get tested, if the tests based only on specification are applied. Accordingly, additional design-based tests are required to be planned. These design-based tests can be worked out at end of the design activity. However, conducting these tests may be difficult, if no provisions are made in the design. Therefore, before the design phase is closed, the design should be modified or extended to facilitate execution of these design-based tests as well. In fact, the design and design-based test planning activities, will have to be iterated till both of these converge.

At end of the design phase, the extra provisions made for specification as well as design-based tests are frozen. These extra provisions are the controllability measures that will have to be programmed in the coding phase to make the resulting software controllable.

1.5 Observability Measures

When a test is executed on a software granule, the output can be examined for correctness. Any abnormal behavior at external level will have to be diagnosed in terms of various execution steps. It will require access to the internal behavior of the software. Generally, print statements are introduced, temporarily, to get access to this internal information. For smaller software granules,

debuggers are also used. Debuggers, however, require more effort on part of the programmer in putting break points and recording various intermediate results manually [Tsub86].

Correctness of output alone is not sufficient, to assert that software executed without any error. Very often, a test may affect the internal state of the software, which is not reflected in the output behavior of the test. If, this internal state is not updated correctly, it may affect the external behavior of the subsequent tests. Observation of change in internal state, is thus very essential, to assure that the test has been executed successfully.

If such an error is not detected immediately, it will be much harder to diagnose later. An error is easier to diagnose, if, it is possible to detect it at the point of its occurrence. More is the gap between error occurrence and error detection, more difficult it becomes to diagnose it due to interference of subsequent processing and additional errors. Therefore, to ascertain the successful execution of a test, access to internal information, including the change of state if any, is very essential.

Observability can be built for different purposes. It can be for purpose of assisting the programmers in unit and integration testing, or providing essential information to the designer to help in design verification [Gupt92]. It can also be built to assist maintenance engineer in diagnosis of fault, monitoring system operations, performance, usage etc. [Gupt91]. Probes for all these purposes, can be identified during the design and the coding phases, and embedded by programmers in the coding phase.

In this thesis, we have introduced the concept of probe as an observability measure, that can be embedded permanently in the software at appropriate places to make it observable. A probe, on execution, generates a message carrying the internal information along with its identity. Probes can be controlled externally, and turned on or off dynamically, providing the desired level of observability of a software granule.

1.5.1 Probe Mechanism and its Salient Features

Probes are very effective tools for software testing and maintenance, and their distinguishing features are the following:-

- Probes are embedded permanently in the code and each probe is uniquely identified by its probe_id. Probes don't interfere with the logical functioning of the software, though it enhances the processing load of the software. Any amount of observability can be implanted in a software.
- Embedded probes can be activated / deactivated using a generic probe activation mechanism. Only activated probes generate and record messages in an event history file.
- Information logged by activated probes in event history file can be queried later for extracting meaningful information for analysis.
- Probes can be associated with classification codes as part of its message which can be very helpful in query formulation for extracting the required information.
- Observability implanted can be for variety of purposes like design verification, unit and integration testing, and fault diagnosis during maintenance. Probes for such purposes can be distinguished by attaching a category code with the probe_id.

1.6 Impact of Testability Measures on Software Development Process Model

As mentioned earlier, testability cannot be built during the testing phase when it is required for performing various tests. It is required to be planned and built in earlier design and coding phases. For this purpose, test-plans, prepared based on the specification document, which are made available only during the

testing phase, should be made available to the design phase as well. Only then, some provision can be made in the design of the system, so that tests which are difficult to conduct, otherwise, can be performed easily during the testing phase. Even the tests, based on the design of the system, should be examined for ease of testing and, if necessary, the design should be suitably modified. This process may be iterated till the design and design-based-tests converge.

Similarly, the observability required for design verification can be identified, once design has been frozen, so that it can be built in the code by the programmer. During coding phase, observability required at code level can be identified by the programmers themselves, and built in the code to help them in unit testing. Observability required for fault diagnosis in corrective maintenance, can be identified during the design and coding phases, and corresponding probes must be embedded in the software by the programmers.

Thus, testability of software can be planned and built in, during the design and coding phases, so that it is available during the testing as well as the maintenance phase. This requires that the software development process be modified so that test-plans are finalized early and made available to design phase.

In hardware systems, all the desired controllability as well as observability measures are planned during design phase and built into the system, as afterwards no changes can be made, even if required or desirable. In software, controllability and observability measures are generally not pre-planned, as software is considered infinitely malleable which can be changed whenever desired. Adhoc measures therefore, have to be used, as and when the need arises for these measures.

1.7 Software Control Interface

The extra controllability measures built in the software to aid in creation of difficult states have to be accessible to the tester. These extra control commands can be made accessible through a separate interface, which can be termed as 'software control interface'. Similar control panels are standard features for many physical systems. For example, tape drives have a control panel, for use by the hardware engineers, to conduct local tape read /write tests etc.

Commands to control observability measures built in the software can also be made available through this control interface. Various commands like enabling and disabling various probes and breaks (discussed in Chapter 3, Section 3.1.5) could be exercised through this control interface. Information captured by various activated probes and recorded in event history file, can also be queried through this control interface.

Thus the various testability measures built into the software should be integrated properly and made available through a separate 'software control interface'. It is highly desirable, as these measures are not only just for the testing phase, but also have relevance during the post-release maintenance phase.

1.8 Testability in Testing & Corrective Maintenance Phases

Testability built into the software is useful during both the testing and the maintenance phases. Testability is relevant in the maintenance phase, as it involves diagnosis as well as testing activities.

1.8.1 Testability in Testing Phase

Testing is performed initially at unit level. Tested units are then combined into modules, and then testing is done at module level. Tested modules are then combined and integration testing is performed for the complete software. For all these testing activities, testability measures can be of great help. Controllability measures can help in creating all the desired test conditions easily, which, otherwise, would have been difficult for many test cases. Observability measures can help in ascertaining the correctness of software behavior as well as diagnosis of errors discovered during the testing process.

Design verification can be done by applying various design verification tests and observing the internal behaviour of the software through appropriate design category probes. Design category probes are identified by the designer during the design phase and embedded in code by the programmers. As the designer need not understand the code written by the programmers, design verification can be done by the designer himself, independent of the programmer. Timing information, available through observability measures can be used by the designer to analyze the performance of various components of the software. Based on this analysis, design can be further refined or redone to meet the performance constraints. A more detailed discussion on design verification is given in Chapter 4.

During system testing, controllability measures can be used for creating various tests conditions and observability measures can be used for diagnosis of discovered errors. System performance can also be analyzed in terms of the performance of various components and necessary steps taken to improve the performance, if required.

1.8.2 Testability in Corrective Maintenance Phase

After release of the software for use, when an error is reported, the first maintenance step is to identify the environment under which the error has occurred. Observability built in the system may help in isolating the software module and identifying its internal state in which the error has occurred. If the system is in operation, very limited experimentation can be done.

The identified internal state is then recreated approximately in the development copy of the software module on a separate machine using the built-in controllability, which is available through the control interface. The approximately identified error environment may have to be iterated repeatedly using controllability provisions till the reported error is recreated. Thus controllability as well as observability is very useful during corrective maintenance phase for fault recreation in the development environment so that it can then be diagnosed and verified.

During maintenance phase, the built-in-controllability and observability measures can also be useful in several other ways. These measures can help, not only in error diagnosis, but also in monitoring the system functioning, observing performance for tuning of various control parameters, and logging the usage of the system. Information generated by various observability measures can be also be used to develop some useful enhancements of the software, quite independent of the main system.

1.9 Testability Measures & Debuggers

Debuggers are the most commonly used tools for software testing. A debugger is specific to a programming language as well as to a programming environment. For testing, a software has to be put under the control of an appropriate debugger, which in turn, gives full control to the tester to execute the software.

Tester can execute a specific component of code; set initial state of the software component by initializing various variables with desired values; stop execution at any point to view the internal state or modify it for next execution; take up next component for execution; or can re-execute the component with modified initial state, etc. Many attempts have been made to further enhance the functionality of debuggers [Agra91, Olss91, Shim91a, Tsub86].

Testing of a software under a debugger presumes that testing is done by one person, who knows the internal details of the software completely, i.e., the author himself. Again, the testing process is completely interactive, and any fault when detected, is also traced interactively. There is no concept of recording of internal state of software for post-analysis. Again, testing under debugger can be done only when the software is off-line, i.e., not in operation, simultaneously.

Since a debugger provides an interactive testing mechanism to the author of the software, it is useful only for unit level testing.

For any large software testing, it is very difficult to decide what to observe on-line. Since a debugger does not provide any mechanism to pre-specify observability, it is not of much use for testing of large software.

Module testing, integration testing, and system testing are done by a team of testers, who need not be authors themselves. Debugger is, therefore, not at all suitable for such type of testing.

Again, since a debugger does not provide any mechanism for recording or to test software in operation, it cannot be used in maintenance phase at all.

Debugger cannot handle distributed systems involving multiple processes (tasks), as the act of breaking the execution for observing the internal state of any one process, disturbs the dynamic behavior of the distributed software itself.

To summarize, though apparently, a debugger appears to have the ability to provide observability, it is not very efficient to use. Ability to change contents of variables is not sufficient for creating difficult states of large software required for execution of certain tests.

1.10 Probe Coverage

During system testing, it is important to measure the extent to which the functionality built in the software has been exercised. Complete functionality coverage may not necessarily result in complete code coverage. For example, when a library of functions are used in a software, completeness of testing of the developed software is in no way related to the code coverage of the library functions. Similarly, system testing of software built using functions provided by various modules, may not result in complete code coverage of various modules. Code coverage is useful only during unit testing, as the tester, who is also the author of the program, knows the program logic very thoroughly and has complete control over its execution and, therefore, can exercise the program exhaustively.

As against code coverage [Beiz90, Mill84], the concept of probe-coverage has been introduced to measure the coverage of functionalities (as defined in the specification and the design documents) during the system testing. Probes having coverage category are only used for probe coverage. To cover one functionality, there can be one or more coverage category probes. Probe coverage can be measured to know the extent of the functionality covered. Uncovered functionality can be identified and additional tests can be planned for its execution.

Complete probe coverage is essential for completion of testing.

1.11 Test-plans and Controllability Measures

Usually for system testing, the software is exercised through the user interface. Test data generation for exercising the various parts of the software is entirely dependent on the capability of the user interface. A software, developed following our approach, has a control interface, in addition to the user interface. Test-plans generation, in this new context will now have to be based on both the user interface as well as the control interface.

Test-plans, which specify the various tests required to be conducted at time of system testing, does not specify how the required test conditions will be created. The design phase which takes the test-plans as one of the inputs and makes provisions for the required observability and the controllability measures in software, can augment the test-plans document by specifying how the various test conditions will be created in terms of the various user interface and control interface commands. It can also specify the observability measures, in terms of various probes, required to be activated to observe the related internal behaviour.

1.12 Initial Research on Automatic Test Data Generation

Initial focus of our research in software testing, was on automatic generation of test data based on the input data characteristics [Raja89, Sinh89]. Our focus was restricted to software developed in a 3GL using application development tools. This effort led to the design and development of an equivalence class definition language. These data definitions were used to generate data for driving the software. Furthermore, the design discipline to be followed to ease development of test driver was also evolved. Using such automatic test data generation methods, software could be exercised more thoroughly.

This kind of automatic test data generation facility is useful, only in exercising the software with more data, for comprehensive testing in pre-release state.

In the initial testing phase, when the software has many faults, the testing activity has to be assisted such that it is more productive i.e. tests can be applied and necessary observation made for analysis. Fault detected should be possible to diagnose easily. It was observed that very significant effort was being spent in this initial testing phase, as no effective tool or methodology was available. The focus of research, consequently, got shifted from automatic test data generation, to what can be done at a more fundamental level, i.e., to facilitate the testing activity itself. The need for building testability measures in terms of observability and controllability measures was subsequently realized. The initial part of the research on automatic test data generation, is also reported in more detail in Appendix A and Appendix B.

1.13 Testability Measures in a Large Distributed Project

The concepts in this thesis have been guided by over a decade of experience by the author in development of large complex systems. The problems faced in testing and diagnosis of such systems have been the main driving force for this research. One such large project, was 'Fault Control System', a distributed fault tolerant software for an Automatic Electronic Exchange (over 3,50,000 lines of 'C' code), where the author managed a major component of the software. The problems faced during the testing of software, and later diagnosis of faults encountered during the implementation phase, provided an opportunity to look for mechanisms, which could help in these activities. Some of the proposed concepts, were evolved and used effectively during the development of another large distributed database system, 'TIFACLINE Host Software' [Tifa90a, Tifa90b], which was undertaken for development subsequently (over 1,50,000 lines of code). In this project, the software development was done in 'C' using a RDBMS package. Some of

the underlying communication layers and distributed data handling, were designed and developed as part of this project.

Observability measures through probe mechanism were used in resolving some of the crucial communication problems. Timing information associated with probe messages proved useful in improving the performance of the communication module to the desired level. The need for controllability was clearly perceived in testing and certain fault handling aspects. As no provision had been made in the software, adhoc measures were used. The problem, however provided an opportunity to think and check possible solutions for relevance and effectiveness. A post analysis led to the concept of controllability measures, which if built could have assisted in more thorough testing.

Contribution of a large complex project in perceiving problems and evaluating the usefulness of possible solutions is no doubt very significant. Many of the concepts discussed in this thesis have proved their usefulness in actual use. A brief description of the TIFACLINE project is given in Chapter 3. Several examples included in this thesis, to illustrate the various concepts, have been taken from the various modules of this project.

As the concepts presented in this thesis have great practical significance, these can be appreciated more easily by those who have undergone through the experience of developing, testing and maintenance of large complex software systems.

1.14 Contents of other chapters

The following describes the structure of remaining part of this thesis in terms of the contents of various chapters :-

Chapter 2 discusses the current testing methods and some of their limitations relevant from the view point of this thesis.

Chapter 3 elaborates on the need for observability and proposes externally controllable probe mechanism for building the desired observability. The mechanism is illustrated, with examples, from a large complex project. Limitations of debuggers, compared to probe mechanism, are also presented.

Chapter 4 discusses, in detail, the use of testability in design verification by the designer. An example from a complex project is used to illustrate the concepts.

Chapter 5 discusses the controllability aspect of testability and its role in testing and fault diagnosis. The concept is elaborated with a few examples.

Chapter 6 elaborates on the role of testability during the maintenance phase. Fault diagnosis activity of corrective maintenance has been analyzed, and how it can be simplified using the built-in testability measures, has been discussed.

Chapter 7 discusses the various implementation related issues, like optimization considerations, probe message logging, and querying of event history file. Use of probe observability mechanism in performance measurement, assertion checking, and real time systems is also discussed.

Chapter 8 discusses testability in various phases of software development process. How testability is built earlier in design and coding phases and used later in testing and maintenance is elaborated. The current testing methods are reviewed in light of the proposed testability measures. The research contributions of this thesis are also summarized.

Chapter 9 proposes some new areas in software testability for more detailed investigation. It proposes how testability can be supported by language compilers; and test space concept can help in more efficient implementation of probe observability mechanism. Use of observability measures in constructing software display panels

is also proposed for further investigation.

Appendix A describes some of the initial research done in the area of test data generation using equivalence class representation of data characteristics. Use of these, in 4GL tools, for automatic testing of application is also described.

Appendix B discusses 3GL based application development discipline to facilitate development of test drivers using equivalence class data characterization.

1.15 Extended Summary of Research Contributions

The research contributions of this thesis can be described, briefly, as follows :-

1.15.1 Software Testability

- This research investigates the types of activities that are required to be done during software development process to produce testable software. A software is testable, if it has testability property, i.e., it can be subjected to pre-defined test-plans easily, systematically, and without following any ad-hoc measures. This research has identified distinct activities that need to be done during the design and coding phases to produce testable software.
- Testability property of a software is defined as a composition of controllability and observability properties.
- Usually, any software produced have extremely limited controllability and observability properties. Extra provisions, viz., controllability measures and observability measures, have to be ingrained in the software, for it to acquire controllability and observability properties respectively.

- Software with pre-built controllability and observability measures would not require any adhoc measures to be developed during the testing phase. All the planned tests can be executed readily, which will result in more reliable software.

1.15.2 Observability Measures

- Observability measures have been defined as provisions in the software to facilitate observation of internal state of the software at appropriate level of abstraction and at appropriate point in execution.
- The probe mechanism has been proposed as an observability measures for a software. A probe is a permanently embedded piece of code, that on being executed, records internal information of the software in an event history file, without affecting the logic of the system. Probes can be activated/deactivated through external commands, and only activated probes record information for post-analysis of execution.
- Probes can be of various categories, e.g., design verification probes, unit testing probes, corrective maintenance probes etc. Furthermore, probes for any category can be defined for capturing information at multiple levels of abstractions. Based on specification, design, and detailed design of the software, the probes (along with its category and level) have to be identified by the end of design and detailed design phases, and have to be embedded by programmers while coding. Various refinements have also been proposed to reduce execution time overheads.

- Design verification can be performed more easily using design category probes identified by the designer for implantation by the programmer. Design verification can be done by the designer independent of the programmers, as there is no need to understand the implemented code.
- For completeness of system testing, the concept of probe-coverage has been introduced. Code coverage is more meaningful for unit testing and not for system testing. During system testing of large software, 100% code coverage is extremely hard to achieve. For large software, functional coverage is more relevant during system testing, which can be measured using the proposed probe-coverage concept more effectively. We define system testing to be complete when 100% probe-coverage has been achieved.

1.15.3 Controllability Measures

- Controllability measures have been defined as provisions in the code to bring the software to specific states which are difficult to achieve through user interface commands for the execution of state specific tests. Special conditions, error, or boundary conditions, are some of the typical examples of states, which are difficult to create.
- Building controllability require modification of the software development process. The types of controllability measures and their provisions have to be decided by the end of design phase. The specification-based test-plans are necessary input for the designer. Since design-based test-plans for design verification can be finalized, only after the completion of design phase, in our model, the design phase has to be iterated till the inclusion of all controllability measures for design-based test-plans is properly synchronized with the design itself.

- The controllability measures for specification verification tests, design verification tests, and code verification tests are incorporated in the software during the coding phase.
- Controllability measures are required to be incorporated in the software to
 - (a) Set Test Environment
 - (b) Set Test State
 - (c) Activate Observability Measures
 - (d) Provide Test Input
 - (e) Perform Test Analysis
- To test a software component in isolation, it is required to set test environment. A software component can be sub-module, module or a sub-system. For setting the state of a component and for providing test input, some additional software (even module) may have to be built which can be activated using the control interface commands.

1.15.4 Control Interface

- In addition to the user interface, the software is augmented with a control interface, that provides various commands for activating controllability measures, in such a way that the various tests are conducted easily, systematically, and without following any ad-hoc measures. The observability measures built in the software are also made available through this control interface along with necessary documentation.
- The control interface is accessible to the designer and programmers for testing purposes, and after release of software, it is made available to system administrator for monitoring purposes and the software maintenance engineer for fault diagnosis.

1.15.5 Corrective Maintenance

- A software is maintainable, if a maintenance engineer can confirm, isolate, and concretize the fault easily, systematically, and externally at user (maintenance) interface level.
- Steps involved in corrective maintenance have been analyzed and defined. Probes can also be inserted to build macro and micro level observability required for software fault diagnosis. These can also be integrated and made accessible through the proposed control interface. 'Software maintenance guide', describing all built-in maintainability measures, can be made available to the maintenance engineer.
- Controllability and observability measures improve fault diagnosability, which reduces the effort required for fault diagnosis and rectification considerably. Thus, cost of ever increasing corrective maintenance, can be reduced significantly.

1.15.6 Performance Considerations

- Some controllability and associated observability measures can be designed as a detachable/attachable components. They can be selectively detached and attached whenever required, so that their overheads can be reduced during the operational use of the software. The attached measures become effective only when activated through control commands.

1.15.7 Test Data Generation & Test Driver Development

- An equivalence class definition based test data generation language (HUTEST), has been developed for fourth generation form-based application development environment (HUMIS). It has been shown to be useful for more extensive and automatic testing of pre-release version of an application software (Appendix A of thesis).
- For 3G programming environment, an application software structure has been proposed which can help in easy development of an application test driver (Appendix B of thesis).

END OF CHAPTER 1

2.1 Introduction

Software Testing is one of the important activities in software development. It is practically not possible to remove all the errors in the software by any amount of testing based on various techniques [Abbo86, Beiz90, Mosl93, Myer79]. Exhaustive or complete testing would require testing for all possible paths, which can be very large even for a simple program. Formal methods of program proving or construction have not been successful beyond very simple programs. Even these are prone to errors as these are not completely automated. Most of the reported research works in software testing have been on test data selection [Good75, Rama76, Rapp85], static analysis of code [Howd81a, Ince85, Whit87], and dynamic testing techniques based on functional and structural aspects of software [Howd81b, Ince85, Whit87]. Comparative effectiveness of these techniques has also been studied [Basi87, Haml89, Ntaf88]. Again, these research have focused largely on scientific software. For critical applications, software structures have been proposed to detect malfunctioning of software and take appropriate remedial actions [Parn77]. Even then, software cannot be made completely reliable [Parn90].

Various testing techniques are essentially trying to improve the effectiveness of testing, so that with limited resources available for testing, more reliable software can be produced. To test software, first unit level testing at module level is performed. The tests applied are based on functional specifications and the internal design of the module. These types of tests are also known as black and white box tests respectively. Code coverage and branch coverage measurements techniques can be used to know the extent to which the code has been exercised. The knowledge about the uncovered parts of the code can be used to plan further tests.

After modules are tested independently, they can be integrated one by one and tested. If all modules are integrated in one step, errors may be difficult to diagnose as all the module interfaces are untested [Ince85]. Stubs or drivers may have to be written depending on whether integration has been done top-down or bottom-up [Pres92, Whit87]. The final integrated system is subjected to functional tests which are derived from the system functional specifications. Tests based on non-functional specifications like performance, overload conditions, fault handling etc., are also conducted [Ince85]. Several tools and techniques are available for conducting the above activities [Lutz90, Mill84].

It is being realized increasingly that the cost of error correction can be reduced significantly, if errors are caught earlier in the software life cycle [Hetz88, Neuf93, Ould86]. Many errors detected, during testing phase, could have been prevented, if specification and design phases were given adequate attention. Errors in earlier phases get amplified in later phases of software development. An incorrect specification can lead to incorrect design and coding. If it is detected during testing phase, it may result in change in design as well as coding of the software. To reduce incidence of errors in earlier phases, concept of reviews, walkthroughs and inspections have been introduced [Faga76, Faga86, Ie87RA, Selb87]. Emphasis has been put on validation of the deliverables of each phase of software life cycle, against its own inputs, before proceeding to the next phase. Using these techniques, many errors get caught in earlier phases of software development, thus reducing the overall software development cost.

Standards for various software life cycle phases have also been evolved to improve the quality of output of each phase. It essentially defines the software development process more rigorously. In the IEEE standards [Ieee87, Ie83TD, Ie87VV, Ie89QA], the various activities and sub-activities have been specified clearly. Structure of various documents, which should be produced in various phases, is also specified. It may be quite difficult to implement the guidelines provided in the standards for the various

phases of the software development cycle. However, these can serve as an excellent checklist to start with and can be implemented by a software house in a phased manner. Following the standards of software development can improve the quality of specification document, design document, and code to a considerable extent.

The importance of measuring the complexity of design (modules specification and pseudo code) has also been emphasized, so that corrective action can be taken earlier in the software development life cycle. Cyclomatic complexity has been extended to module design complexity and integration complexity [McCa89]. Structure metrics based on "fan-in" and "fan-out" of procedures have also been shown to be useful for design complexity measurements [Henr90].

Using software inspection, static analysis techniques including complexity measurement, and software standards, the cost of error correction is reduced significantly, as errors are caught earlier in the software life cycle. However, the need to test the software is not reduced. The software still has to be tested thoroughly. The only contribution, which the above process makes, is that the number of errors is reduced significantly, especially those in specification and design of software which require much more effort to fix in later phases.

Various tools [Lutz90, Mill84] and techniques as well as the rigorous process defined by software standards, do not focus attention on improving the testability of the software. It does not propose any provision to be incorporated in the software to ease the testing activity. The basic process of testing and diagnosis has not been analyzed with a view to make it more productive, especially when large amount of resources get spent. Much of the research works focus attention on how to select test data [Good75, Rama76, Rapp85], do static and dynamic testing etc. [Howd81a, Howd81b, Ince85, Whit87]. The test plans finalized for the software are not at all used during the design phase. Neither the designer, nor the programmer (of each module) keep the testability of

software in mind while executing their tasks.

2.2 Programming Environment

Till recently, most of the software development used to take place in third generation programming languages. Over the past few years, powerful fourth generation language (4GL) tools have become available for developing application softwares which constitute a major part of software development.

In a 4GL programming environment, the various components of applications like input forms, reports, queries etc., can be specified to the system. These definitions are used to generate code, or are interpreted directly during execution of the application. In such software, testing gets reduced to only validating the application with respect to the user specification, i.e., one only need to validate that the developed application is as per the specification of the user in terms of functionality, control flow etc. As the applications are simply defined, the generated code or interpretation is error free.

Significant software development still takes place in third generation programming languages, as 4 GL can take care of only certain classes of applications. The third generation programming languages are also being used for considerations of processing efficiency. Large complex software continues to be developed in 3GL, as these provide the most general programming environment. The testing issues related to software discussed in this thesis are related to 3GL programming environment.

Various programming languages differ from each other in terms of error proneness. For example, the programming language 'C' is more error prone compared to Pascal. Pascal is better "block structured" and has parameter type checking built into it. Such differences may result in some differences in programmer's productivity. Choice of languages for software development is made by the development

agency based on various factors, such as, the nature of application, available libraries of functions, control over low level operations, execution efficiency, general industry trends etc. Currently, for developers who are developing software for Unix or other associated operating systems, the programming language 'C', is one of the most popular programming languages, as it gives advantages of both higher level languages as well as give low level control, even though it requires more effort in program testing.

2.3 Typical Testing Environment

The programs developed in 3GLs require considerable effort during the testing phase. Typically, it takes more time to test the software than to write the code. Most of the time is spent in debugging the software, as initially the software is full of bugs and hardly any test case run successfully without encountering several bugs. The only commonly used tools are typical debuggers (like 'sdb' for 'C') which are available as part of the programming language support.

2.3.1 Print Mechanism

Typically, no provision is made in the software to help in the code testing. Once code is written, the programmer applies one test after another. In case, any fault is found, print statements at appropriate places inside the code are introduced, and the program is recompiled and re-executed. The information displayed as a result of print statement indicate to the programmer what really happened at various suspected stages of processing. Once the error has been traced and corrected, some of these print statements are commented, so that displaying of information is restricted to what is needed for further analysis. A debugger is used only when the problem is not resolved using the above simple print mechanism.

Advantage of print statements is that one need not put break points and issue display commands as required in debugging environment. All the required information is described by the embedded print statements, which get printed every time the software is executed. However, print mechanism requires frequent commenting and uncommenting, and recompilation of code, each time changes are made. Moreover, what to comment or uncomment for diagnosing a particular problem requires intimate knowledge of the program. In this style of testing, lot of efforts get spent in diagnosis of faults.

2.3.2 Debugging Tools

A debugger is used by the author of a program since it requires thorough knowledge of the program before it can be used. It is an interactive tool, and hence takes lot of time and effort to diagnose a fault in large applications. The mechanism is, however, general purpose in nature. Breakpoints can be put at the entry of any function or at any line in the source code. Variables can be examined and modified on-line. But on-line source code level modification is generally not possible. Programmer has to come out of the debugging session to do source code modification. The modified code has to be recompiled, and again run with the debugger for further testing. Limitations of debuggers, which are generally available in programming environments, have been realized. Several efforts have been made to enrich its functionality [Aqra91, Shim91b, Tsub86].

If the software development is on a small scale, a debugger is useful, but for large, complex or distributed systems, debuggers are not that useful particularly during integration and system testing.

2.3.3 Code Coverage

Code coverage [Beiz90, Mill84] is an important measurement of software testing activity. It can reveal which part of the software has not been executed at all so that additional tests can be planned to execute those parts of the software. Though execution of every line of code once is not a guarantee for correctness, but it certainly helps in removing many simple bugs which surface by simple test runs. If every line of the code has been executed successfully at least once, only subtle or non-trivial bugs will be left in the software. These bugs can be removed only by more rigorous testing, requiring comparatively more resources. Covering fresh code through additional test runs, provides the best return on testing efforts.

Profilers or code coverage measurements utilities [Unix92] are available which can provide detail information on coverage at function and source code levels. As these are available as part of operating systems like Unix etc., these can be used easily for code coverage measurements. Using such measurements, code which has not been executed can be easily identified.

However, these tools do not provide any semantics of the uncovered / unexecuted code. It is upto the programmer to figure out the higher level concept associated with the uncovered part of the code in order to plan fresh tests.

For large software testing, code coverage is more meaningful at unit level and not at system testing level. To understand this, we will have to first look at the way complex systems are designed and coded. Design of large system takes place in two steps. The first design step consists of overall architectural design of the software which breaks the complex system into manageable modules with well defined interfaces. In the second design step, the detailed design of each module is done, which is followed by their coding.

Each developed module can be thoroughly tested independently, i.e., complete code coverage can be achieved using tests based on the specification of the module and its detailed design. Code coverage, during module's unit testing can be measured using code coverage tools. After each module has been unit-tested separately, these are integrated in a phased manner, either in top-down, or in bottom-up, or in a mixed manner. In this process, integration related faults are discovered and rectified. Finally, the fully integrated software has to undergo system testing. Here, tests based on software specifications and its design architecture are performed. These tests may not lead to complete code coverage, as certain detailed design conditions may not occur readily.

During system testing, what is important to measure automatically if possible, is whether all the functionality both due to specification as well as design architecture has been tested or not. Code coverage being at code level cannot measure this aspect.

2.3.4 Test Execution

The developed software is tested against the test-plans prepared based on the specification as well as the design (architectural as well as detailed design). Software is designed without any information about the specification-based test-plans. Design-based tests are concretized once the design is complete. Some of these tests are very difficult to perform as the specific state required may be not be easy to create using the user interface commands.

Even by reaching inside the system, many design-based tests may be difficult to conduct, as no thought had been given as to how tests would be conducted. Consequently, such tests are not done thoroughly. In more demanding situations, if these difficult tests have to be conducted thoroughly, adhoc measures are used which require a lot of extra efforts. Generally, once the tests have been performed, these adhoc measures are discarded and forgotten.

In case, these difficult tests are not thoroughly performed during the testing phase, and when an error is encountered during the system operations, the software maintenance engineer finds it extremely difficult to rectify such errors. This is because the maintenance engineer may not have been involved in the design and development of the system and, therefore, is not thoroughly familiar with the code. In such a situation, building test environment using adhoc measures, if required, to test the software for diagnosis purposes, is even more difficult for him.

As software is not designed for testability, not all the desirable tests can be performed, resulting in many potentially unreliable parts in the system. Faults in such unreliable parts lead to increase in maintenance cost as the corrective maintenance is more difficult to perform.

2.4 Summary

The main points of this Chapter can be summarized as follows :-

- Testing of developed software is essential as formal methods are still inadequate to certify correctness of large software.
- Software development process and software engineering standards have given stress on reviewing the product of each phase of development. This helps in reducing the number of errors in the software during the development process and hence reduces the cost of error correction.
- Even the software produced using the proposed standard software development processes has to be thoroughly tested before it can be released for use.

- Testability of software i.e., provisions to ease test execution, analysis of test results, and diagnosis of reported fault, has not been paid enough attention.

- Test-plans prepared based on specification and design of software are not taken into account by designers as well as programmers to ensure their ready execution. Consequently, many tests are very difficult to perform.

- In 3GL programming environment, considerable efforts get spent in debugging and testing of software.

- Print statements are often used for testing and debugging purposes. These are somewhat cumbersome to use, as it requires frequent commenting and uncommenting of statements, and recompilation of program.

- Debuggers are useful for testing by the author of the program. It is interactive in nature and not all that useful for large software testing and debugging.

- Code coverage measurement is useful only at the level of unit testing. It is not very useful during the system testing, where functionality coverage is more important.

- Certain tests are difficult to execute, as the state required by them cannot be created easily using the user interface level commands. For this purpose, it becomes necessary to use adhoc measures which require extra efforts. Furthermore, this approach leads to unsystematic testing and adequate testing of software become too difficult to achieve. Again, it poses difficulty in the maintenance of the software.

END OF CHAPTER 2

3.1 Introduction

Observability is one of the two important components of testability. It asserts that during execution of a test, it is important to know not only the output but also the various intermediate results.

If the output is correct for the given input, then the test is assumed to be successful. However, this is true only if the test does not change the internal state of the software, which may affect subsequent behaviour of the software. If the change in internal state is not done correctly, and the output is correct, clearly, even then a fault exists. This incorrect change in internal state can be noticed, only when it is visible to the tester at the time of the execution of the test.

In the absence of information on the change in the internal state, it will not be noticed until it affects the results of subsequent tests. Such errors becomes harder to diagnose in circumstances where the cause of error is far removed from its visible symptom. Therefore, it is essential to get access to the internal processing performed by a test, so that one can be sure that not only the output is correct but it has changed the state of the software correctly as well.

In case the output is incorrect, even then to diagnose the fault, one has to look at the intermediate results. The fault is localized by checking the output of each of the sub-activity underlying the test. For this purpose, sufficient information about the various processing steps should be made available to the tester, who has the responsibility of rectifying the fault observed in the output of a test.

From the above discussion, it is very clear that access to internal information is essential both for fault rectification as well as

ascertaining that a test has executed correctly i.e., observability of software processing at the desired level of detail or abstraction is required for both of these activities.

As discussed in Chapter 2 on current testing methods, the above required information is extracted using either a symbolic debugger or simple print statements. Print statements are introduced in the software to extract the desired essential information. First, a higher level information may be extracted through very few print statements, inserted only at the beginning and end of each block of code, which may localize the fault to a small section of the code. Then, more detailed information may be extracted from that section by inserting more print statements, and repeating the test, which helps in identifying the faulty code.

The same can also be achieved using a symbolic debugger. The extraction of the desired information can be done, as all the variables are available for display at any point in execution of the software. Using breakpoint mechanism of debugger, processing can be stopped at any point either at procedure level, or at source code level. The process of debugging is interactive and repetitive, and requires understanding of the control flow of the entire software [Tsub86].

These mechanisms have worked well for testing of software of smaller size. Invariably, large software is developed by a team of software professionals as a project. It has been observed that in large software development, more effort is getting spent in debugging and testing of software, especially written in languages like 'C', than the effort spent on writing the code itself.

Rectification of fault becomes even harder as testing goes beyond the unit testing, which is testing in the small and is done by the programmer who has written that portion of the code. There are no tools available to help in system testing and fault rectification in corrective maintenance.

Debuggers have become somewhat more powerful in functionality with time, but have not changed in its basic principles. Some of the significant developments are (a) control breaks on predicates defined over state variables [Tsub86], (b) backtracking program-execution [Agra91], (c) facility to observe higher level behaviour defined in terms of primitive events [Bate83,Olss91], (d) guidance to programmers in error localization process based on program structure [Kore88], and (e) automatic identification and display of relevant parts of the code responsible for a class of errors [Shim91b].

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 one moves from unit testing to system testing and then to corrective maintenance. It can only used be for small programs. Debuggers also have some serious limitations in handling distributed systems [McDo89, Cheu90], where tasks are distributed on multiple processor sites and real-time applications; where temporal aspects are involved. Debugger presumes centralized control and hence cannot be utilized to test distributed software. In case of real-time applications, the act of debugging disturbs the processing behaviour itself. Limitations of sequential debuggers has led to the development of a new framework for distributed debugging [Cheu90, Garc84, LeBl87, McDo89, Mill88].

What basically one needs for this task of fault localization and ascertaining correctness of test execution, is observations of internal processing at various levels of abstractions and details. The quest for a more effective mechanism led to the concept of **permanently embedded probes**. Each probe has an identification code and can be activated from outside whenever required to get the desired information. Information captured by **activated probes** is recorded in a log file, which can be queried later, for analysis. These probes have to be planned during the design and coding phases and implanted in the code by programmers.

Using probe mechanism, designer as well as programmer can plan what is required to be observed for various purposes. Thus, later on, one has to only choose what aspect one wishes to observe, and corresponding probes can be activated.

Each activated probe, generates a message containing all the necessary information at that point. Probes can be defined at various levels of abstractions. Thus behaviour of the software can be observed at any level of abstraction, by activating the corresponding levels of probes. Based on observation of software behavior at a higher level, fault can be localized to a specific part of the code, which can then be observed in detail with detailed level probes.

Probe messages with timestamp information can be used to measure the performance of the designed system and identify weak links in the system, for further tuning or redesign. During on-line operational phase, such information can be used to tune various system parameters for better performance.

Probe like mechanisms, referred to as "software monitors" and "software sensors", have been used for software instrumentation. These have been found to be useful in measurement of code coverage [Rama75], during testing of software etc. [Rama75a, Schu87, Gupt92]. Instrumentation has also been shown to be useful in detecting data flow anomalies, some of which (like array handling) are difficult to tackle by static analysis [Huan79]. As probe mechanism does not alter the dynamic behaviour of the software (except cause marginal processing overheads), these are being used in distributed systems to analyze problems related to dynamic behaviour at system level [Cheu90, Grac84. McDo89].

The traditional probe mechanism used for event recording has been enhanced. Concept of probes for various levels of abstraction, like design verification, code testing, maintenance, system monitoring, etc., has been introduced (refer to Section 3.2.1.3 of this Chapter for more details). Probe identifiers can be structured,

reflecting the structure of the software and used to address a group of probes in a generic manner. Information gathered by active probes is recorded in event history file, which can be queried for analysis purposes. The proposed probe mechanism has been implemented and used in a large complex distributed database project developed in 'C'. It was found to be very effective in system testing and fault rectification.

The following section describes the probe mechanism in more detail. Section 3.5 presents some examples from the distributed project mentioned above. Section 3.8 discusses observability present in various types of software and hardware systems. Section 3.9 compares the probe mechanism with debuggers, highlighting the advantages of the former over the later. Section 3.10 summarizes the important points of this Chapter.

3.2 Probe Mechanism: A Mechanism for Event Recording

Probe mechanism is best suited to record various events occurring in the system. Probes can be embedded anywhere in the program by the programmer.

A probe is a function call with the syntax:

```
Probe (Probe_id, Event_message)
```

where Probe_id is a unique structured identifier which identifies the point of origin of the event_message. The event_message contains all the important information relevant at the point where the probe is placed in the code.

As probe is a single call with certain number of parameters, it does not affect the control flow or logic of the software in which it is embedded.

Events can be recorded in one or more event history files, but for simplicity, we presume that all events are recorded in one file. By placing probes at appropriate points in the code, all the information required for ascertaining the correct execution of a test can be gathered. Moreover, using the information generated by various probes, one can reconstruct the global state of the system as well.

Timestamp is automatically prefixed on each message generated by probes. It is very essential for systems having multiple processes (tasks), as it can be used to order the various events for analysis. Also log information can be used to replay the message-generation in proper time order. Timestamp is not required in single process software for fault diagnosis. However, it is essential for performance measurement and tuning.

3.2.1 Probe Identifier Structure

Each probe is identified by a `probe_id` which can have a very simple structure, e.g, it could just be a unique number; or it could be a complex structure, where various parts represent various aspects of the information captured by the probe.

The structure of `probe_id` discussed in this section is what was found to be very useful. However, the user is free to choose any other structure suitable for one's own environment. The structure defined here has been used in the rest of this thesis.

Examples, in this section, are being taken from the Communication module software which is described in detail in Section 3.5. In brief, this module assembles messages from various processes into files for various destination hosts. Similarly, messages received in files from other systems, are disassembled and sent to queues of various local processes.

Let us start with a simple example and gradually build up a more complex and powerful structure around probe_id.

3.2.1.1 Probe Name

At the simplest level, probe_id could be of the form

```
fun.n1.n2.n3....
```

which is very similar to various sections, sub-sections of a report. Each '.' represents a further level of detail. 'fun' represents the abbreviated code for the function or procedure where the probe is residing. n1 represents the higher level of information, n2 the next lower level and so on. This probe name uniquely identifies a probe within a software.

For example

```
sqrt.1  
sqrt.1.2
```

The behaviour represented by such probes are basically at function or code level.

3.2.1.2 Probe Levels

Probes can also capture higher level behaviour of the software. It could be at module or sub-modules levels. Probes which capture information at such higher levels can be associated with "level code" as prefix to the probe number. e.g.

```
L1.L2.L3/fun.n1.n2.n3 or
```

```
Module.Sub_module.File_name/fun.n1.n2.n3
```

Large software is composed of modules and sub-modules. Each sub-module may have several files, each containing logically related functions.

If the probe message information represents software behaviour at module level, probe level code would have only module name present in level code. The other parts L2 and L3 would not be present and would be represented by '_'. For example

Comm._._/load.1.5

represents a Communication module level probe identifier.

If the probe message information represents sub-module level behaviour, then both module and sub-module name could be present and only L3 would be absent. For example

Comm.Assem._/pack.1.2

The number of levels i.e. L1,L2,L3.. would depend on the particular software. Sometime sub-module or file levels may not be present. In that case, shorter level code structure like

L1.L2/fun.n1.n2.n3

or

L1/fun.n1.n2.n3

may be used. Sometimes, if more levels exists in software, then more levels can be created in a similar manner.

3.2.1.3 Probe Category

Probes can also be categorized based on what aspect of the software behaviour is being observed. It could be for purpose of

- (a) Algorithm level behaviour for code verification by programmers.
- (b) Design level behaviour for design verification by the designer.
- (c) Macro and micro level behaviour for rectification of reported faults by the maintenance engineer.
- (d) Monitoring usage of system and resources for purposes of tuning its performance (or even charging the user).

The probes capturing the above categories of software behaviour, can be assigned a code, prefixed to the level_code in the probe_id (separated by '/'). The codes for above categories have been defined as follows :-

Code behaviour	'A'
Design behaviour	'D'
Maintenance	'MAC' for macro behaviour 'MIC' for micro behaviour
Monitoring	'MON'

For example

```
A/Comm.Assem._/pack.1.1
D/Comm.Assem._/pack.1
MAC/Comm.Assem._/pack.5
MIC/Comm.Assem._/pack.5.1
MON/Comm.Assem._/pack.4
```

Probe category code can also have a structure like that of level code and probe name (i.e. `._.` , so that maintenance probes can be coded as M.MAC and M.MIC). But, single part category code has been used for simplicity.

3.2.2 Probe addressing for control commands

A set of probes can be referred to, by using a generic expression. For example

```
D/Comm.Assem._/*
D/Comm**/*
D/**/*
D/Comm._._/*
```

Here `**` matches with any numbers of any character except the code terminator `'/'`. `'?'` would match any single character except `'/'`.

The first example represents all design-category probes in assemble sub-module of communication module, i.e., 2nd level design-category probes in 'Comm' module. The second example represents all design-category probes in communication module at all levels, i.e., first level, second level, and third level, design probes in 'Comm' module. The third example refers to all design-category probes in all the modules. The fourth example represents all first level design-category probes in Communication module.

3.2.3 Event Recording

When a probe is executed, the event message associated with the probe is computed. The probe-id and the associated event message along with a timestamp, is recorded in the event history file as one record.

Event-message is a string, containing values of several variables, each preceded by the name of the variable.

Examples

```
"Telephone_No : ?t Docket_no : ?d Fault : ?state"
```

```
"Docket_no : ?d Fault_class : ?class"
```

In the above examples, t, d, state, and class are program variables, that must be valid at the probe location. For the first example, at run time, the probe may record the following message in the event history file.

```
"Telephone_No : 7254 Docket_no : 231 Fault : Dead "
```

3.2.4 Probes and Software Execution

For different test cases, different probes will be meaningful. To avoid insertion and removal of probes from the source code for different test runs, we propose that the probes reside in the code permanently, as the overheads are minimal.

Any software embedded with probes can run in one of the two modes: test mode & operation mode. When the software runs in "operation mode", no events are recorded. When the software runs in "test mode", events get recorded in the event history file. In "test mode", the software is proposed to be controlled through a test-control position, which could be a workstation. Several commands are available at the test-control position.

In a software having multiple processes, the event messages are sent to a event monitor process, which is responsible for recording the messages in the event history file. If the software is run in "operation mode", this process remains in dormant state, and does not record any message. But, when the software is run in "test mode", this process becomes active. It then receives commands given by the test-control position and behaves accordingly.

3.2.5 Test-Control Commands

Software can be put to a specific test run through test-control position. Again, to have recording of only relevant event messages, only a specific set of probes should be active for a specific test run. Probe Activation and Deactivation commands are available at test-control position. A deactivated probe does not record its event in the event history file. By default, all probes are deactivated.

The sequence of activation and deactivation commands are stored in a probe command table. This table can also be loaded initially from a standard command file and further commands can be added later on-line. On encountering a probe call, the probe command table is used to determine the activation status of its probe_id. Only active probes generate messages, which get recorded in the event history file.

3.2.5.1 Probe Activation/Deactivation

Probes can be referred to in a generic style so that a group of probes can be addressed through a single command at test-control position.

Examples

```
Enable D/*._._/*
```

This will activate all the level-one design category probes in the entire software.

```
Enable */Comm.Rec.__/*
```

This will activate all level-two probes in Receive sub-module of Communication module.

```
Disable */Comm._._/*
```

This will deactivate all level-one probes of Communication module.

3.2.5.2 Probe Breaks

Break points can be put on selected probes from test-control position. On encountering a break point, execution control returns to the test-control position to enable the tester to activate or deactivate other probes, or query event history file.

Example

```
Break on A/Comm.Rec.Deas/dispatch.5.2
```

3.2.5.3 Event Display

For displaying event messages, multiple windows can be defined on the test-control position work-station. Each window can be associated with a filter, i.e., a condition to select messages for display. Thus, different windows can display different types of messages from same or even different processes of the software. This facility helps in understanding and analysing the dynamic behaviour of the software.

3.2.6 Query of Event History file

The message recorded by all the activated probes, in event history file, can be queried using a query language.

Examples

```
Select * where Telephone_no >= 7000 and Fault = "Dead"
```

```
Select * where Probe_id = "**/Comm._./**"
```

The second select command will display all level-one messages generated by the Communication module. (Please note that, '*' in a string matches with any set of characters except '/').

The query on event history file can be done, interactively on-line through test-control position, or later off-line through an Event-Screen package.

3.3 Discipline to be followed for Design and Coding

To effectively use the probe mechanism, to aid in the testing and maintenance of software, the following design discipline is proposed. It will help in testing as well as localizing the faults to module, sub-module and code level.

For building observability in software, one need to plan for it during the earlier phases of design and coding. Large software should be designed first at architectural level. It should break the software into modules, with well defined interfaces and their interactions. Interaction among various modules can be captured using design-category level-one probes. These probes should be defined by the designers of the system. These probes will help the designer in design verification during system testing.

Each module is given to a software team for detailed design. Each module may be further broken into sub-modules. This detailed design should also be documented. The design category level-two probes required to capture the internal design level behaviour of the modules should also be defined.

Definitions of both level-one and level-two probes are to be documented, and given to the coding phase so that these probes can be inserted in the software at appropriate places in the code.

Each sub-module is given to a team of programmers for coding. The programmers can define the level-three probes, which are algorithmic probes. Usually, all the functions related to an algorithm or a specific purpose, are grouped in a file. The file name can be used to represent this third level code. Programmer can also introduce more detailed level probes in various functions. He can utilize the probe name structure to capture these further levels of details at function level.

The probe_messages for various purposes should be constructed in such a way, that it provides all the information necessary which

will help in testing as well as in localization of faults.

Maintenance Category Probes : Probes for maintenance (macro and micro level observability) can also be identified during the design and coding phases. As the purpose of these probes is to localize the reported fault and not design or code verification, these may be different or include a subset of design and code category probes. Macro level probes could be for capturing the occurrence of various events. Micro level probes would have more detailed information about various events. These maintenance probes should also be specified during the design and coding phases, and embedded, so that these are available during the maintenance phase.

System Monitoring Probes : Similarly, the probes for system monitoring, in terms of usage of various resources, should also be identified, embedded, and documented during the design and coding phases. Information captured by such monitoring probes can be used for tuning of the system parameters dynamically. This information can also be used to build add-on modules independently; for instance, accounting module.

All the observability measures in terms of the embedded probes in the software for various purposes, should be properly documented, so that appropriate probes can be activated when required.

3.4 Observability Measures in Testing and Maintenance of Software

The observability measures, built during the design and coding phases using the probe mechanism, are useful during the testing as well as the maintenance phases. These are discussed, in detail, in the following sub-sections.

3.4.1 Testing of Software

Though the test probes are embedded permanently, enabling and disabling of probes are externally controlled. During the execution of a test, the desired probes can be activated, to observe the relevant internal level behaviour of the software. Any incorrect output can be analyzed in terms of the various processing steps. Even for correct output, internal processing, which affect the state of the software, can be observed to make sure that the change of state has been done correctly.

Testing for a specific module/sub-module/procedure should be done by activating level-two and level-three probes. Testing can be done either top-down or bottom-up by activating different probes.

Integration and system testing can be done by activating level-one probes. Design-verification can be done by the designer using the level-one design-category probes. Performance of various components can also be measured, using the timestamp information attached to probe messages, for further tuning of the software.

When the software is released for the first time, the module level probes, i.e., level-one probes (and level-two and level-three probes of sensitive piece of codes) can be left activated all the time, till the software stabilizes.

Thus observability measures built for various levels of the software behaviour, help in observing the internal behaviour to the desired level of details, which facilitates test analysis and diagnosis of faults discovered during testing or post-release period.

3.4.2 Maintenance of Software

A software designed and implemented following the above discipline should produce a maintenance manual, which gives in detail, information about all embedded probes, their levels, identity, and the meaning associated with their probe_messages. The software testing or maintenance team, then, need not use code-visualization tools, or extract design from the code, etc. [Bigg89, Chik90, Choi90, Oman90b]; to solve the corrective and adaptive maintenance problems.

On reporting of any software error during operations, level-one probes can be activated. Browsing through the messages of enabled test probes (saved by the probe monitor), the maintenance engineer may localize the problem in a specific module. Once a module has been identified, the engineer can disable probes related to all other modules, and enable probes of level-two of the suspected module, and so on. In this way, by dynamically controlling (i.e., activating and deactivating) the test probes in a systematic way, bugs can be localized easily, thus, reducing testing and maintenance time and efforts.

The ability to view the software behaviour at a global level, and then slowly zoom-in to the possible problem area, is very useful in fault isolation. Only when the fault has been localized to a small part of the code, debugging aids need be used.

The above scenario shows, that in localizing the problems, the maintenance engineer uses probes that were thought at design and implementation phases itself. If the probes are not able to guide the maintenance team to isolate the bugs, it would mean that proper attention was not given to the observability (and controllability) measures of the software during design and implementation phases.

3.5 An Example from TIFACLINE HOST Project

To illustrate the concepts presented above, we will take an example from TIFACLINE HOST software [Tifa90a, Tifa90b]; a large project in which these concepts were used in design verification and in tuning up of the design to deliver the desired performance. The following section gives a brief outline of the project. Section 3.5.2 and 3.5.3 describe two scenarios of faults for illustrating the use of probe mechanism.

3.5.1 Brief out-line of the Project

The TIFACLINE project consists of a number of host machines, each located in a different location and having information on certain technologies available within the country. Information in all the hosts put together constitutes the complete technology database. The host software provides to the user a query facility over this distributed database. Apart from this primary service, it also provides secondary services, like bulletin board for sharing of information among members of closed user groups and on-line interactive help facility. Figure 3.1, depicts the overall architecture of the TIFACLINE Host software.

User interface, interacts with users providing the various facilities. The Handshake module keeps information about all the resources available in the network, and helps in establishing connections to various resources either available locally or on remote sites. The Communication module takes care of message transfers across various hosts.

The Smart database contains information which helps the user in identifying the technology records which are of interest to him. The TIFACLINE database servers, service the user's request for detail information about technology records. The Help servers represents the logged in Helpdesks, which can be connected to any user asking for on-line interactive help.

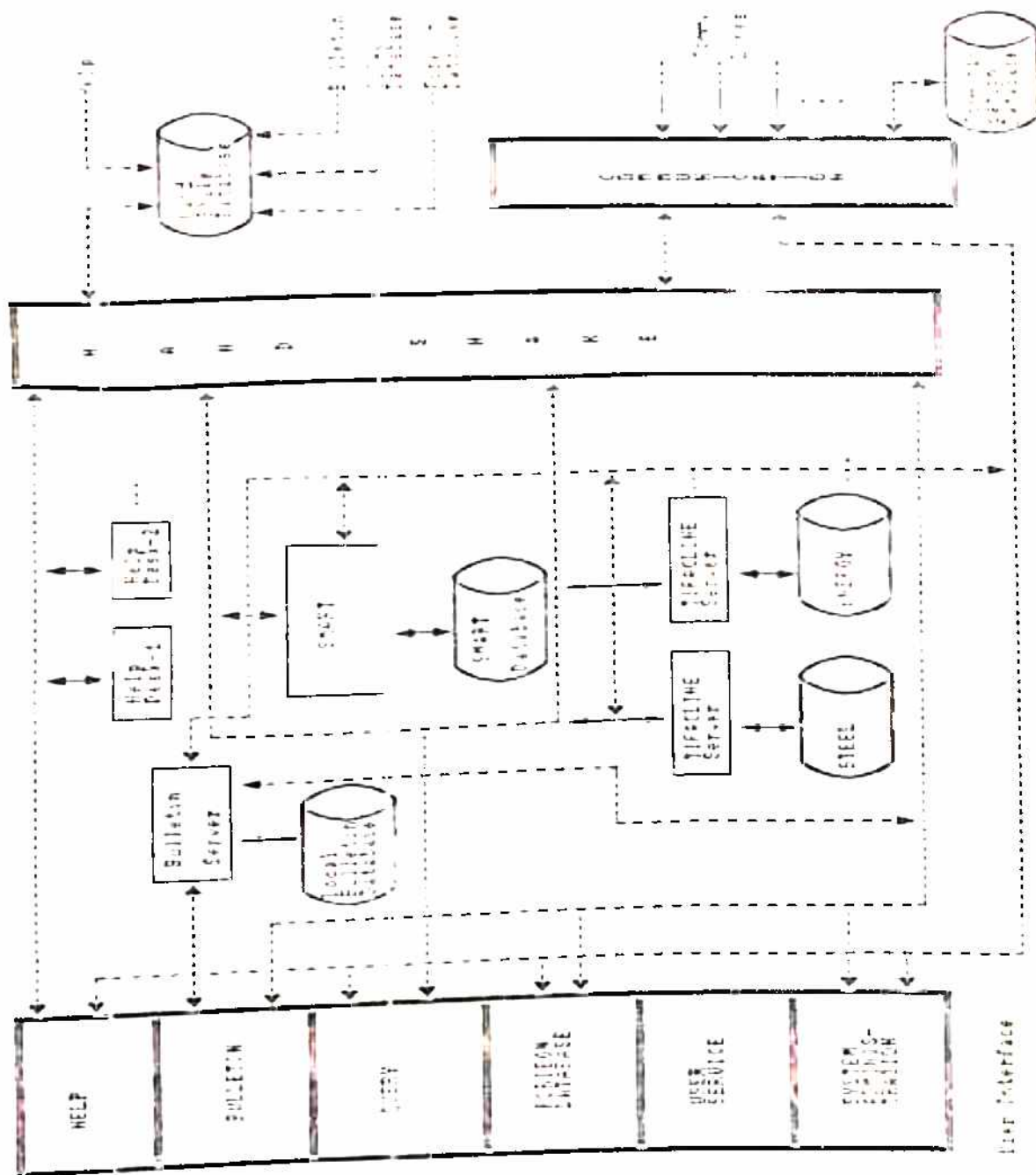


Fig31 : Detailed System Architecture

3.5.2 Error example 1 : Unable to Connect

Fault Description : On invoking Helpdesk option [Tifa90a, Tifa90b], one is not able to connect to any Helpdesk, even though one can notice a free Helpdesk.

There could have been several reasons for this fault. Some of the possible errors could be

- Helpdesk registration not done
- Open connection message not received by Helpdesk module
- Handshake not able to process open connection

Unless one gets more internal information, one cannot reach any conclusion. Using a debugger to resolve this problem would have been a very difficult exercise, as multiple processes were involved. Secondly, it would have required the presence of programmers who had developed the Helpdesk, on-line help, and Handshake modules.

Appropriate probes were activated and the event history file was examined to check the status of the Helpdesk and see the messages exchanged between on-line Help, Handshake, and Helpdesk processes.

In this case, it was discovered that Helpdesk was able to connect, immediately after it registered itself with Handshake module. Afterwards, Helpdesk was not able to connect, because, on closing of the previous connection, Handshake module was not informed by Helpdesk, so that it could reset its status from 'busy' to 'available'. Therefore, next time when a request for 'open connection' was received from Help module, Handshake reported 'no free Helpdesk', even though one Helpdesk was available. More detailed level probes were activated to localize the fault, further, in the code.

Thus, by activating appropriate probes and analyzing the event history file, one could localize the problem to a small section of

the code. The programmer who wrote the code, analyzed the faulty part of the code further, using a debugger and the error was rectified.

If probe like mechanism had not have been incorporated in the code, the problem would have involved all the concerned programmers, and would have required more effort in the initial stages of fault diagnosis itself. In this case, only the concerned programmer was required to be told about the faulty part of his code, after the problem had been analyzed and sufficiently localized by the system designer.

3.5.3 Error Example 2 : Lost Messages

Fault Description : Some of the messages exchanged by on-line help and Helpdesk are getting missed.

On report of the above problem, the event history file was examined, which had only macro level or very important messages, to avoid excessive overheads. As the recorded information was not sufficient, more detailed level probes were activated to get more information about the internal processing.

It was discovered that the messages were being put on queues, but were not received. The 'send message' call was not programmed properly. The returned 'error code' was not being processed by the software. Once error was localized to a very small section of the code, attention was focused on the problem area and the error was rectified.

Thus, probe mechanism helps in identifying the likely area of error, for close scrutiny, where very limited possibilities needs to be examined thoroughly, mentally or using a debugging tool. In the absence of such an efficient and flexible mechanism, adhoc methods, including debuggers, are generally used which consume large amount of time and effort in this initial phase of error localization. Probe mechanism make error diagnostic more systematic

and efficient, reducing the effort and time significantly.

3.6 Event Message Keywords

In order to facilitate query over the event history file, to get information about the various aspects of the software behaviour, it is proposed to attach one or more keywords as part of the event message.

To illustrate the usefulness of this concept, an example from TIFACLINE Host software [Tifa90a, Tifa90b] is given. Various modules of this software, exchange messages for various services. Each message can be associated with some of the following keywords:-

<description>	<keyword>
Open connection	oc
Register module	reg
Send message	s_msg
Receive message	r_msg
Help Module	help
Handshake Module	hs
Helpdesk	h_desk

Associating relevant keywords with every exchanged message, would make queries for various aspects of the software behaviour very simple. For example, a message sent from Help module to Handshake module, for opening a connection with a Helpdesk, can have several keywords, like

```
MSG_KEYS : help/hs/s_msg/oc
```

Similarly, Handshake module, on receiving this message from its message queue, can record a probe message with keywords

```
MSG_KEYS : hs/help/r_msg/oc
```

indicating that a message was received by Handshake from Help module for open connection.

Several useful queries can be made over the event history file using these keywords. For example,

```
Select * where MSG_KEYS = *help* and
        MSG_KEYS = *hs*
```

would retrieve all the messages exchanged between Help and Handshake modules for operations, like open connection, close connection, etc. If several Help modules were operational, then, interaction of all the Help modules with Handshake would be retrieved. Messages of individual Help module could be identified using the process_id field in the message. Similarly, a query like

```
Select * where MSG_KEYS = *help*   and
        MSG_KEYS = *s_msg* and
        MSG_KEYS = *r_msg*
```

would display all messages sent or received by Help module to and from any other module, which includes Handshake and Helpdesk modules. Similarly,

```
Select * where MSG_KEYS = *hs*     and
        MSG_KEYS = *h_desk*
```

would retrieve all message exchanged between Handshake and Helpdesk. These would be for initial registration by Helpdesk, and subsequently, open connection requests and replies, close connection, and de-registration, etc.

Each of such query, represents some aspects of the software behaviour which is useful for analysis. Associating message keywords, is only one of the possible mechanisms, proposed for extracting the desired aspects of the software behaviour from the event history file. In debugger environment, such kind of analysis

indicating that a message was received by Handshake from Help module for open connection.

Several useful queries can be made over the event history file using these keywords. For example,

```
Select * where MSG_KEYS = *help* and
        MSG_KEYS = *hs*
```

would retrieve all the messages exchanged between Help and Handshake modules for operations, like open connection, close connection, etc. If several Help modules were operational, then, interaction of all the Help modules with Handshake would be retrieved. Messages of individual Help module could be identified using the process_id field in the message. Similarly, a query like

```
Select * where MSG_KEYS = *help*   and
        MSG_KEYS = *s_msg*   and
        MSG_KEYS = *r_msg*
```

would display all messages sent or received by Help module to and from any other module, which includes Handshake and Helpdesk modules. Similarly,

```
Select * where MSG_KEYS = *hs*     and
        MSG_KEYS = *h_desk*
```

would retrieve all message exchanged between Handshake and Helpdesk. These would be for initial registration by Helpdesk, and subsequently, open connection requests and replies, close connection, and de-registration, etc.

Each of such query, represents some aspects of the software behaviour which is useful for analysis. Associating message keywords, is only one of the possible mechanisms, proposed for extracting the desired aspects of the software behaviour from the event history file. In debugger environment, such kind of analysis

is not possible, which is important for facilitating error diagnosis.

3.7 Probe Coverage

Code coverage measurement, is used to find out the parts of the software which have not been exercised, so that more tests can be planned and executed to exercise those parts [Beiz90, Mill84]. We assert that, code coverage is more meaningful at unit level and not at system testing level.

During unit testing, complete code coverage can be achieved as greater control is available to the programmer. Programmer completely understands the logic which he has built, and, therefore, can exercise it exhaustively, by direct or indirect means.

System testing may not necessarily result in complete code coverage, even if the entire functionality is exercised. This is because, certain decisions taken during the design of the software, are not reflected in the user specification. These can only be tested using the design-based tests, during module and integration testing.

To measure the extent of system testing, probe mechanism can be used very effectively. All the important activities of the system can be associated with appropriate probes. Coverage of such probes can be used to identify the functionality not exercised. This information will be at a higher level of abstraction, i.e., functionality of the system which is easier to correlate, by the user and designer of the system; compared to the information provided by code coverage tools, which has to be interpreted by the programmers to identify the related higher level concept.

Thus probe-coverage, i.e., coverage of certain set of identified probes can be a very effective mechanism to measure the extent of system testing, and identify the functionality of software, left

untested.

3.8 Observability in Software Systems

Some degree of observability is present in all softwares. An input usually produces an output. If the system is simple and small in size, by looking at the output, one can ascertain its correctness. Any abnormal output can be analyzed, and corresponding faulty code can be localized, identified, and debugged. However, as software gets more and more complex, and large, an incorrect output may not immediately lead to the faulty piece of code, for debugging. In such a situation, the fault has to be diagnosed by executing the software and examining or observing intermediate processing.

The degree to which observability is present in a software, depends on its nature. The following sections, discuss the level of observability present in scientific and non-scientific software, and also computer hardware systems.

3.8.1 Observability in Scientific Software

In scientific software, the output is usually a direct function of the inputs. Whereas, in many non-scientific applications, the output is dependent on the inputs as well as the state of the software. Therefore, in scientific software, analysis of the output should be sufficient to check the correctness of processing done by the software.

However, it is usually very difficult to say anything about the correctness of processing by just observing the output results, as the inputs go through very complex mathematical transformations. Therefore, the stress in scientific testing techniques, is to detect as many errors as possible in the process of transformation of the algorithm into programs. For example dataflow techniques basically try to identify coding errors.

In scientific software, observability is essential, but presence of desired level of observability alone is not sufficient enough for purpose of fault identification and fault diagnosis. In scientific software, an algorithmic solution is transformed into a program. The first important desirable step, would be to ascertain that execution follows the desired course of computation. For this purpose, source code level trace facility, is highly appropriate. It can indicate the control flow, as well as the various values computed which affect further control flow. Thus, observability, in the form of source level trace, is very important for scientific software.

3.8.2 Observability in Non-Scientific Software

In non-scientific software like system software, application software, etc., it is relatively much easier to ascertain correctness of processing by observing the output values and some internal behavior. Moreover, the processing done is usually quite simple, but the number of inputs and their combinations, on which the processing depends may be large. Thus emphasis, in non-scientific software testing is on exercising the software as thoroughly (using all possible meaningful combinations of inputs) as possible, so that malfunctioning, if any, can be detected by observing the outputs.

The behavior of the interactive part of the software, is visible to the user, and hence, there is no need for building any observability for it. However, the application logic, may require some level of observability to be built especially where the logic is complex. Generally, most of the application logic is fairly simple. The results of processing are recorded in the database, which are available to the user / tester through various reports and interactive queries. Only for those parts of the software, where complex processing is being performed, additional internal observability would be desirable.

Certain third generation programming languages (like 'C', in which considerable application software development takes place, for efficiency considerations), which are quite error prone, observability can prove to be very useful.

Software which are highly interactive in nature, like word processor, spread sheets, etc., errors in software will result in improper responses which are visible to the user. Thus many faults will get discovered simply with the use of the software. To help in diagnosis of these faults, some level of internal observability would be useful, as such softwares are quite complex internally compared to applications software, described above.

3.8.3 Observability in Hardware Systems

In-built observability is very intrinsic to hardware systems. Probe or test-point concept is an integral part of integrated circuit (IC) chips/cards design. The hardware itself is structured into modules with well defined interfaces. Test points brought out at interfaces have no role in operation of the circuitry, but for ascertaining the correct functioning of various internal components at the time of testing or fault diagnosis. In other words, testability of IC chips/cards is an important criterion in chip/card design. To facilitate testing at later stage, additional circuitry is introduced permanently, and test points (extra pins) are brought out [Levi92].

In hardware, observability is built for diagnosis of faults, which may get created during manufacturing process or due to wear and tear of the system with use. It is not meant for detecting fault in design of the hardware.

In software, as the complexity of the system is much higher, not all the faults are possible to remove, even after very rigorous testing. Some provision is required to help in the diagnosis of residual faults. Observability in hardware, is for diagnosis of

faults which occur due to physical wearing out of components, which is different from reasons of faults in software. Nevertheless, observability has been found to be very useful and essential for diagnosis of faults. In software also, observability is required and is used in fault diagnosis, but it is not pre-planned. Building observability a-priori, will considerably reduce the effort required to get access to internal information later.

Probe mechanism provides a means to embed observability, which can be activated to the desired level of details, whenever required.

3.9 Debugger and Probe Mechanism : A Comparison

3.9.1 Debugger : The Salient Features

Debugger has been used very extensively for testing and error diagnosis. It provides tremendous amount of execution control over the software. The tester can put breakpoints at a function as well as statement level. Various global and local variables can be examined and their values changed during execution, on the fly, if so desired. Advanced versions of debuggers, permit break on change of values of variables or whenever the desired condition defined in the form of an expression holds true. The program stack can be examined to trace the sequence of calls. Software can be executed in step mode or till the next break point. Thus debugger provides a great deal of controllability as well as observability. However, debugger approach to testing, has the following limitations :

(i) Author Testing (Small Programs):

The above approach to program testing and debugging, requires that the tester thoroughly understands the program, as break points have to be put at appropriate places in the code, and variables to be examined must be known in advance. Therefore, debugger is most suited for unit testing by the author of the program.

(ii) Interactive and Less Efficient

Debugger requires more effort and time in testing and error diagnosis, as a lot of manual intervention is involved in this process. To examine the content of variables, appropriate break points have to be put. On occurrence of a breakpoint, commands can be given to display the contents of required variables. In case, the tests have to be repeated, earlier effort of defining breakpoints and display commands has to be repeated again. Though, debugger provides a sufficient mechanism, it is not a very efficient or productive mechanism for testing and diagnosis of errors.

(iii) Not Suitable for Large Software / System Testing

Debugger is suitable for testing of small programs by the author of the program. System testing of large software, which is developed by several teams of programmers, becomes more difficult, as programmers only understand their own code thoroughly. The designer understands the whole system, but is not familiar with any of the code written by programmers. The designer and all the concerned programmers have to work together for fault diagnosis, which is very tedious and time consuming.

Moreover, as the size of the software increases, from unit testing to system testing, what to observe becomes harder and harder to decide interactively and in on-line mode. Flexibility to view or change any internal information, becomes counter productive as the view increases beyond a manageable size.

(iv) No History for Post Analysis (Only Snapshot at Break-point)

Debugger does not record any information for post-analysis. Debugger provides whatever information is required when it arrives at any breakpoint. There is no option with the tester

to post-analyze the test results (internal as well as external information) away from the system. Intermittent faults, are difficult to diagnose using the debugger, as it may not occur during the debugging session.

(v) Not Suitable for Corrective Maintenance

During the maintenance phase, when a fault is reported, the software maintenance engineer has a difficult time in locating the error, as he is not thoroughly familiar with the code (required by debugger) because, usually, he was never involved in the design or coding of the software. Even if he was part of the development team, in a large software he would have handled only a part of the coding. Debuggers can help once the fault has been localized to a small part of the code, which the software maintenance engineer can understand without much difficulty. Therefore, debugger is not an effective tool for fault diagnosis during the maintenance phase.

(vi) Limitations in Handling Distributed Systems

Debugger can help in creating states, which are data dependent, and not time dependent. In fact, if it is used to diagnose a time dependent fault, it seriously affects the dynamic behaviour itself, making it difficult to observe the problem. In distributed systems, complexity is largely due to dynamic interaction of various processes. Many faults are related to this dynamic behaviour. The conventional sequential debuggers can handle one process at a time. Even if these are extended to handle multiple processes, they will not be able to deal with time-dependent faults for reasons explained above.

3.9.2 Probe Mechanism : The Salient Features

If access to internal processing can be provided by some simpler means, a major part of the test and error diagnosis can be performed without the need to use the debugger. Debugger may be used when the error has been localized to a function or code level. In probe mechanism, information of interest can be identified and appropriate probes can be inserted permanently in the code. Properly structured probe identifiers can be used, so that probes at various levels of details, can be referred to easily for activation and deactivation. The information captured by activated probes, is recorded in a log file, which can be analyzed using a query language. Most of the test analysis and error diagnosis can be done within this framework. Probes once identified and embedded can be used whenever any testing is to be performed, or diagnosis of any reported problem is to be done.

Externally controlled, permanently embedded, probe mechanism has the following important features, which make it a more effective testing tool for large software.

(i) Pre-Planned

Observability using the probes mechanism is pre-planned. Probe are identified during design and coding phases for various purposes, and embedded in software during the coding phase. These probes can be activated to get the desired internal information during the execution of tests. Though it requires efforts in this planning process, but later, one simply has to activate relevant probes to log / display the information necessary for any test verification or even fault diagnosis. Designers and Programmers of the software can easily identify the information necessary for various purposes, including that required during the corrective maintenance phase. Dependence on code understanding, gets reduced considerably, for the initial stage of fault localization.

(ii) Software Behaviour Categories and Levels

Probes can be identified for various purposes like code verification, design verification, maintenance phase fault diagnosis, software monitoring, etc. The required behaviour of the software can be observed with different categories of probes. Moreover, for each category of probes, behaviour of any part of the software can be observed at any desired level of detail.

(iii) System Testing / Post-Release

During integration testing, debugger cannot be used very easily, as control flows from module to module, which are developed by different programmers. Observability, as provided by probe mechanism, is more convenient, as break points are not required to be put dynamically. Higher level design and code category probes can be activated to observe the behaviour of the system at global level. In case of any abnormal behaviour, more details of suspected parts can be observed. Thus faults discovered during the system testing or in post-release period, can be localized very easily to small part of the code, which can then be rectified by the concerned programmer.

(iv) Performance Measurement / Monitoring / Tuning

Performance of various components of the system can be measured using the timestamp information in probe messages. It can be used to identify weak points in the software so that these can be strengthened by appropriate minor or major redesign. During the operational use of the software, the usage can be monitored and control parameters of the software tuned to get better performance.

(v) Post Analysis

Probes to observe the desired behaviour can be activated. The probe messages get recorded in the event history file, which can be analyzed later using a query language. Thus, important information about the faults can be gathered during the live operation of the software, and analyzed, later off-line. Based on this analysis, more detailed diagnosis can be carried out using the development copy of the software. In distributed systems, the event history file can be used to replay the exact sequence of processing for easier understanding of the otherwise complex dynamic behaviour of the software.

(vi) Corrective Maintenance

During maintenance phase, the maintenance engineer generally does not have a complete understanding of the code, as he was not involved in the development process. Even if, he was involved in the development process, he would not have code level understanding of the entire software; as large softwares are developed by several teams of programmers. Using probe mechanism, macro and micro categories probes can be identified and embedded in the software to help in localizing the faults to the procedure or small part of the code, which can then be debugged. This initial process of localizing the fault to a small part of the software would otherwise be very difficult.

(vii) Distributed Softwares

Probe mechanism is a very effective mechanism to handle software for distributed systems. In a distributed system, one cannot break the processing to analyze dynamic / temporal faults, as it would affect the dynamic behaviour of the distributed software itself. In such systems, probe like mechanism has been found very effective. The probe overheads

can be reduced to very low levels. The various event messages recorded in event history file can be analyzed off-line. Replay facilities can be built to recreate the actual sequence of events to facilitate the understanding the dynamic behaviour for purposes of fault diagnosis.

Proposed Probe Mechanism vs. Event-based Debugger for Distributed Systems : Event-history recording used in event-based debuggers for distributed systems, primarily address the problem of repeatability of dynamic behaviour faced while using conventional sequential debuggers [Cheu90, Grac84, McDo89]. The information recorded in event history file is analyzed for fault diagnosis.

Some of the event-based debuggers record only essential information in event history file and use it to synchronize re-execution for reproducing the same dynamic behaviour [Jones87, Lebe87, Mill88]. During re-execution, breakpoints can be put and variables examined for more detailed information for diagnosis. Observability is incidental in these approaches, as stress is still being laid on use of debuggers in replay mode.

In all the above approaches, the importance of event recording, in sequential software, is not being stressed at all. Even when analysis is entirely based on event history information, various levels of abstractions of event information is not being supported. Also, in what phases of software development, events for recording are identified and embedded, are not clearly defined.

3.10 Summary

The important points of this Chapter can be summarized as follows.

- Observability i.e. access to internal processing at various levels of abstractions is very essential for ascertaining correctness of test execution and fault rectification.
- Print mechanism and debuggers provide observability, but are very efficient mechanisms only for unit testing. These are not suitable for system testing, acceptance testing and fault diagnosis for corrective maintenance. Also, debuggers cannot be used for analysis of dynamic behaviour of distributed systems.
- Externally controlled probes, permanently embedded in the code, is a general mechanism, which can provide any desired level of observability in the software.
- Concept of probes for capturing behaviour of the software for various levels of abstractions, like design verification, code testing, fault diagnosis during corrective maintenance, and system monitoring, has been introduced.
- Probes for various purposes are planned during the design and coding phases and embedded by the programmer in the software. What to observe for what purpose is planned a-priori once for all, unlike in debugger approach. It is, therefore, very effective for system testing and corrective maintenance.
- Probe messages with timestamp information can be used to measure performance of the designed system and identify weak links for further tuning or redesign. During the post-release phase, such information can also be used to tune various system parameters for better performance.

- As probe mechanism does not alter the dynamic behaviour of the software (except cause marginal processing overheads), it can be used in distributed systems to analyze problems related to the dynamic behaviour, at system level.
- Observability can be attached or detached as required. Attached observability can be activated and deactivated externally. Only activated probes generate messages, which are recorded in event history file.
- Probes are identified by structured identifiers, reflecting the structure of the software and various levels of abstractions. Probe can be referred to by a generic mechanism for activation / deactivation of the desired set of probes.
- The probe messages logged in event history file, can be analyzed using SQL like query language. Messages can be associated with multiple classification codes, to facilitate query for pre-defined purposes.
- Code coverage is more meaningful for unit testing and not for system testing. During system testing of large software, 100% code coverage is extremely hard to achieve. For large software, functional coverage is more relevant during system testing; which can be measured using the proposed probe-coverage concept more effectively. We define system testing to be complete when 100% probe-coverage has been achieved.
- The observability built in the software using the probes mechanism, can be documented in the form a 'maintenance manual'; which can help the maintenance team to localize the fault, with considerably less effort and cost.

END OF CHAPTER 3

4.1 Software Design : Need for Verification and Analysis for Improvisation

Development of software, as a solution to a complex problem, has a distinct design phase. Ideally, the design activity means, exploring several alternative solutions to the same problem by same or different set of persons; comparing the merits and demerits of all the proposed feasible solutions, and then selecting the best possible alternative for the detailed design and software implementation.

Unfortunately, the design phase in software development does not go on these lines. Generally, an obvious solution is adopted for more detailed analysis and improvisation. Certain problems, like problems in distributed systems, are quite complex intrinsically. Major contributors to this complexity are asynchronous processing by multiple processes, on same or different systems, with communication links, which may not be reliable.

The design of such complex system cannot be shown to be correct easily on paper. Many aspects are not even comprehended clearly, unless the design is put to some test. Therefore, testing of complex software systems consists of, not only verifying the developed software against the design, but also validating the correctness of the design itself.

Usually, all the functionalities built in a software are not easily testable by using only the commands available at the user interface level. It is very important to test the design very thoroughly so that its correctness can also be ascertained. Some extra efforts are required to test such aspects of the software which are not easily testable.

Performance, which is an intrinsic requirement of any system, cannot be predicted very accurately, just based on the design. Variables, which affect the end response, are so many, that it is difficult to model the end response in terms of these variables. Therefore, measuring performance of the developed system becomes very essential, after the system has been built.

Based on the performance of various sub-components, the design can be judged comprehensively. In case, the performance is not upto the mark, weak links can be identified for improvement, or redesign and development, to achieve the desired level of performance. Sometimes, a major redesign effort for the whole software may have to be undertaken, since minor adjustment and component-wise redesign may not be sufficient.

4.2 Design Verification Using Probe Mechanism

Software life cycle model, consists of well defined phases of development, which include user specification, design, coding, testing, implementation, and maintenance phases [Gilb88, Pres92]. On analysis of user requirements, the specification document is prepared, which is given as an input to the design phase. The output of design phase is a design document, which should be validated against the specification document. The design document is an input to the coding phase. Once the coding is complete, apart from code testing, it must be verified against the design as well as the specification documents [Hetz88, Ie89VV, Neuf93, Ould86].

No approach towards testing of code, distinguishes the role of designer from that of programmers. Usually, all approaches presume that either both the programmer of the code and designer of the system are one and the same person, or they are testing the program together. We assert that the verification of design is a distinct activity and it can be done by the designer himself. A programmer's activity is restricted to unit level testing of software, and his responsibility is limited to the correct implementation of

algorithms.

Design document not only specifies the design of the software, but it also further details or concretizes the user specifications. Further, the design document defines the overall architecture of the software and behaviour of the various modules/sub-modules under various situations. The description is somewhat like rule based specification.

Verifying the behaviour specification of the module/sub-module need not require the detailed knowledge of the underlying code. If the necessary behavioral information can somehow be made available to the designer, he can cross check the actual behaviour against the specification in the design document. Issue of design verification arises at the time of integration and system testing, and not at the time of unit testing.

Integration testing is performed after each individual module has been tested at unit level. In integration testing, some or all modules are put together and tested as a whole. It may reveal further errors, which may be due to misinterpretation of module specifications, mismatch or inconsistency among module specifications, or unacceptable performance of the system as a whole. We characterize all these errors as design errors or design interpretation errors. These errors can be traced during design verification time only.

To verify his design, a designer will apply different test cases to the software and will like to execute specific control flows, or will like to gather different kinds of data during software execution than those required by the programmers to test their implementation.

We now evaluate various testing approaches for their suitability for design verification.

4.2.1 Limitations of Existing Approaches

Debugging have been used traditionally for unit, module and even system level testing [Akar91, Tsub86]. Several attempts have been made to enhance the capability of conventional debuggers [Agra91, Kore88, Shim91b]. Debugging cannot be a good mechanism for design verification. Though any part of the state of software can be examined at breakpoints, it becomes difficult to decide which part of the state is important for design verification. The act of debugging tends to disturb the natural flow of processing itself, and hence, performance and issues related to timing cannot be tested.

Black box testing verifies functional specifications and hence it is not useful for design verification. White box testing is meant for verifying the internal control structures and is generally used for unit level testing of algorithmic aspects. Similarly, the concept of static testing involves off-line analysis of code with the intention of finding logical errors in design and coding at unit level.

Since issues of design and algorithm are so interwoven in the code, the designer normally has to go through the entire code for design verification. Furthermore, the largeness of the software makes these approaches unsuitable for design verification.

4.2.2 The Event History Approach using Probe Mechanism

In the event history based approach, the testing activity has four steps: (1) construction of a test case; (2) embedding code with probes to record all important information and events relevant for the test case; (3) application of a test case to the software and event recording; and (4) analysis of the recorded event history [Bate83, Cheu90, Grac84, McDo89]. Following this approach, software can be run under various test cases. The recorded event history can be analyzed later to check whether the software conforms to the specifications or not. The designer can easily identify test cases

and related internal information for design verification. Thus event history approach can be readily used by the designer for design verification.

4.2.3 Design Verification & Event History Approach

The techniques largely used for design verification require an understanding of architecture of software, and control flow and data flow at a higher level. A programmer who concentrates at unit level, does not have a total perspective of the software. And hence, he cannot have a feeling of events that are important for overall design verification. To verify his design, a designer will have different test cases than those constructed by the programmer. Again, the designer will like to get events recorded at different points in code execution for its design verification.

A designer, apart from providing the functional specification of each module to programmers, must also ask programmers to embed probes at appropriate points in the code so that events that are useful for design verification can be recorded. In addition to design verification, the designer is also responsible for ensuring a desired level of performance of the software. Unlike functional specifications that can be frozen during the design phase, performance of software cannot be guaranteed by the designer in advance. After coding is complete and software is integrated, the performance of the software is studied by the designer using specific test cases. In case, the performance is not satisfactory, the software will require tuning which can be done only when the performance of each software component is known. The components that cause bottlenecks, can be re-coded to improve performance. If the performance criteria are not met even after re-coding of components, some modules or a set of modules will have to be re-designed. To study the performance of various modules/sub-modules of the software, the designer has to specify, in advance, what events are to be recorded and where these events occur in the software.

We have extended the event history approach to take care of design verification. In our scheme, probes have been categorized into design and algorithm probes (also introduced, briefly, in Chapter 3 Section 3.2.1). Algorithm probes are useful for unit level testing by programmer, whereas the design probes are used for design verification. To understand performance behaviour, probes are time stamped.

In Section 4.3, the concept of design verification is discussed in detail, and the Section 4.4 describes the the design and algorithm probes suitable for design verification. Section 4.5, gives an example taken from a large distributed database project, viz., 'TIFACLINE host software', where some of these concepts have been used in practice and found useful.

4.3 Design Verification

For large software, the design document gives the overall architecture of the software and specifications of all components viz., module, sub-modules, processes etc. It specifies modules interface specifications, modules behaviour under various situations, interactions among modules, and performance criteria. The behaviour specification is very similar to the rule based specification in expert systems. Implementation of these rules into a coherent procedure is largely the responsibility of the programmer.

To verify this behaviour, it is not necessary to have an understanding of the underlying code as long as the information necessary to evaluate the behaviour can be made available to the designer. For example, if module 'A' has to send a message to module 'B', it is sufficient that whenever a message is sent, this event is recorded with all the necessary information for study by the designer later. It is not at all relevant as to how the programmer has implemented it at code level. The correctness of the underlying code to achieve the desired behaviour is the responsibility of the programmer.

Design specifies the behaviour of the module at a certain level of abstraction. Thus, test results should provide the designer with the behavioral information at the same level of abstraction so that it is easier for him to match the actual behaviour with that laid down in the specifications and also with his expectations.

4.3.1 Information Required for Design Verification

During the execution of the software, one needs a tool to record all information of importance for post-execution analysis. It may consist of values of parameters or various intermediate results. However, it may also be necessary to have access to the system's global state which may be present in various data structures in local or shared memory, message queues, files, and databases, etc.

Again, to understand the performance behaviour, timing information of various events is very important. If, by execution of software, one can identify the components that are consuming too much of time, thus causing poor response, then these components can be tuned, re-coded, or even redesigned to achieve the desired response.

4.3.2 Probe Mechanism: A Mechanism for Event Recording

The probe mechanism is best suited to record the various events occurring in the system. Probes can be embedded anywhere in the program by the programmer. We have enhanced this concept to take care of the issue of design verification.

The message parameter of probe function can contain all the important information relevant at the point where the probe is placed in the code. By placing probes at appropriate points in the code, all the information required for design verification can be gathered. Moreover, using the information generated by various probes, one can reconstruct the global state of the system as well.

Temporal behaviour of the various components of software can be well understood, as the probes carry the timestamp information.

4.4 The Enhanced Probe Mechanism

The probe mechanism can be used to capture any type of information. The designer identifies the important information for his use. The programmer can also use it to get information to help in the testing of programs. If the designer can distinguish its events from that of programmers, the event history file can be browsed independently by the designer and programmers. This makes the designer independent of the programmer.

4.4.1 Probe Level & Probe Category

The probe mechanism talks about probes at various level of abstractions so that the problem can be understood in top-down fashion [Gupt91]. What constitutes each level of abstraction has been left open deliberately. For example, a designer may designate software specifications and module specifications, as the first and second level of abstractions. Depth of levels depends on design approach, complexity, architecture, and size of the software.

Similarly, the internal behaviour of software can be studied at various categories of abstraction. We define two orthogonal internal categories of abstraction i.e., the design category and algorithm category. Both of these can be captured by probes, which can be termed as design-category and algorithm-category probes, respectively. Since a probe-id is a unique structured identifier, a probe can carry its category as a part of its probe-id. We propose that probe-id, apart from its category should carry the module name, function name, and a label to identify its position in the code.

Examples of Probe-ids :

D/Help/Input.5.1
A/Help/Display.5.2.3

The first probe_id represents a design category probe (denoted by 'D' in the first part of the probe_id) of Help module with probe name, "Input.5.1". The second probe_id represents an algorithm category probe (denoted by 'A' in the first part of the probe_id) of Help module with probe name, "Display.5.2.3".

4.4.2 Examples of Design /Algorithm Probe Usage

In this section, examples are given to illustrate the use of probe mechanism in the context of design verification.

Design Event-messages:

U_MSG_No.: 5 No._of_Lines: 3 Total_Bytes: 200
S_MSG_No.: 10 Line_No.: 2 MSG: xxxxxx Buffer_Line_No: 15
Operation: Open From: User_Process To: Hand_Shake

Design / Algorithm Probe Activation / Deactivation

Enable D/Help/*

Activates all Design category probes in Help module

Disable A/Help/Display*

Deactivates all Algorithm category probes in "display" procedure of Help module

Query over Event History File

```
Select * where U_MSG_No. = 5
```

This query displays the complete history of user message no. 5 on test-control screen.

```
Select * where Probe_id = "D/Help/*" and Operation="Open"
```

This query displays all design category messages related to operation "Open".

4.4.3 Probe Mechanism & Design Verification

This approach assumes that information required to verify the design is identified and specified by the designer in the design document. The programmers can plant the necessary probes at appropriate places in their components of the software in order to generate the desired information when the software is run in test mode.

As probes can be readily activated externally by the designer, design specification can thus be verified, independent of the programmers, and without understanding the code.

4.5 An Example from TIFACLINE HOST Project

To illustrate the concepts presented above, we will take an example from the TIFACLINE HOST project [Tifa90a, Tifa90b] described earlier in Chapter 3, Section 3.5.1, in which these concepts were used in design verification as well as in tuning up of the design to deliver the desired performance. Figure 3.1, which depicts the overall architecture of TIFACLINE HOST software, is reproduced here for ready reference as Figure 4.1.

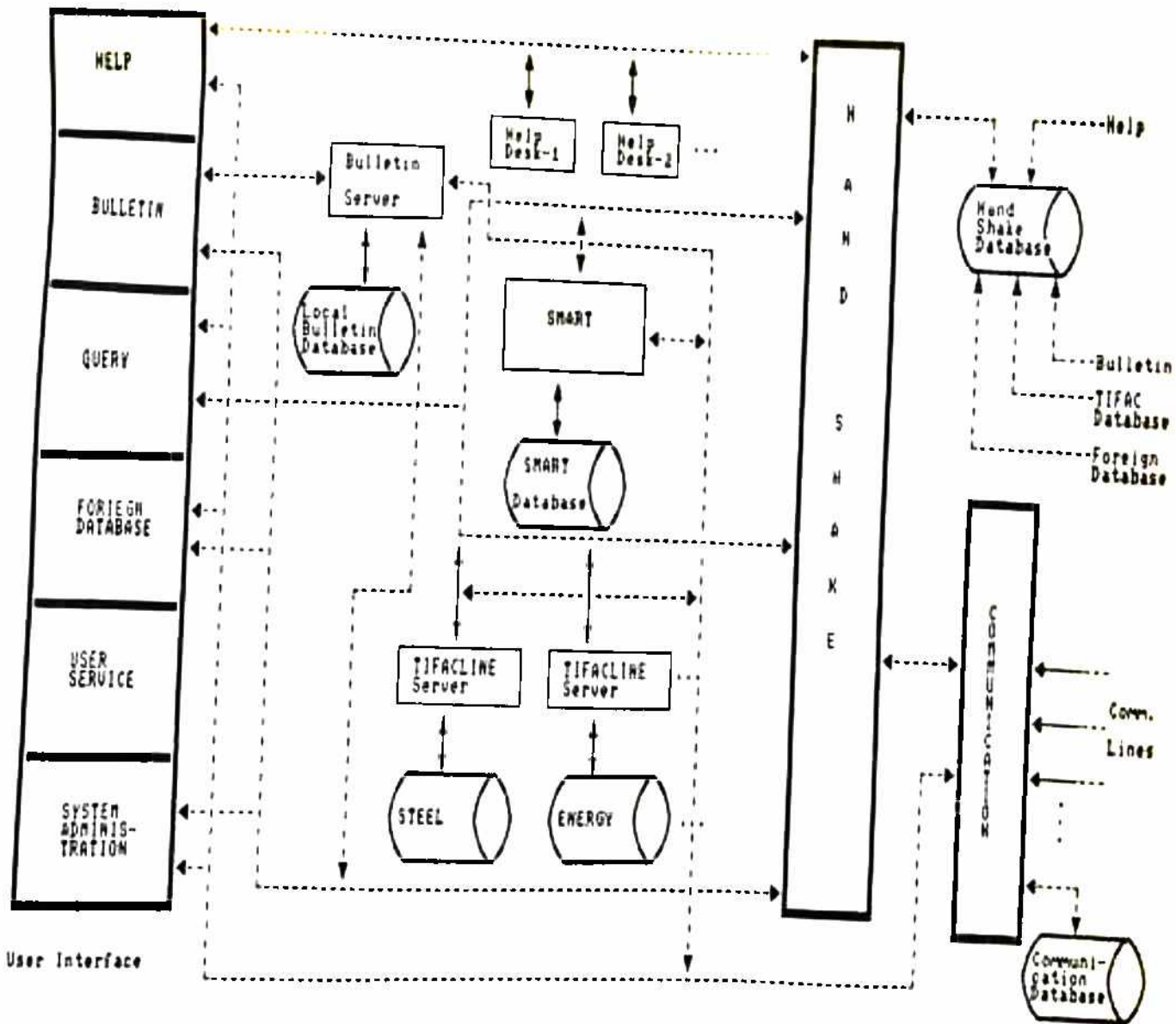


Fig.1 : Detailed System Architecture

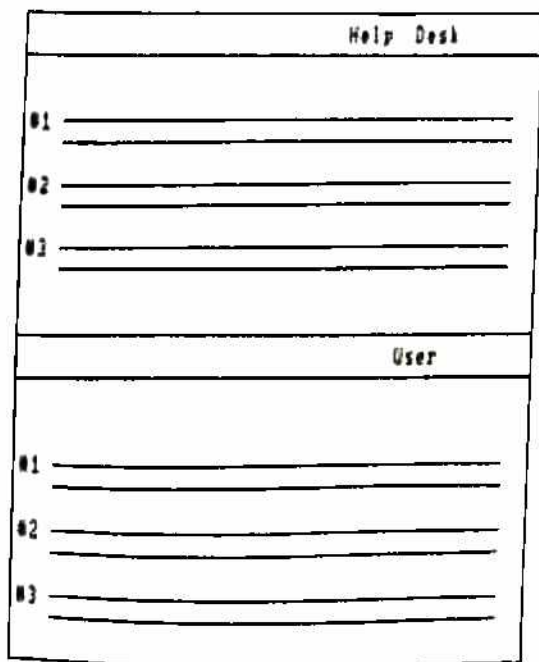


Fig.2 Help Screen

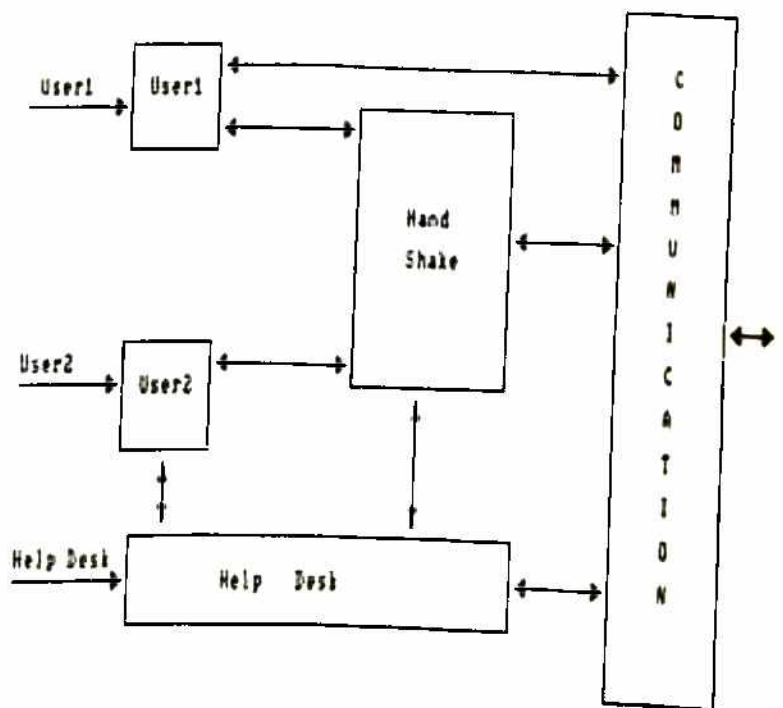


Fig.3 Help Service

4.5.1 User Specifications of Help Module

For the purpose of our discussion, we will restrict ourselves to a simplified version of the on-line Help module and its interaction with the Communication module. To make the presentation simple, we have written specifications in a simple language and in brief. User level specification of Help module can be specified as follows :-

- User should be able to connect to an available Helpdesk.
- The screen may be split into two halves as shown in figure 4.2. The upper half can display the messages received from the Helpdesk. The lower half can display the queries sent to the Helpdesk by the user.
- Each message should be numbered.
- It should be possible to scroll up and down the two parts of the display independently.
- It should have a response time of less than two seconds.

The above specification is perhaps complete as far as the user is concerned.

4.5.2 Design Level Specifications

On detailing the design, the following additional understanding emerges :-

- On choosing the Help option, a message should be sent to the Handshake requesting a connection to a free Helpdesk server available either locally or on a remote site.
- If no such connection is feasible, an appropriate message should be displayed on user's terminal and control should be returned to the previous menu.

- Each query should be packed as a message and sent to the Helpdesk directly, in case of a local server; and to the Communication module if the Helpdesk is on remote site.
- The Helpdesk process should send the message on the user process queue.
- The scroll up and down facility can be achieved by keeping a copy of the displayed messages in memory buffers. A maximum of 5 pages of messages can be kept in buffers. Whenever this limit is exceeded, the first message in the buffer will be dropped to create space for new messages.

The above is not an exhaustive list of design related specifications. Interaction between various relevant modules is depicted in Figure 4.3.

The designer does the integration to the extent that he has designed the various modules and their interactions. The interactions between various modules is also defined by the designer. For example, to establish a connection with a resource, the user process will send a request to Handshake module, which will check the request against the resources available. If the resource is locally available, it will send a request to the specific resource process. In case, the resource is available on a remote site, it will send the request to the Communication module for transfer to the Handshake of the remote site and which, in turn, will send the request to its resource process. Finally, the remote Handshake will send a reply indicating success or failure of connection requested by the user process along with other necessary information.

In TIFACLINE project, different programmers implemented the different modules. Each programmer had detail knowledge of the module that he implemented. But it was the designer who had a complete knowledge of modules' interactions in various situations.

4.5.3 Verification of TIFACLINE Host Software

4.5.3.1 Specifications Verification

To verify the user level specifications, the information available via the user interface is sufficient. There is no need for any extra information or mechanism to aid in this task. In general, the user specifications can be verified using the black-box testing methods.

4.5.3.2 Design Verification

What is essentially required for design verification is to give access to internal information which identifies the various situations and depicts the behaviour in these situations. It is not necessary to go through the whole code to achieve design verification. For example, on choosing the Help option, the first activity is that of establishing a connection with a resource; which involves exchange of messages between various modules. If this exchange of messages, can be captured by the probes and displayed along with their time-stamps, the designer can get a fair idea about the way the software is behaving under this situation. The following sample event recording depicts some aspects of the behaviour of the above software:

```
10:20:52 Help_uid : 50          To : H_shake  CMD   : Open_con
10:20:55 H_shake To: Help_desk ID : 75        CMD   : Open_con
10:20:57 Help_Did : 75          To : H_shake  MSG   : OK
10:20:58 H_shake To: Help_uid  ID : 50        Reply : OK Con.No:10
```

Similarly, each time, the message is exchanged by the user process and the Helpdesk process, necessary information can be made available to the designer for verification. Each of the design behaviour specification can be captured by suitable probes in order to completely verify the design. There is absolutely no need for designer to go into the code written by various programmers.

This mechanism was used not only to verify the design specifications of the Help service but also to resolve the crucial response requirement. Initially, the response was found to be too slow to be acceptable. The messages generated by the probes in the Help process, Helpdesk process, Handshake and Communication modules were displayed on four different windows on the test-control screen. The timing information attached with the various event-messages helped in identifying the bottleneck in the whole cycle. The slow response was traced down to excessive wait at the Communication module. The Communication module was then redesigned and tuned till reasonable response was achieved.

The designer worked at the design level and programmers worked at the algorithm level. Programmers had their own algorithm category probes and the designer asked each programmer to embed design category probes, as specified by him.

Whenever any problem was encountered during the integration of software, the above behaviour was first analyzed in terms of information at design level, which helped in identifying the weakness in the design or the incorrect implementation of the design which may be due to lack of understanding on part of the programmer. At no time, the designer was required to understand the code.

4.6 Summary

The important points of this Chapter can be summarized as follows:-

- Verification of design is a distinct activity compared to the testing of the algorithm/code.
- Permanently embedded probe mechanism has been used to define design and algorithm category probes.
- Information available through design category probes can be used to verify the design behaviour of the modules/software.
- Using the probe mechanism, the designer can verify the design without going through the code and independent of the programmer.
- Faults should be first analyzed at design level using the information available through the design category probes. This helps in localizing the fault, which can be further analyzed using the algorithm category probes.

END OF CHAPTER 4

5.1 Introduction

Designed and developed software has to be tested thoroughly to remove various errors which are intrinsic to the whole process of software development. Therefore, software goes through various phases of software testing. Test-plans are prepared after the specification has been finalized. These are made available only during the phase for system testing. Some additional tests are prepared based on the design of the system. These tests are also used during system testing. Tests are also prepared by the programmers themselves which are used during the unit testing of the modules developed by them.

Many of the tests based on specification and design (henceforth referred to as specification verification tests and design verification tests respectively) are difficult to conduct, as the states of the software required to be reached before these tests can be executed, are very rare and cannot be created easily, using the commands available at the user interface level. Many of these tests are related to special conditions, exceptional conditions and error conditions, which do not occur normally, but have to be taken care of by the software. For example, if software has provision for handling disk read error, or disk full conditions, these cannot be created readily. Various tables maintained in the software also have logic to handle boundary conditions when table is full. If table is large, it may be difficult to fill it through user interface commands. These are only a few examples of rarely occurring states to which the software must be brought for testing.

More complex systems, which cater to concurrency, automatic fault recovery, etc., have complex software to handle all type of possible situations. Many of these conditions will not occur in normal execution of the software, and also cannot be created by commands available at the user interface. For example, concurrent B-tree software has provision to handle simultaneous read and

updates by multiple users, while maintaining the consistency of the tree data at all time. Various complex situations, that the software algorithm handles through built-in programming logic, have to be tested. If multi-access b-tree software is left to run freely, the various important sequences of b-tree node traversal, for which algorithm has built-in-logic, will have very little probability of occurrence. For testing of all such difficult cases, either adhoc measures are used or these cases are left untested.

However, if the test-plans (to test that the software conforms to the specification) can be made available to the designer in the design phase, as one of the inputs in addition to the specification of the system, then some provision can be made in the design which can be coded, so that these difficult to create states or conditions can be created easily for performing these difficult tests.

Similarly, the test-plans prepared at the end of the design phase (to verify the design of the software), should be used to review the design, to check, if these test-plans can be executed readily. If required, the design should be suitably modified or extended. The above should be iterated till the design and designed verification tests converge, i.e., the design is such that it makes it possible to execute all design verification tests easily. Similarly, the same process should be repeated for the detailed design phase. During coding phase, the programmer may also have to make some provisions, to ensure ready execution of code verification tests. All such extra provisions in the software, to facilitate execution of otherwise difficult to execute tests, are termed as 'controllability measures'.

Once it is known, what tests will be difficult to conduct in normal test environment, one can work out what needs to be build to achieve testability. For example, Table handling would require an artificial way to create dummy entries in the table which are ignored by the software logic. For disk full and bad sector, additional code has to be provided that will trigger, artificially,

such conditions through some extra control.

Even interfaces constructed temporarily for internal modules, are basically controllability measures to do module tests in artificial environment, in the absence of other modules. But such temporary interfaces are generally built exclusively for testing purposes (and may get discarded / destroyed) and are not an integral part of the software, and hence, do not get designed and created with the same seriousness like the rest of the software, even though these are also required during the entire life cycle of the software i.e. beyond the testing phase; in the maintenance phase also.

In the absence of such measures, many of the tests are not conducted, and the delivered software remains untested for many conditions. In critical cases, some adhoc measures may be used, during the testing phase, to perform such difficult to execute tests. As all such measures have not been planned in the required phase i.e. the design and coding phases, adhoc measures require more effort and even then it may not lead to systematic testing required for producing reliable software.

Moreover, such adhoc measures (and also the temporary interfaces built for testing of internal modules) have to be removed, as they may not fit very well in the operational environment. Also, one tends to assume that once the test have been done, somehow, there is no need for such measures to remain with the rest of software all the time. However, as no software is totally error free, some undiscovered bugs remain in the software, even after rigorous testing. These bugs may surface during the operational use of the software. To diagnose reported faults, it requires tests to be re-applied with a view to diagnose the reported fault. If tests have to be applied, which are similar to the tests for which adhoc measures were build and discarded after the testing phase, the maintenance engineer, will be in a very difficult situation. Maintenance engineer being not so familiar with the code, as he was not involved in the design and coding, would find it extremely difficult to construct such adhoc measures for testing again.

Therefore, adhoc measures, even if used, have to be documented and left in a form, that these can be re-used, whenever required, during the corrective maintenance phase. If such measures have to last during the life cycle of the software, it is advisable that these extra measures required for execution of difficult to do tests are built as an integral part of the software systematically, during the proper phases of design and coding.

Building of such measures during the testing phase, once the design and the coding has been concretized, becomes a patch work exercise. If the need for such measures is known during the design and coding phases, then effective systematic provisions will get built and become an integral part of the software and can live beyond the testing phase. Therefore, test-plans should be used in the design to build a testable design with the required controllability measures, which are coded and documented, for use during testing as well as the maintenance phases.

Availability of such properly designed and coded controllability measures, can be very useful during the maintenance phase as the maintenance engineer is not all that familiar with the internals of the software, and needs all possible help to handle the alien code.

5.2 Controllability Development

In order to understand, how controllability can be built in the various phases of development, let us look at the conventional and the proposed methods of software design and coding.

Figure 5.1 (a) & (b) shows two software design and coding scenario. In the first scenario, design phase has specification document as input. Design phase is followed by the detailed design phase, which is then followed by the coding phase. Deliverables of each of these phases are d1, dd1 and c1 respectively (i.e., design document, detailed design document and the developed code). In parallel, corresponding test-plans are prepared and kept ready to be used during the testing phase. St1 is the specification based test-plans, dt1 is the test-plans for design, ddt1 is the test-plans for detailed design, and ct1 is the test-plans for the unit code testing. In testing phase, first ct1 is applied which is followed by ddt1, and then dt1. Finally, st1 is applied. Here test-plans are prepared, after the corresponding specification documents (i.e., d1, dd1, c1) are ready.

In the second scenario, design phase has specification as well as the specification based test-plans as inputs. It produces d2, which is used to prepare dt2. After preparation of dt2, d2 may be reviewed and necessary changes made, if required to reflect the needs of dt2. For the next phase of detailed design, both d2 and dt2 are available as input, with an objective that it should make provisions for testing of dt2.

Once dd2 is ready, test-plans ddt2 is prepared and dd2 is reviewed and revised w.r.t. the requirements of ddt2. Obviously dd2, which is now based on d2 as well as dt2, with the objective that dd2 should facilitate execution of dt2, would be different from dd1 which had only d1 as input.

Both dd2 as well as ddt2 are given as inputs to coding phase along with the objective that code produced should facilitate execution of ddt2. Code c2 is produced and has its corresponding ct2 test-plans. c2 is also reviewed and revised to take care of requirements of ct2.

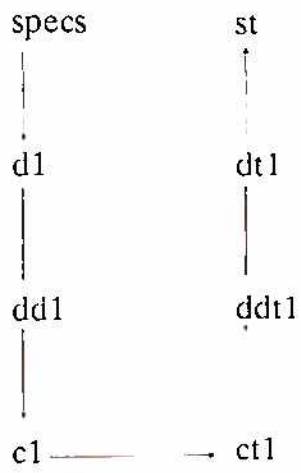


Figure 5.1(a) Conventional Approach

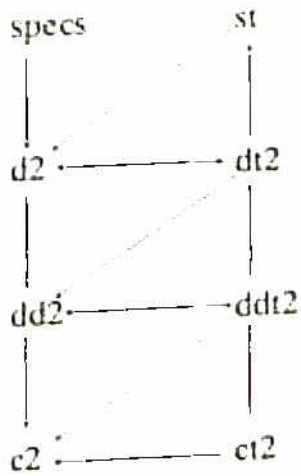


Figure 5.1(b) Proposed Approach

In the second scenario, each phase has more inputs and additional objective conditions to be met. Even if no mechanism or methodology is provided to assist design and coding in producing the end deliverable, meeting this additional criteria, the resultant deliverables are bound to be different and easier to test, compared to the corresponding deliverable in the first scenario. The degree to which the additional objective is achieved, will depend on the ingenuity and experience of the software persons involved.

In the first scenario, st, dt1, ddt1, and ct1 only state as to what to test, and how to test is left open or is adhoc. In the second scenario, st, dt2, dd2, and ct2 not only state what to test, but how to test is also taken care of in the corresponding phases and the following phases. The software produced as a result of the second approach has adequate provisions for executing all the required tests easily, and hence, is more testable than the first approach.

5.2.1 Controllability Measures

Controllability measures required in general to help in the various phases of test execution are described in this Section. Before these measures are discussed, the need to integrate all the controllability measures in the form of a 'control interface' is discussed below.

Control Interface : As controllability measures (as well as the observability measures) built in the software are not limited just to the testing phase, but have relevance even in the post release corrective maintenance, all the provisions should be properly integrated and made accessible through a proper interface. This interface is termed as 'control interface' to distinguish it from the 'user interface', through which the software is used by the user.

These controllability measures (as well as observability measures) available through the 'control interface' become accessible to the tester or software maintenance engineer only when the system has been set to 'test mode' by the system administrator.

The various controllability measures should provide the following facilities :-

1. Set Test Environment
2. Set Test State
3. Activate Observability Measures
4. Provide Test Input
5. Perform Test Analysis

(1) Set Test Environment

Initially, the test environment required for conducting the desired test is to be identified and set, using a set of control commands created for this purpose. It could be in terms of isolating a component of the system for testing, which could be a module, sub-module or a sub-system. The aspect of the component to be tested is also specified as part of the setting up of the test environment, so that relevant commands for setting state and providing appropriate input are made available to the tester. It may also activate some of the embedded controllability logic built within the selected component (it may use some control flag to signal activation).

(2) Set Test State

Once the component to be tested has been identified and isolated; and the aspect to be tested has been selected, the next step is to bring the component to the desired state before the test can be applied. Appropriate control commands will have to be provided to the tester to change the required states in the desired manner. These control commands are

implemented using additional logic built in the form of control functions, which operate on the state of the software using the inputs received through the control commands. For example, if we have to test the logic for table overflow condition, the table will have to be filled with dummy values.

(3) Activate Observability Measures

Before the test is executed on the desired state of the software, the appropriate observability measures will have to be activated so that the internal processing can be observed to the desired level of detail. Control break can also be put on certain probes so that, when such probes are encountered, control is given to the tester so that he may change the activation of the observability measure to focus attention to other desired parts as might be suggested by the occurrence of probe break.

(4) Provide Test Input

The test is executed by providing the desired inputs. Sometime the inputs may be given using the commands of the user interface itself, provided it formed part of the selected test environment. However, in general, for providing input for test execution, a required test interface may also be a part of the controllability measures. This may happen, if internal modules or components are being tested, which do not have associated user level commands for the desired test. In some cases, a whole control module may have to be developed to provide the desired test inputs for testing (e.g., the case of communication module, see Section 5.5.5 of this Chapter)

The processed output may be directly available to the tester or he may have to use some commands, which are either a part of the user interface or have been built as part of the

controllability measures. During test execution, active probes messages get recorded in event history file.

(5) Perform Analysis

The output test results as well as the active probe messages recorded in the event history file, can be analysed. For analysing the event history file, a query language can be used, as event history file has large amount of information which may be difficult to analyse manually.

If the test is being conducted for purpose of fault diagnosis, the above may have to be repeated several times, as initially only higher level probes would be activated to localize the problem to a sub-system. As fault get localized to smaller and smaller part of the software, more detail level probes will have to be activated. Once the fault has been localized to a small part of the code, a debugging tool can be used to debug the error. After correction of the software, the above process should be repeated to ascertain that error has been rectified correctly.

Controllability measures should be designed in such a manner that these are detachable as far as possible, so that memory overheads can be reduced to the minimum during the post-release operation of the software. However, whatever controllability measures cannot be detached, should remain as part of the software. These would cause only memory overheads and not execution overheads, as these would normally be in deactivated state. During pre-release period, all the controllability measures should remain embedded, as frequent fault diagnosis would be required to be done.

5.2.2 Modifications Proposed in Software Development Paradigm

From the above discussion, it is clear that more testable software can be produced, if test-plans are also made available to the design phase. An objective, that necessary provisions have to be made to facilitate execution of these tests, is also given to the design phase.

The above implies that specification verification tests, which are prepared after the specification have been finalized, will have to be prepared before the design phase and not concurrently with it. Instead of the tests being made available during testing phase, these tests will have to be made available as additional inputs to the design phase itself.

After the design of the software has been done (keeping in view the testability of the specification verification tests), design verification tests should be prepared. The design should now be reviewed and modified to ensure easy execution of these design verification tests. The design review and design verification tests, will have to iterate, till the design and the design verification tests converge. Similarly, the detailed design should also ensure easy execution of tests based on detailed design (henceforth referred to as detail design verification tests) through a similar process. The coding phase also defines code verification tests for unit testing. The code should also be reviewed and modified suitably to ensure testability of code verification tests.

During the design, detailed design and coding phases, all the test-plans (i.e., specification verification test-plans, design and detailed design verification test-plans, and code verification unit test-plans) should be augmented with necessary procedures to be followed for conducting the various tests in terms of user interface commands and controllability measures.

If the procedure proposed above, is followed, then during testing phase, no extra activities, in terms of writing of temporary interfaces for testing of internal modules or coding for any other type of adhoc measure, would be required to be done. In all the testing activities (i.e., unit testing, integration testing and system testing), testing would simply be performed using all the test-plans and the procedural descriptions for each of the tests.

As it is expected that residual bugs may exist, even though fresh bugs are not likely to occur (unlike in physical systems which wear out with use), all the controllability measures will have to remain as an integral part of the total system (even though some of it may be in detached form) with all the necessary documentation, ready to be used, whenever it is required.

Controllability measures built to facilitate testing, are also useful for the corrective maintenance phase, as it also requires tests to be conducted for purposes of fault diagnosis. During maintenance phase, controllability measures will help in creating suspected states, required to conduct various tests to isolate faults.

5.3 Controllability and Testing

As controllability (as well as observability) gets designed and built during the design and coding phases, the various test-plans (i.e., specification, design, detail design and coding verification test-plans) get augmented with procedures to be followed in terms of various user and control interface commands.

Most of the tests can be executed using the commands available in the user interface. Before the tests are executed, appropriate observability measures will have to be activated. During test execution, active probe messages will get recorded in event history file. The external behaviour of the software and internal behaviour recorded in event history file, can be analysed for detecting faults.

For difficult tests, controllability measures would be required to be used. Environments required by tests can be created using the "set environment" commands; and desired state can be set using "set state" commands. Required observability measures can also be activated before test execution. Test is executed by providing inputs through appropriate user or control interface commands. External as well as internal behaviour can be observed and analyzed for detecting faults.

For a complex software having several modules, it may be possible that some of the modules are at the internal level and may not have any user interface associated with them. Such modules are also provided with an appropriate interface for testing as part of the controllability measures. Such an interface could be at module level, sub-module level or even at some sub-system level depending on the software.

Thus, the various phases of testing (i.e., unit testing at module level, integration testing at sub-system level or system testing) make use of the appropriate part of the user interface and the controllability measures (and observability measures) through control interface.

No testing is required to be done using adhoc methods, as provision for executing all the tests has already been made during the earlier phases. Testability required in terms of observability and controllability is "pre-built" in software before it is subjected to testing during testing phase. As testability is an integral part of the delivered software, all the tests performed during the unit testing, integration testing, and system testing, can be done later even during the maintenance phase required for purposes of fault diagnosis.

5.4 Controllability and Fault Diagnosis

Controllability can help in the process of fault diagnosis during the testing, implementation and maintenance phases. Observability built in the software can help in identifying the state of the system in which the fault has possibly occurred. If the fault is intermittent, it is due to some state which occurs very rarely during the execution of the system. Such suspected states can be created using the built-in controllability measures and relevant tests can be conducted for diagnosis purposes.

If any specific module is doubted to be faulty, it can be tested in isolation, by setting the required test environment and providing direct inputs through the appropriate commands available at the control interface. This is possible as module, sub-module and sub-system testing which is normally done through temporarily constructed interfaces, is now done through such interfaces integrated properly in the 'control interface'.

For more a detailed discussion on fault diagnosis, please refer to Chapter 6 on Corrective Maintenance.

5.5 Examples of Controllability

This section presents some examples to illustrate the need of controllability measures in different problem areas and their possible solutions. It addresses controllability problems related to table handling, filing system of an operating system, concurrent systems like b-tree, and message based distributed systems. Mechanisms are also suggested to build the desired controllability. A more detailed example from TIFACLINE host "Communication" module is also presented, for which some controllability measures were built and used.

5.5.1 Table Handling

Many a times, tables are required to be maintained in software for various purposes. The software has provisions to handle the cases when the table is full or it overflows. If the table is large, then getting it filled by actual data will be very difficult. Therefore, some measures are required to be built so that such type of boundary conditions can be tested.

To handle such boundary conditions, provisions can be made to fill the table artificially with dummy data through a program to create nearly full boundary condition. However, the table handling software will have to take care of these dummy entries properly (i.e., ignore these for reference purposes, etc.). Some extra code will have to be introduced in the table handling software for this purpose. Some extra code will also be required to fill the desired number of dummy values in the table on specific request by the tester.

Building such a mechanism through the proper phases of design and coding is quite simple. If it is not designed and pre-built in the software a-priori, then even this simple problem of creating boundary conditions and overflow condition for tables may be difficult during the testing phase.

5.5.2 Filing System of an Operating System

Filing system of an operating system, has many provisions to take care of various situations like index table overflow, bad sector occurrence, etc. It is also designed to deliver certain level of performance. Using the available filing system commands, it will be very cumbersome to test such features.

Index table overflow condition can be created by the method suggested in the previous section. For simulating bad sector, the code for disk I/O will have to be modified so that it can artificially report bad sectors. The extra code inserted for this purpose could make use of the some control information set by a control command, to guide it as to when to report a bad sector. Thus even though a sector may be readable and writable but, as it has been reported as "bad", appropriate follow-up action will get executed, which may mark the sector as "bad" and so on. Similarly, read and write failure conditions will have to be created so that the corresponding error handling code can be tested.

Similarly, a "control" module can be developed which may create files of various sizes and numbers to fill the disk space. Files may be deleted and created again. By exercising the filing system in this manner for some time, typical fragmentation of disk space will take place and performance of the filing system can then be measured.

Thus the controllability required to create various conditions for which the filing system has been programmed, can be created without much difficulty.

5.5.3 Concurrent Systems : B-Tree

B-tree used in RDBMS for maintaining indices of tables, has to handle read accesses and updations by multiple users concurrently in such a manner that waiting time for individual users can be reduced as much as possible. To provide concurrent access to such data structures in read mode does not pose any problem. Each user process can proceed as if no other user existed. If updates are also permitted, then access to the B-tree has to be synchronized in such a fashion that concurrent operations do not lead to inconsistent view or incorrect updation of the tree.

To handle such problems, special algorithms are designed and used. The code written will have to be tested simulating all the conditions for which the algorithm has made provisions. Some of these conditions may be very difficult to achieve, if multiple processes simply execute the developed B-tree software. The probability that such special conditions will occur by themselves is very remote.

It would require some extra measures to be built in the software. A possible measure to help in creating a desired combination of sequences of B-tree operation by multiple processes, would be to have a stepping control over the execution of each process. The step size would depend on the particular synchronization algorithm. Such a provision can make it possible to create any combination of sequences of execution by multiple processes.

All synchronization problems like concurrent B-tree, have to be tested for various important combinations of sequences of multi-process execution. Control of execution, in terms of the relevant granularity, can be pre-built as controllability measures. A "control flag" could be used by the software to distinguish the controlled execution from the normal execution. If the control flag is "set", then the software could check it at appropriate predetermined points in the software and stop for a go ahead signal from the user for execution, till the next break point is encountered. The control flag could be "set" or "reset" through a control interface command.

Similarly, software with built-in fault recovery also has to be tested for system failures at different points in execution. A similar "control" flag based controllability measure can be used to bring the software to a desired point in execution, and then system can be made to fail artificially, so that the corresponding fault recovery logic can be tested.

5.5.4 Messages based Distributed System

In a distributed system like TIFACLINE, various processes interact with each other using the message based inter-process communication mechanism.

Each process has to take care of various types of responses, which it may get in return for messages it sends to other processes. For example, Help module requests for an open connection. In return, it may get a message containing - "connection opened", or "desired resource not available". Sometimes, it may not get any response at all due to some problem; or it may even get a wrong message on its queue. The first two responses can be created by facilities available at user interface level (i.e., the first response message by having a free Helpdesk and second response message by not having a free Helpdesk). But the last two are abnormal conditions, which may occur very rarely. Even then, these have to be taken care of in the software and tested.

A possible controllability measure, which can facilitate creation of such rare conditions, could be that each process has provision to interact with the tester so that he can choose the desired response. A "control flag" could be used to distinguish controlled execution from normal execution of the software. The "control flag" could be "set" by a command available in the control interface. Such a simple extension of the software can make it possible to create an otherwise very rare condition.

Such a mechanism can also be used to take care of interactions between all the processes in the TIFACLINE software. However, where selection of response by the tester, is likely to cause unacceptable delay and change the software behaviour, this response selection can be done based on pre-defined, more detailed control information.

5.5.5 Communication Module of TIFACLINE HOST Project :

A Detailed Example

To illustrate the concept of controllability presented above, we will take an example of Communication module of TIFACLINE HOST project, in which some of these concepts were experimented with, and has already been described in Chapter 3, Section 3.5.1.

The Communication module handles the transfer of messages meant for other host machines. It also receives messages from other hosts and distributes them to various processes in the local machine. It maintains a list of connected host machines and dial-up numbers for establishing connections.

Messages are collected in various files meant for different hosts. Periodically, files are closed and file transfer is initiated. Only a controlled load is sent for transfer, to facilitate each side in getting frequent chances to transfer information. If no message is pending for transfer, a dummy load is sent in order to keep the connection live, and keep it ready for sending any message later on. However, if both sides have nothing to send for a specified period, then the connection is broken.

Two message priority levels are supported. Control messages that are used for establishing connections are short and require fast turn around time. These are sent on high priority. Detailed data records, which are large in size, are sent in low priority. The messages for these two priority levels, are collected in separate files; and a file is closed, whenever it exceeds the specified number of messages, file size, or time out period. It also keeps a count of total number of bytes pending for transfer to each site. When this count exceeds a threshold, a message is sent to Handshake module to indicate communication line overload condition, which, then, disables opening of new connections to that particular host. When the pending load drops below another lower threshold value, another message is sent to Handshake module to enable opening of

fresh connections. If a line breaks, then a retry is made after a time period. After repeated trial, the line is declared "dead" and Handshake module is informed accordingly.

5.5.5.1 Controllability in Communication Module

For proper testing of the Communication Module, it is important to create all the conditions it has been designed to take care of. Using the external user interface controls, it will be difficult to create many of such situations easily (e.g., overload and under load conditions, mix of high and low priority messages, faulty line or host-down conditions).

A small control module was developed which could produce dummy messages of various sizes and priorities. The disassemble communication sub-module, on recognizing dummy messages, will simply "return" the message or "create" several messages of required sizes to simulate fetching of data records. It could also send a message to communication module to activate or deactivate a host line to create line fault or host down situations.

Sufficient observability was also built in using the probe mechanism to observe the internal working of the software for fault diagnosis. The timing information attached to each probe message proved very vital, in understanding the various time delays. As the initial design did not deliver the desired message turn-around-time, a new design was worked out, after some experimentation with the first released module. Providing controllability required additional functionality to be built in communication module. An extra module was also developed to drive the communication module in the desired manner.

Controllability could also have been built for other modules of software. For example, each module has provision for handling time-out conditions, ignoring wrong messages, etc. If required controllability was built to create such situations, then the task of fault diagnosis for these modules would have been simpler.

5.5.6 Conclusions from Examples

From the above examples, we have seen that software systems cannot be tested very thoroughly if one is solely dependent on user interface for creation of various test conditions. Generally, the software is designed to cater to many special conditions, exceptional conditions and error conditions, which are difficult to create using the user interface commands. Testing would be incomplete without testing these conditions. Appropriate mechanisms were also proposed and discussed to handle such conditions.

Awareness of the need for planning for such difficult to do tests is very important. Appropriate mechanism can be found without much difficulty. However, this exercise should be done during the design phase itself, as these mechanisms are not add-ons, but have to be built intrinsically in the design and coding itself. If these are thought, afterwards during the testing phase (by that time the design and code are already frozen), incorporating such mechanisms would require changing the design as well as the code, and this would lead to distortion of the structure of the software. These changes would not only affect the quality of the software but also be difficult to carry beyond the testing phase as an integral part of the delivered software.

Therefore, testability issues should be considered during the design and coding phases, so that appropriate controllability mechanisms can be devised and built into the software as an integral part.

5.6 Summary

The important points of this Chapter can be summarized as follows

:-

- Controllability measures has been defined as provisions in the software to facilitate execution of the tests requiring

states of software, which are difficult to achieve through user interface commands. Special conditions, error or boundary conditions are some of the typical examples of states which are difficult to create.

- Building controllability requires modification of life cycle model. Test-plans based on specification will have to be made available to the design phase, alongwith an additional design criteria, that the design has to make adequate provisions for the easy execution of the tests, which, otherwise, would be difficult to execute.
- Design should also be reviewed and modified to ensure easy execution of the design-verification tests defined at the end of the design activity. The design and design-verification tests will have to iterate, till the above condition is satisfied. Similarly the detailed design should also ensure easy execution of detailed design verification-tests through a similar process.
- During the design and coding phases, specification verification tests, design verification tests, detailed design verification tests, and code verification tests are augmented with procedures to be followed for conducting the various tests in terms of user and control interface commands.
- Controllability measures are required to be built to
 - (a) Set Test Environment
 - (b) Set Test State
 - (c) Activate Observability Measures
 - (d) Provide Test Input
 - (e) Perform Test Analysis
- 'Set test environment' isolates the component required for testing. The component can be a module, sub-module or a sub-system. The required embedded controllability is also

activated (using control flags). For setting state and providing test input, some additional software (even module) may have to be built, which can be activated using the control interface commands.

- To illustrate the concept, controllability measures have been discussed for creation of various difficult test conditions for concurrency, fault recovery, tables handling, and message based distributed system. From these examples, it is clear that once the objective of building controllability is set, it can be achieved without much difficulty.
- Controllability measures and the associated documentation form an integral part of the delivered system. However, some of the controllability measures may be detachable, which can be attached whenever required.
- Software with built-in controllability measures would not require any adhoc measures to be developed, during the testing phase, as software would have adequate provisions to execute all the test easily.
- Controllability measures built to facilitate testing are also useful for the corrective maintenance phase, as it requires tests to be conducted for purposes of fault diagnosis. Controllability measures will help in creating the suspected states required to conduct the various tests to isolate the fault.
- Building controllability measures is a creative design activity. Only with increasing experience in building and using controllability, some general principles will emerge.

END OF CHAPTER 5

6.1 Software Maintenance

Activities of the software maintenance phase have been classified as corrective, perfective and adaptive [Gilb88, Pres92, Schn87]. Invariably, the engineers engaged in software maintenance are different from those, who are involved in the design and the development of the software. Maintenance engineers have to understand the software prior to undertaking any maintenance activity. For large software, these tasks become harder to cope with, due to increase in size as well as the complexity. It has been established that more than 50% of the total resources, which are spent in the life cycle of a large software, are spent in the maintenance phase [Pres92, Schn87]. As more and more software for large applications are being developed and put into use, it is consuming greater share of human resources engaged in software development.

Corrective maintenance is the minimum essential activity that is required for any released and in-use software. Many times, absence of modularity and structured code, coupled with the inadequate documentation of the software, make the tasks of corrective maintenance more difficult. For ill-documented software, a lot of research has been focused on the extraction of design and other documents from the source code of software [Bigg89, Chik90, Choi90, Oman90b]. Book paradigm for program documentation has been shown to be very effective for use by the software maintenance engineers [Oman90a].

Maintenance effort has been shown to be related to software structure [Gibs89, Gill91]. Software complexity metrics can also be used to identify poorly structured component of the software so that it can be restructured to reduce maintenance efforts [Harr82, Kafu87]. Volume of changes during maintenance can be measured, using software science for more effective resource allocation and restructuring of modules [Harr90].

Even if we assume modular design, structural coding, acceptable levels of code complexity and adequate documentation, a close look at corrective maintenance will reveal that diagnosis of any software fault is not easy and it involves certain distinct activities on the part of the maintenance engineer. Furthermore, the problems of corrective maintenance are never kept in mind, while designing and developing the software.

After release of the software, when the reports of faults start pouring in, corrective maintenance activity emerges out of the blue. How an engineer should approach to solve a software fault, reported after its release, is never specified either by the designer or by the developer of the software. In current practice, the approach towards corrective maintenance is largely "ad-hoc" and "intuitive" in nature.

In this Chapter, the various components of corrective maintenance activities are analyzed, and a systematic approach to corrective maintenance is proposed. Again, the systematic approach is applicable only when appropriate measures (viz., observability and controllability) have been taken a-priori in the design and coding phases of the software development.

Observability measures are permanently embedded statements called probes, planted at appropriate places in the software (refer Chapter 3 Section 3.2). These probes, which are of various types and levels, can be activated, selectively and dynamically producing the execution-traces of desired modules at desired level of details.

Diagnosis of software faults can be performed more effectively, if adequate observability (macros as well as micro level) is in-built in the software. The macro level observability can aid in isolating the faulty software module, while the micro level observability will aid in concretizing the fault to a specific function or a procedure, which can then be rectified.

Controllability measures are commands in addition to those provided to the users (refer Chapter 5 Section 5.2.1). These additional commands are incorporated at the time of software design, to help maintenance engineers to reach those states of software that are unreachable (or very cumbersome to reach), through external user interface. The prior-built controllability measures can aid in confirmation of intermittent faults, isolating the faulty module, and also aid in fault concretization.

The observability and controllability measures have to be thought of well in advance. These have to be conceptualized and concretized, when modular architecture of the whole software is being frozen (i.e., in the design phase). The freezing of micro level observability measures may get delayed, till the detailed design or even the coding phase.

A large software developed following the proposed approach will have an associated document "Guidelines for Software Maintenance", that should carry information about all observability and controllability measures provided in the software and prescribe an approach to be taken for fault diagnosis, when any fault is reported.

6.2 Corrective Maintenance

Corrective maintenance comprises of following activities: fault confirmation, fault isolation, fault concretization, fault rectification and testing. Once a software fault is reported, the first activity is to "confirm" that the reported fault is in the software and not a reflection of some hardware faults or underlying system software faults. The task of fault confirmation is easy if the reported fault can be regenerated; and is difficult if the fault is of intermittent nature. After confirmation, the fault needs to be isolated to the specific component (module) of the software which contains the fault. On fault isolation, the faulty software component is scrutinized, in detail, (called "fault

concretization") to zero-in to the piece of code that has been implemented incorrectly. The fault rectification activity not only removes the fault, but also sees to it that no new fault is introduced in this process. After fault rectification, the new version of the software has to go thorough testing before it is released.

For a small size software, fault isolation and fault concretization may be one and the same, but for large software, due to their increased complexity, it is extremely hard to concretize the fault in one step. Again, the familiarity with the design of the software is essential for fault isolation and it is precisely this reason that many researchers are focusing their attentions in extracting design information from the source code of the software, which has inadequate design documentation [Bigg89, Choi90, Oman90b].

6.2.1 Fault Diagnosis

The fault confirmation, the fault isolation, and the fault concretization activities of corrective maintenance are collectively called fault diagnosis activities. The efforts required for a fault diagnosis cannot be visualized a-priori, since there does not exist any generic approach, which can be systematically applied to any large software. In contrast, efforts required for fault rectification and testing, once the fault is clearly diagnosed, can be estimated to sufficient accuracy and its volume is relatively smaller than that required for fault diagnosis.

In short, the software diagnosis efforts account for larger share of corrective maintenance and is also of unpredictable volume. Since there is not much room to shorten the fault rectification and testing activities, the corrective maintenance activities can be reduced, only if, we can reduce the efforts required for fault diagnosis.

6.2.2 Improving Fault Diagnosibility

We assert that, if fault diagnosibility is to be improved, we have to take measures in the design and development phases of the software development itself. The two measures that we propose to improve fault diagnosibility, are in-built observability and controllability in the software. The observability measures will aid in fault isolation and fault concretization, while the controllability measures will aid in all the three activities of fault diagnosis.

6.3 Observability for Fault Diagnosis

Ability to observe the internal behavior of a software is very essential for corrective maintenance. Limited observability is available in all softwares at user interface level, in terms of the output response generated as a result of some input. This level of observability is not enough for purposes of fault diagnosis. Generally, an input may not only produce an output but it may also update the internal state of the software in terms of various internal tables, variables, etc. If some of these updations are not done properly, then it may not be immediately reflected in the output, but may affect subsequent behavior of the software. It is for purposes of observing these changes in internal state of the software, that observability is very essential.

The probe mechanism proposed in Chapter 3, Section 3.2 and also reported earlier in [Gupt91, Gupt92], is best suited for recording various events occurring in the software. Probes can be embedded anywhere in the software by the developers. By placing probes at appropriate places in the software, all the information required for fault diagnosis can be gathered. We propose the utilization of probe mechanism as observability measure for the corrective maintenance of the software.

6.3.1 Probe Mechanism & Stepwise Focusing for Fault Diagnosis

Examples in this Section are being taken from the Communication Module of TIFACLINE Host Software [Tifa90a, Tifa90b], which has already been described in detail in Chapter 5 Section 5.5.5. In brief, this module assembles messages from various processes into files for various destination hosts. Similarly, messages received in files are de-assembled and sent to queues of various local processes.

Probe Identifier structure can reflect the various levels of abstraction. The probe messages should be constructed such that all the information considered important is available for fault diagnosis. For example,

Probe_id :

MAC/Comm.Send/Assemble.5.1
MIC/Comm.Rece/Dissemble.3.2.1

Probe_message :

Msg_to_send From_id : 25 Pri : Low,
Size : 45 Dest_host : 5 To_id : 35

Msg_recd Size : 50 Process_id : 25

Probe_ids given in the example above represent a macro level (represented by string 'MAC') probe name, Assemble.5.1, in "Send" sub-module and a micro level (represented by string 'MIC') probe name, Dissemble.3.2.1, in "Receive" sub-module of Communication Module (represented by string 'Comm') respectively. The messages contain information, considered important for each packet, that is being sent out or received.

Probes can be activated or deactivated during execution. A deactivated probe does not write anything in the log. As probe_id is structured, several probes can be grouped and addressed by a single command, e.g.,

```
Enable      MAC/Comm.*/*
Disable    MIC/Comm.Rece/*
```

Enable command given in the example, activates all macro level probes in all the sub-modules and procedures of Communication Module. The second command deactivates all micro level probes in all procedures of "Receive" sub-module of the Communication Module. Though we have given examples of only two levels (viz., macro and micro levels), there can also be multiple levels. How many levels of probes should be embedded in the software, depends on the modular architecture of the software, and the step-wise focusing approach that the designer will like to provide to the maintenance engineer for the fault diagnosis.

A maintenance engineer, while running the software for fault diagnosis, can also put break points on selected probes. On occurrence of a break point, control is given to the console. For example,

```
Break on  MAC/Comm.Send/Assemble.5.1
```

This statement will bring back the control to the console when the execution encounters probe name, "Assemble.5.1", in "Send" sub-module of Communication module.

The maintenance engineer can then activate or deactivate other probes, put additional break points, or query the event history file in which all the messages generated by active probes are recorded. It is to be noted that breakpoint can be put only on pre-embedded probes and not on any statement chosen on the fly. This characteristic distinguishes it from the usual debuggers.

Query can be made using a query language. For example

```
Select * where probe_id = "MAC/Comm.*/*" and  
      Pri : High and Dest_host = 5
```

The select query will fetch all entries from event history file containing 'Pri : High' and 'Dest_host = 5' (i.e., all high priority messages sent to destination host 5).

6.3.2 Fault Diagnosis : Responsibilities of Software Designer and Developers

In our proposed scheme, any released software must carry many permanently embedded probes, which will be helpful for fault diagnosis in the corrective maintenance phase. While designing a large software, the designer must keep, in view, the problems of the maintenance engineer. On finalization of overall architecture of the software, the designer must advise (the programmer) to put appropriate probes (macro level) at proper locations so that fault isolation can be done easily by the maintenance engineer following step-wise focusing approach.

Similarly, in detailed design and coding phases, the developers should also keep the issue of corrective maintenance in mind, and should put some additional probes (micro level) so that fault concretization becomes easier for the maintenance engineers.

Probes, at both the macro as well as micro level, can be controlled selectively, using generic addressing possible, due to the structure of the probe_id. This provides a very powerful way to focus attention on any desired part of the software to the required degree of details. When a software is released for use, all embedded probes must be in disabled state.

Probes are like simple statements, and hence, embedding probes will not introduce any logical bug in the program. Since probes will consume resources, both designers and developers are cautioned against excessive probing. The set of embedded probes must satisfy necessary and sufficient conditions for fault diagnosis.

A software that has been designed and developed, following this approach, will have an associated document that gives information about all the embedded probes.

6.3.3 Approach for Fault Diagnosis

Enhanced observability makes fault isolation and concretization easier. Maintenance engineers are advised to follow step-wise focusing approach for fault concretization.

On fault confirmation, it is advised to enable only macro level probes to start with. Information captured by these probes can be analyzed using the query language, and faulty module of the software can be isolated. In the next step, only the micro level probes of the faulty module can be enabled to pinpoint the fault to code level.

The document giving information about probes must also contain an approach for fault diagnosis, such as, the successive steps for enabling various levels of probes and how to analyze the various types of logs. Since all softwares have their own specificity, each software will have its own document for maintenance.

6.4 Controllability for Fault Diagnosis

Any software system is operated by the user through the provided user interface. All the functionality built into the system is available through this user interface. However, there may be certain states of the software, which may not be readily achievable through the commands provided at user interface level. Most of the large software have internal states (values stored in internal data structures of the software represent its state) and hence, same input may produce different outputs at different occasions.

In case, a reported fault is of intermittent nature, one reason of such a behavior might be that it is a state dependent fault. Rarer the state, less frequent is the occurrence of fault. Fault confirmation of an intermittent fault is hard, since the rare state may not be easily recreated through user interface commands.

Many large softwares are composed of multiple processes internally and it is very difficult to determine the state at any given time. The indeterminacy of the state stems from independent execution of its various processes. In case, the intermittent fault is related to the temporal characteristics of processes, it will be almost impossible to confirm the intermittent fault only by external user-interface [Cheu90, McDo89].

For software having temporal characteristics and data-dependent state, it is proposed that additional controllability measures to set each software module to desired state must be provided. These controllability measures will help the maintenance engineer to confirm and concretize the fault easily.

Again, for software having multiple components, where each component has relative independence, additional controllability measures would be required, for maintenance engineer to scrutinize some of them in isolation.

To summarize, controllability measures are required for activities of fault diagnosis.

6.4.1 Controllability Measures and Fault Diagnosis

Controllability measures proposed in this thesis are totally transparent to users and are made accessible through a separate interface termed as 'control interface. Controllability measures may remain dormant in normal execution runs, but are activated only by maintenance engineer for fault diagnosis. On activation, the maintenance engineer applies the controllability measures through the control interface.

Confirmation of intermittent fault can be easier with the help of additionally provided controllability measures. If a fault is doubted due to specific state, the desired state can be created with the help of additional commands before running the software. Similarly, if a specific process is doubted to be faulty, it can be tested in isolation by providing input, not from other processes, but from additionally provided commands through control interface.

Controllability measures built in the software may exist as an integral part of software as well as independent modules. The parts attached to the software, intrinsically, may be activated by setting some control parameters through the control interface command. Independent control modules are entirely executed through the control interface commands. In the released software, some of the former type of measures need not be linked with the software to reduce memory overheads. When a fault is reported, these measures can be linked again for purposes of fault diagnosis. Independent control modules don't cause any memory overhead in normal operations of the software, as these get invoked only when required during fault diagnosis.

We want to assert that for large software, provision of additional controllability measures are crucial to reduce the ever increasing corrective maintenance cost.

6.5 Corrective Maintenance & Observability and Controllability Measures

Observability and Controllability measures built in software make fault diagnosis in corrective maintenance far more easier. Fault diagnosis is manpower intensive and consume lot of resources. Hence, proper provisions for observability and controllability in software is highly desirable, even if it incurs some overhead. Some level of observability and controllability can be left permanently in software, and the rest can be attached or detached, as and when required, for fault diagnosis.

For generic hardware systems, observability and controllability are extensively built for fault diagnosis. All internal states are made directly or indirectly observable. Hardware also have provision to set desired states at various points internal to the system. Hardware fault diagnosis is entirely dependent on in-built high level of observability and controllability.

Experience of building and using observability and controllability in large distributed project like TIFACLINE has shown very effective results. The fact that such concepts are also being widely used in hardware testing and maintenance, should encourage software engineers to systematically provide for these concepts in software, rather than depend on brute force adhoc methods at the time of crisis.

To assist the maintenance engineer in fault diagnosis, a manual containing all the necessary information related to observability and controllability built in the system should be provided. It should contain a list of all the macro and micro level probe information and important standard queries, which can be made on the event history files. Control commands and parameters, control

modules and processes, and the context in which they can be used, should also be documented. Description of what is in-built and what is attachable, and the procedure to attach and detach, should also be described. In brief, this document should give "Guidelines for Software Maintenance".

Observability and controllability measures cause overheads in terms of memory requirements and execution time processing. Even though such measures can be built, such that, most of these measures are detached during normal operations of the software and attached when needed for fault diagnosis. Some measures, however, should be present all the time so that preliminary fault analysis can be done with the system in operation and then, suitable component of controllability and observability measures can be plugged in, for more detailed diagnosis. Several optimizations proposed in Chapter 7 can help in reducing the execution time overheads. Overheads, however, cannot be eliminated totally. In real-time system, these measures are also very relevant (refer Chapter 7 Section 7.7). As real-time applications are highly time-critical, such measures should be used very judiciously.

6.6 Summary

The important points of this Chapter can be summarized as follows:-

- Activities of corrective maintenance are composed of fault confirmation, fault isolation, fault concretization, fault rectification and testing. The first three activities, collectively called "fault diagnosis", are indeterministic, and accounts for major share of corrective maintenance.
- Efforts required for fault diagnosis can be reduced, if issues of fault diagnosis are considered in design and coding phases of the software development itself.

- Probe mechanism can provide the desired level of observability in the software required for fault diagnosis.
- Controllability is required to make the software or a component of it, to reach a state, not readily reachable by user interface.
- Software designed and developed with observability and controllability measures, will show improved corrective maintainability.
- 'Guidelines for Software Maintenance' describing all the controllability and observability measures built in the software, can greatly simplify the fault diagnosis task performed by the maintenance engineer.
- Observability and controllability measures load the software, and hence they should be used judiciously. The concept may not be readily usable for time-critical applications.

END OF CHAPTER 6

7.1 Testability Overheads & Optimization Considerations

Observability as well as controllability measures built in software will add overheads, in terms of execution load as well as the memory space. To reduce these overheads to whatever extent possible, several approaches have been worked out and described in the following sections.

7.1.1 Attachable & Detachable Observability & Controllability

We have proposed that observability and controllability measures should be embedded permanently in the code. Furthermore, we have classified these measures as design level, unit level, and maintenance level. In order to reduce the overhead of these measures (once the software has been thoroughly tested), the design level and unit level observability and controllability measures may be detached (removed) from the code, before releasing the software for use. Detachment of observability and controllability measures can be achieved, either through a compile time switch, or through a specially designed pre-processor for the source code which can recognize the observability and controllability measures and removes them from the code.

In case, the detached observability and controllability measures are needed later in the software for fault diagnosis, a version with these measures can be re-created. On-line fault diagnosis is not possible for software version from which some of its observability and controllability measures have been detached.

Detachment of some observability and controllability measures reduces processing as well as memory overheads.

Each probe call is also given a unique number by a pre-processor, so that status processing of probes can be expedited. A table having two bits for each probe, called probe status table, can be maintained. Initially the state is unknown for all the probes. On encountering a probe call, if its status is found to be unknown, then it is processed through the activation / deactivation command table. The result is set in the probe status table so that, next time, it is not necessary to check against the command table, which is much more time consuming. Next time, its status can be simply read from the probe status table. This mechanism can reduce the processing overheads of deactivated probes to a very large extent.

Whenever a new activation / deactivation command is given, its entry is made in the command table. As it may change the status of any of the probes, the probe status table is reset to unknown state for all the probes, so that these are reprocessed afresh. In this approach, the status of only the encountered probes are processed.

An alternate solution to efficiently implement checking of probe activation status would be, to reset the probe status table at the beginning, and update the status of all the probes, whenever any activation and deactivation command is given. Thus, during execution, activation status of any probe can be simply read from the up-to-date probe status table. However, in this alternative, status of all the probes have to be updated, whenever any activation or deactivation command is given.

For software released for use, the required activation / deactivation commands can be put in a control file, and probe status table pre-computed and stored for ready loading at execution time. For additional commands required during the operations of the system, new probe status file can be prepared off-line and handed over for use, so that commands processing overheads do not affect the response of the system.

Probe call is implemented as a macro call. First, test mode is tested. If it is "on", then probe activation status is tested. Only if probe is an active probe, then its message is constructed and a function call is made to record the message. This procedure avoids unnecessary construction of messages for deactivated probes.

7.2 Logging of Probe Messages

Event history or log file used by probe mechanism, records messages generated only through activated probes. This log file is, later, used for analysis using a query language.

For a software having multiple processes (tasks) internally, there are two options for logging messages generated by active probes : (a) each software process has its own process-specific log file, or (b) there is only one system-wide log file. In the former case, fault diagnosis can be done, by taking into consideration, log files of all processes of the software, and hence it will be relatively difficult.

In the latter case, recording into the log file is handled by a distinct process called 'logger process'. Various processes send messages generated by the active probes to this logger process for recording. Each message is timestamped, before it is dispatched to the logger, so that correct relative execution sequence can be extracted, even if, the messages reach the logger process in any order.

In the former case of process-specific log file, there are two options for logging : (a) logging may exist as a distinct process or (b) it is linked with each of the software processes.

The separate logger process mechanism has several advantages over the latter option. Firstly, as message sending is faster than writing to a disk file, the execution overhead is less. Secondly, if any process crashes due to some internal error condition, then

even the last message generated by it, will be recorded in the process-specific log file. It makes it easier to locate the point of process crash.

In the latter option, where the logger is linked with the software process itself, messages towards the end of process crash will get lost, as these may still be in the unflushed buffers. Closing log file or flushing buffers, after every message recording, may retard the execution of the process beyond acceptable limits. Thus, a separate logger process provides a faster and more effective way to record messages generated by various processes of the software, even where process-specific log file is kept.

7.3 Display of Probe Messages

Event history or log file generated through active probes are used for the post-analysis of the software behaviour. Many a time, the designer or the software maintainer may like to view the messages as they are generated, so that they can take decision on-line, either to stop the execution instantly or let it continue further. Facilities to scroll the messages can be provided as well.

7.3.1 On-line Behaviour Window & Filter

We propose to associate a behaviour window with the log file that can be activated or deactivated selectively. On being activated, the behaviour window is mapped to a specific terminal and displays all messages logged in the log file on-line. The user may attach a filter with the window. In case, the window is attached with a filter, only those messages that satisfy the filter criterion are displayed, and the remaining messages that cannot pass through the filter, are not displayed.

The concept of single behaviour window can be extended to multiple behaviour windows. A software can be associated with multiple behaviour windows, each one attached with distinct behaviour filters. Depending upon the type of testing being done, a specific set of behaviour window can be activated for monitoring the software behaviour. This kind of testing environment is specially suited for software having multiple processes and distributed system softwares.

The concept of multiple windows can be, further, extended to create and activate a behaviour window with specific filter on-line and to modify its filter dynamically as well. In this "on-line" mode, the direction of focus as well as the depth of observability can be dynamically controlled, which can save many iterations of software execution compared to "off-line" analysis of information. In this "on-line" mode, a window can be deactivated and even destroyed dynamically.

7.3.2 Off-line Behaviour Window and Filter

The behaviour window concept, proposed for on-line monitoring of software behaviour, can be equally used for off-line analysis of log files

The probe information recorded in the log file can be replayed off-line, where complete control is provided in terms of speed of going forward, or even going back, so that software behavior can be analysed easily.

For software having multiple processes or distributed systems, independent behaviour windows may be specified for various processes. The information in various windows needs to be synchronized so that it replays the actual sequence of processing. This synchronization is achieved using the timestamped information contained in each probe message.

7.4 Query Processing

The probe message has two parts. The first part is fixed in format, i.e., it contains

```
( Probe_id,  
  Time_stamp )
```

The second part contains probe dependent information. Each value is tagged with a legend or field name. The fixed part can be loaded into a relational table and the variable part can be put in the variable length sub-record in the same record. Query can be made using a standard relational SQL on the fields of the fixed part of the record. The query language has to be extended for handling the fields contained in the variable length sub-record. Here, each field has to be referred using the tagged legend. The query processing will require scanning all the records. Each record has to be first checked for the presence of the fields referred in the query, and then process it for for the selection condition. If the fields of the fixed part appear in the query, then these can be used first to filter out the records, and then processed as described above. Suitable indices can be used to expedite processing of the fields of the fixed part.

As messages generated from the same probe contain the same structure of information, some pre-processing can be done to attach a unique record type with each such class of messages. In query processing, once a message is found relevant, as it contains the fields referred in the query, then all the records of its class can be flagged as 'relevant for query processing' so that filtering of records can be done more efficiently.

It is to be noted that efficiency in processing the query is important, but not really a stumbling block, as analysis does not demand very quick real time response. It may not be worth while to invest large effort in optimizing the query processor, as it is only a tool for software testing.

7.5 Performance Measurement using Probe Mechanism

In any software, performance is certainly an important issue. The designer has the performance at the back of his mind, while engaged in the creative process of design. His thinking is guided in the direction through which, the expected level of performance will be possible. However, by reviewing the design, one cannot predict very accurately that the system constructed with the proposed design will certainly deliver the anticipated performance. Performance is usually measured in terms of response time. It is the final outcome of many activities in the system, many of which cannot be characterized precisely. Only where margins are very large, acceptable response will result easily. However, in very demanding systems, where design is heavily influenced by performance criteria, the resulting performance will rarely be achieved in one attempt. Therefore, some analysis of the built system will be required, in order to see the effectiveness of the design. The analysis may lead to some tuning of the system to achieve the desired performance, or may lead to minor or major re-design activity.

Thus, in complex software, performance analysis is very essential for the evaluation of the design, and tuning or redesign, if required. For performance measurement, processing time taken by various components in the software is a must. An overall performance has to be comprehended, in terms of performance of various components so that the weak link can be identified for further tuning or re-design.

The probe mechanism very naturally captures the timing information through out the software, as each probe message is timestamped. This information can be used to derive performance of various components of the software. Thus, no extra effort would be required, in case, probe mechanism has been used in the software for observability.

In distributed systems, timing information is of greater importance, as it indicates how various independent processes has executed in real time in relation to each other. Using this information, replay utilities can be developed to give the illusion of actual execution. Messages from various processes can be displayed in different windows so that the interaction can be easily understood, with a view to ascertain the correctness of software behavior or diagnose a reported fault.

7.6 Assertion checking

Programs are written to transform input-data into an output which satisfies certain desirable properties. For example, a sorting program may take a set of records and convert them into an ordered sequence of records using any of the techniques for sorting. Similarly, an update of a database modifies the database in such a fashion that certain properties continue to hold true even after updation. It may require relatively very simple program to check whether the post-processing conditions hold true or not. For example, sorting may be a very complex program, but to check the correctness of sorted output may be a very simple order-checking program. Checking the correctness of an output against a certain valid assertion may be another simple way to detect possible errors in program, which is transforming the data/state. In fact, in non-scientific application, such as business applications, it is very easy to ascertain the validity of an output even by simple observation. This is because we are mentally checking for certain assertions, which the output must satisfy.

More complex assertions can be checked only through processing. For example, consistency of database is dependent on several layers of information. It has to be supported by complex database logic, which, in turn, is based on some basic functionality provided by the filing system. The cumulative result of all these factors can be checked by checking the various assertions or consistency constraints that the database has to satisfy at all times. Checking for such consistency may be easier through program.

However, executing these consistency checks may involve reading of entire database, which may be quite resource consuming, and hence, cannot be done very frequently. If assertions checking takes very little time, then it can be used not only to detect faults but also to localize the faults. If assertion checks are performed after every single action [Stuc77], and if the check fails, then fault lies obviously in the last action.

The real advantage of assertion checks will accrue where the logic is very complex [Haye86] and several factors contribute to lack of confidence in the final delivered software. For example, a concurrently accessible, fault recoverable, indexing system is quite complex. The consequences of various situations as well as complexity of program is extremely high. In such a situation, assertion checking would be of great help. Periodic assertion checking will ensure that indexing system is correct, and any inconsistency can be reported so that corrective action can be taken.

The assertion checking in above concurrent environment would require that the software has been brought to a quiescent state, i.e., non-transient state before the assertion checking is performed.

6.1 Probe Mechanism & Assertion Checking

If assertions checking is done after every single action, then any inconsistency can be obviously attributed to the last action. Thus, fault is immediately localized. However, in general, it will be impractical to do assertion checking, after every single action e.g., index checking will take very long, even though insertion of a key value may take very little time.

Probe mechanism can be helpful in fault isolation, even if assertion checking is applied periodically i.e. not after every single action. Probe mechanism can be used to log changes made to

the state (here index system) by various parts of the program. Whenever any assertion inconsistency is detected, the log of all message entries, after the last valid assertion checking, can be analyzed to locate the fault.

Thus, for situations, where assertions checking is not practical, after every simple change in the state of the software, probe mechanism makes it possible to use assertions checking effectively for fault isolation.

7.7 Application of Probe Mechanism in Real-Time Software

Since real time software has to satisfy stringent time constraints, overheads of observability and controllability measures may be difficult to be accommodated in such software. However, limited application of these may still be useful and feasible. Certain type of low level observability can also be built at hardware architectural level without causing any execution time overheads[Tsai90].

Limited observability concept is already being used in many real-time applications. For example, in power stations, critical parameters of the systems are constantly recorded in a circular recording medium, so that these are available for analysis, in case of tripping of the system, due to some malfunctioning. In applications of these kinds, after reporting of faults, the system must be rectified before it can be restarted. Re-running the system to recreate the fault (for fault diagnosis) is out of question.

Similarly in avionics, flight black-box recorder records important flight and system parameters which includes conversation of the pilots and ground staff. The black-box can survive any accident and is available for post-analysis, in case of an accident. If provisions for such limited observability are not made, causes of accidents will be almost impossible to find and fix. These observability measures have to be provided with the system to

prevent such failure in future.

In real time systems, the basic objective is to sense the various state parameters and exercise appropriate controls automatically to whatever extent it is possible. The important state parameters are also displayed at the control panel. Controls are also provided to the operator or supervisor to directly affect the functioning of various components. In computerized process control systems, sensed parameters may also be logged for later analysis.

Thus, the requirements of observability and controllability measures is very essential to real-time systems. However, all these are required for the environment, which is to be controlled. The software used, in such systems, undergo very rigorous testing and quality tests as the cost of failure is usually very large and unacceptable. As rigorous testing of real-time systems is very essential, observability and controllability measures can help in this very objective, by helping in testing of a large part of the software, where time critically is not involved. Thus Observability and controllability measures are of relevance for the testing phase at least.

For real-time systems, the system failure, even though unaffordable, may still happen for two distinct reasons. Firstly, the system may be imperfect, in the sense that the system may fail to respond appropriately to some unforeseen situation. Secondly, the software may still have some residual bugs, inspite of rigorous testing. In both situations, it is important that sufficient information is recorded for post analysis, as recreation of such faults are either impossible or too costly to afford.

Thus some minimum level of observability is highly essential in real-time systems.

8.1 Testability & Software Development Process

Testability is the intrinsic property of the software which facilitates the process of test execution and analysis. Also during the execution of the test, the internal behaviour should be observable for test analysis as well as to help in localization of discovered faults. Testability requires extra provisions in the software to execute the desired tests easily, systematically and externally at interface level (i.e., without any code modification).

Testability measures has been defined in terms of observability and controllability measures, which should be built in the software in a pre-planned manner and cannot be added on the fly (i.e., during the testing and maintenance phases). The existing software development process models do not support activities, which are essential for making the software testable.

Various activities, which need to be done to build testability as well as to utilize the built-in testability of a software for testing and maintenance purposes, have been discussed in this thesis. These aspects have already been discussed in the context of observability and controllability properties of software individually. The following sections briefly describe these activities as viewed from the various software development phases. Testability is built during the design and coding phases and used during the testing and maintenance phases.

8.1.1 Testability Building Phases

In order to built testability, specification based test-plans should be prepared and made available to the design phase. It will help in designing the software such that the various tests proposed

can be conducted easily. To achieve this, the design may have some extra provisions in terms of additional control code, commands etc. The design based test-plans prepared after the design phase is complete, should also be used to review and modify the design itself. The modification in the design should enable the software to execute these design verification tests easily. The same process should be repeated for the detailed design verification test-plans. At the end of coding, code verification test-plans are defined. The code should also be reviewed to ensure that these tests can also be executed easily. The above process will ensure that the required controllability is identified and built into the design and the code of the software being developed.

During the above process, the observability required for analysis of test-plan execution should also be identified. Probe mechanism has been shown to provide observability for observing the various aspects of the software behaviour at any desired level of abstraction. The observability built using the probe mechanism can be turned "on" and "off" as required. Observability can be built-in for different purposes. It can be for the purpose of providing essential information to the designer to help in verifying his design, or for the purpose of assisting the programmer with his unit testing and integration testing. It can also be built-in to assist software maintenance engineer in diagnosis of software faults, and to provide system administration aids for monitoring system operations, performance, and usage, etc. For all these various objectives, the types and positions of all probes in the software can be identified during the design and coding phases. Programmers should embed these probes during the coding phase.

The controllability and observability measured built into the software during the design and coding phases have to be documented. Various test plans should be augmented with procedures to be followed in terms of the various commands (user or control interface commands) for execution of the tests as well as to observe the specified internal behaviour. Extra commands for invoking the desired controllability as well as for activating

probes for required observability, and post-execution analysis are bunched together in a separate 'control interface' of the software. Use of these measures is not only just limited to the testing phase, but also have to last the entire life of the software. It implies that, when the software comes to the testing phase, everything has been planned and no ad-hoc measures are required to aid in testing.

The testability required for corrective maintenance phase should also be planned during the design and coding phases. Fault diagnosis, which is an important activity and is essential as no software can be developed totally error free, can be aided by building appropriate macro and micro level observability. Certain observability can also be built for monitoring the system performance during its operations.

Thus, the controllability and observability measures required for execution of various types of tests, during testing and maintenance phases, should be identified, built and documented during the design and coding phases of the software itself.

8.1.2 Testability Application Phases

Testability built into the software is useful during the testing as well as the maintenance phases. Testability is relevant in the maintenance phase, as it involves diagnosis as well as testing (after rectification of fault) activities.

Testing is performed initially at unit level and then at module level. Tested modules are then combined, and integration testing is performed. For all these testing activities, controllability measures help in creating the desired test condition and executing selected test runs, whereas observability measures help in observing the software behavior as well as in the diagnosis of faults discovered during this process.

Design verification can also be done by the designer by applying various tests and observing the design related internal behavior of the software using the design-category probes. The designer need not understand the code written by programmers and can do the design verification himself, independent of the programmer. Timing information available through observability measures can be used to analyze the performance of various components of the software. To improve the performance, necessary tuning of the design can be done.

Similarly, for system testing, controllability measures can be used for creating various test conditions prior to executing the corresponding test run, and then executing the test run. Observability measures facilitate localization of discovered faults to small component of the software. Various levels of observability measures build in the software and the ability to dynamically activate and deactivate the desired observability measures, help in focusing attention to the desired component of the software as well as to control the depth of observation, which may be very useful for fault diagnosis. System performance can also be analyzed, in terms of the performance of the various software components, and then, if required, necessary steps can be taken to improve the performance. Probes can also be defined to measure the coverage of built-in software functionality at design or system level.

For software maintenance, the built-in controllability and observability measures can be useful in many ways. These measures can help in fault diagnosis, monitoring the system functioning, observing the performance for tuning of various control parameters, and logging the usage of the system as well. Certain types of extension of the software can also be done, using the information accumulated by various observability measures.

Thus, software testability measures, built-in during the design and coding phases, are useful both during the testing as well as the maintenance phases. The software can be tested more thoroughly and very conveniently. The help, these measures can provide to the software maintenance engineer, is very valuable, as the software maintenance is performed by persons, who are usually not involved either in the design phase or the coding phase of the software development.

8.2 Testability & Current Testing Methods

In Chapter 2, on 'Current Testing Methods', typical testing environment was discussed which consisted of use of simple mechanism like 'print' statements, debuggers, and code coverage measurement tools. Other practices for producing more reliable software by reducing the incidence of error in each phase by reviews, inspections, walkthroughs etc., and also the role of standards, were discussed.

The concept, presented in this thesis, focused attention on the test execution activity itself. It consists of creation of test conditions / environment, and running the test and observation of the internal processing during test execution. Testability of software (i.e., ease of test execution) has been defined in terms of controllability and observability properties of software. Controllability measures are provisions to help in creation of difficult-to-reach states. Observability measures are provisions to observe internal behaviour to any desired degree of detail to ascertain correctness of test execution and help in fault diagnosis.

Print mechanism essentially provided observability, but it is adhoc in the sense that it is not pre-planned and it has to be removed after testing is over. It is introduced when faults are encountered. Commenting and uncommenting is cumbersome. Controllability, provided by debugger in terms of commands to

change contents of variables, is very rudimentary and is not at all sufficient for execution of various tests. Using debugger, contents of any variable can be examined at any breakpoint, but what variables to examine becomes more and more difficult to decide, as one goes from unit testing to system testing of large software. Debuggers are interactive and therefore, ties the tester down completely; and are useful for debugging of a fault, once it has been localized to a small part of the code (refer to Chapter 3 Section 3.9 for more detailed discussion).

Observability, as provided by probe mechanism, is planned observability, which is externally controllable as well. Moreover, one is working at software behaviour level rather than at code level. Behaviour of the software can be observed at various levels of abstractions (e.g., design level, code level, macro and micro levels etc.), which is not possible with debugger. Observability can be used by the designer himself to verify the design as implemented by the programmers, without the need to understand the code and also independent of the programmer. Debuggers cannot handle concurrent systems where temporal behaviour is important. Distributed debuggers also use the concept of event recording, but do not support the concept of various levels of abstractions and are largely dependent on sequential debugger like mechanisms for analysis during "replay" mode (refer to Chapter 3 Section 3.9 for more detail discussion).

Probe observability mechanism can also be used for more meaningful coverage measurement. Code coverage is more relevant only during unit testing. During system testing, it is not possible to achieve complete code coverage, as detailed design decisions taken at code level cannot be easily exercised at that level. During system testing, it is important to exercise the entire functionality built in the software as specified in the specification and the design documents. Probes can be embedded in the software at places where this functionality is coded. Probe coverage measurement can help knowing the extent of the functionality covered. The uncovered functionality can be easily identified and exercised through

additional testing.

Controllability concepts asserts that we need to plan how various tests will be applied. Some tests, which will be difficult to apply due to difficult to create states via user interface, have to be considered during design stage itself so that adequate provision can be made in the software. At present, test-plans are made available only during the testing phase. The test-plans should be an essential input to the design phase, if controllability of software is to be ensured.

All the observability and controllability measures, built in the software, can be made available through a separate control interface, even for the post-release maintenance phase. Though these measures cause memory and execution overheads, these can be built as attachable / detachable (pluggable) components. Some of these measures that are highly integrated with the software, can remain permanently in the software. The attachable component can be attached only for testing and fault diagnosis. These testability measures can be of great help to maintenance engineer in fault confirmation, fault isolation, fault concretization, fault rectification, and testing.

Testability, like quality, cannot be injected on the fly during the testing phase. If testability is appropriately planned and built into the software, the testing of software can be made more thorough and less time consuming, and thereby producing more reliable software. With built-in testability, the maintainability of the software also improves, reducing the time and cost of maintenance significantly.

As more and more complex softwares are being developed, the testability and maintainability of software have become more important.

8.3 Extended Summary of Research Contributions

The research contributions of this thesis can be described, briefly, as follows :-

8.3.1 Software Testability

- This research investigates the types of activities that are required to be done during software development process to produce testable software. A software is testable, if it has testability property, i.e., it can be subjected to pre-defined test-plans easily, systematically, and without following any ad-hoc measures. This research has identified distinct activities that need to be done during the design and coding phases to produce testable software.
- Testability property of a software is defined as a composition of controllability and observability properties.
- Usually, any software produced have extremely limited controllability and observability properties. Extra provisions, viz., controllability measures and observability measures, have to be ingrained in the software, for it to acquire controllability and observability properties respectively.
- Software with pre-built controllability and observability measures would not require any adhoc measures to be developed during the testing phase. All the planned tests can be executed readily, which will result in more reliable software.

8.3.2 Observability Measures

- Observability measures have been defined as provisions in the software to facilitate observation of internal state of the software at appropriate level of abstraction and at appropriate point in execution.
- The probe mechanism has been proposed as an observability measures for a software. A probe is a permanently embedded piece of code, that on being executed, records internal information of the software in an event history file, without affecting the logic of the system. Probes can be activated/deactivated through external commands, and only activated probes record information for post-analysis of execution.
- Probes can be of various categories, e.g., design verification probes, unit testing probes, corrective maintenance probes etc. Furthermore, probes for any category can be defined for capturing information at multiple levels of abstractions. Based on specification, design, and detailed design of the software, the probes (along with its category and level) have to be identified by the end of design and detailed design phases, and have to be embedded by programmers while coding. Various refinements have also been proposed to reduce execution time overheads.
- Design verification can be performed more easily using design category probes identified by the designer for implantation by the programmer. Design verification can be done by the designer independent of the programmers, as there is no need to understand the implemented code.
- For completeness of system testing, the concept of probe-coverage has been introduced. Code coverage is more meaningful for unit testing and not for system testing. During

system testing of large software, 100% code coverage is extremely hard to achieve. For large software, functional coverage is more relevant during system testing, which can be measured using the proposed probe-coverage concept more effectively. We define system testing to be complete when 100% probe-coverage has been achieved.

8.3.3 Controllability Measures

- Controllability measures have been defined as provisions in the code to bring the software to specific states which are difficult to achieve through user interface commands for the execution of state specific tests. Special conditions, error, or boundary conditions, are some of the typical examples of states, which are difficult to create.
- Building controllability require modification of the software development process. The types of controllability measures and their provisions have to be decided by the end of design phase. The specification-based test-plans are necessary input for the designer. Since design-based test-plans for design verification can be finalized, only after the completion of design phase, in our model, the design phase has to be iterated till the inclusion of all controllability measures for design-based test-plans is properly synchronized with the design itself.
- The controllability measures for specification verification tests, design verification tests, and code verification tests are incorporated in the software during the coding phase.

- Controllability measures are required to be incorporated in the software to
 - (a) Set Test Environment
 - (b) Set Test State
 - (c) Activate Observability Measures
 - (d) Provide Test Input
 - (e) Perform Test Analysis

- To test a software component in isolation, it is required to set test environment. A software component can be sub-module, module or a sub-system. For setting the state of a component and for providing test input, some additional software (even module) may have to be built which can be activated using the control interface commands.

8.3.4 Control Interface

- In addition to the user interface, the software is augmented with a control interface, that provides various commands for activating controllability measures, in such a way that the various tests are conducted easily, systematically, and without following any ad-hoc measures. The observability measures built in the software are also made available through this control interface along with necessary documentation.

- The control interface is accessible to the designer and programmers for testing purposes, and after release of software, it is made available to system administrator for monitoring purposes and the software maintenance engineer for fault diagnosis.

8.3.5 Corrective Maintenance

- A software is maintainable, if a maintenance engineer can confirm, isolate, and concretize the fault easily, systematically, and externally at user (maintenance) interface level.
- Steps involved in corrective maintenance have been analyzed and defined. Probes can also be inserted to build macro and micro level observability required for software fault diagnosis. These can also be integrated and made accessible through the proposed control interface. 'Software maintenance guide', describing all built-in maintainability measures, can be made available to the maintenance engineer.
- Controllability and observability measures improve fault diagnosibility, which reduces the effort required for fault diagnosis and rectification considerably. Thus, cost of ever increasing corrective maintenance, can be reduced significantly.

8.3.6 Performance Considerations

- Some controllability and associated observability measures can be designed as a detachable/attachable components. They can be selectively detached and attached whenever required, so that their overheads can be reduced during the operational use of the software. The attached measures become effective only when activated through control commands.

8.3.7 Test Data Generation & Test Driver Development

- An equivalence class definition based test data generation language (HUTEST), has been developed for fourth generation form-based application development environment (HUMIS). It has been shown to be useful for more extensive and automatic testing of pre-release version of an application software (Appendix A of thesis).
- For 3G programming environment, an application software structure has been proposed which can help in easy development of an application test driver (Appendix B of thesis).

END OF CHAPTER 8

9.1 Test Space for Efficient Implementation of Probe Mechanism

The attached probes linked with the executable version of the software incur memory overhead, even if not activated during the execution of the software. A probe library is also linked, implementing the probe commands and the associated message recording logic. This library is contiguous and, therefore, may not pose much overhead, as it would get swapped out if not executed. However, the code for various embedded probes will be spread through out the software, and that part of code has to be present in the place where the probe call has been made. As this code is not an integral part of the software, and is required only when activated, it may be worthwhile to have some internal architectural support to reduce this overhead.

Usually, processors are designed to have two modes (some times three) of operations : user mode and kernel mode. Some instructions can be executed only in kernel mode, and if these instructions are encountered when processor is in user mode, the execution is interrupted and control is taken to a specific routine. We propose to have an additional mode for the processor called the "test" mode. The hardware also has to provide a set of test instructions. In case the processor is in test mode, the test instructions are executed. When the processor is in any other mode, they are treated as "no-op" (no-operation). Further more, in the memory, similar to "instruction space" and "data space", there should be an additional space called "test space", which is also an instruction space for all practical purposes.

Compilation of probe commands should result into test instructions where the probe is embedded, but all the code pertaining to expansion of probe commands should go into the test space. If software is executed in test mode, on encountering the test monitor jump to test space is made with return address on stack; otherwise,

the instruction is treated as no-op (no operation). The "test-space" code should have access to all the variables, as if it is part of the main instruction space. Thus probe related code will be in "test-space" pages and which will get loaded only if required for execution; otherwise, these may get swapped out, if memory contention arises.

The provision of "test-mode" and "test-space" will reduce the execution and memory overheads to a very insignificant level. In such a hardware environment, sufficient observability may be left linked with the executable version, that can be activated, whenever needed, and without the need to recompiled the software.

9.2 Software Dashboard For Large Software Systems

Large software systems like OS, DBMS packages, communication software, distributed applications etc., are very complex in design and operation. Very little attention has been paid to provide adequate insight into their functioning during operations. Some adhoc commands are provided to see the internal states, resources used, etc. As these commands are considered extra provisions, and are not as an integral and essential part of the total system specification and design, they are generally neither comprehensive nor properly presented. This is because, the main focus of development is on providing the desired services and functionality.

If adequate attention is paid to depict the operations of such systems in graphical/non-graphical form in a systematic manner, such that all the important and relevant information is available, the user will have a better understanding and control over the system. The availability of such information will also make the maintenance of such systems easier. Graphic visualisation has been shown to be useful in specialised contexts, like behaviour of parallel programs [Heat91], link-list fault detection and diagnosis [Shim91a], and process state observation at various levels of abstractions [Mohe88].

Any complex system is composed of interacting modules/components which can be diagrammatically represented on the screen. On selection, any block should be further expandable into more detailed level blocks. Relevant internal information should be depicted in easy-to-understand character or graphical form. Thus, system operation will be easy to understand, administer, and tune to get better performance. These objectives are not difficult to achieve, once their usefulness is realized.

The display software can be built provided the required information is available from the system. Some of the required information is available in the internal state of the software. However, information about the dynamic behaviour of the software (e.g. a message sent) which is not recorded in the system state, may also be important for purposes of the display. Therefore, such display software can be built provided the dynamic behaviour is also recorded and made available.

What is basically required, is observability of the behaviour of the software, which can be in terms of the state / dynamic behavior of the software. The probe mechanism can be used very conveniently to get the observability required for building the appropriate display, termed as software dashboard. A display-category probes can be defined (coded as 'SD'), embedded, and activated permanently.

An independent module can be developed to provide the desired display facilities using the messages recorded in a separate display log file. As the display software is written independent of the main software, the complexity of the main system also does not increase.

The on-line multiple behaviour window feature, discussed in Chapter 7 Section 3.1, can be used to define a primitive software dashboard. For more powerful dashboards, general software primitives can be identified to facilitate their development. Once a library of such primitives is available, building such a software

dashboard for any software, based on the information captured by the display-category probes, would be very simple.

Thus, probe mechanism provides a very convenient framework to capture the dynamic behaviour of complex systems and facilitate building of suitable display software independent of the base software, which is very essential for better user understanding and control of the software.

9.3 Compiler Support for Observability and Controllability

The need to use debuggers arises primarily to observe the changing state of the software. However, it has one serious limitation in terms of its use. It requires the tester to have a knowledge of the control flow of the software so that appropriate break points can be put, relevant values of variables can be observed and changed, and if desired, to continue processing further.

A simple and more elegant way to achieve what largely a debugger provides, would be to extend the programming languages so that the programmer can specify in the program itself, what variables he would be interested in observing and changing dynamically for testing purposes. Variable declaration can be associated with additional attributes like `:display' `:change' etc.. A default attribute can be associated by the compiler itself. User can be provided facility to control the execution step when specific conditions are met.

Thus, no extra effort will be required for debugging the software. One will simply execute the software, observe the displayed values, modify desired variables, and control further execution. This kind of observability will be very helpful in ascertaining correct working of the software, and any problem can be quickly traced back to the code for correction.

The above may also be considered a pre-planned debugging. The effort spent in planning the "display" and "change" variables beforehand, eliminates the effort required to mentally identify the required variables and giving "display" and "change" commands repeatedly, in every debug session. However, it should also be possible to invoke the debugger, if the pre-planned view is not sufficient.

Control over layout of display, grouping of variables into various windows, etc., can be some useful extensions in this context. For this purpose, user handles over variables for display and changes, and a suitable library can be designed and provided.

Thus, basic observability required can be built-in by making necessary declarations. The facility to change the contents of observable variables (i.e., changing the state of the software) would provide some low level controllability required to exercise the various parts of the software.

END OF CHAPTER 9

Appendix A : Test Data Generation using Equivalence Classes

A.1 Test Data Generation Using Equivalence Classes

An application consists of one or more entities about which information is stored in a database. Each entity may have one or more sets of attributes. Each attribute or field has distinct characteristics. For example, it can take a value only from a valid range of data, or from a set of values, etc. The set of values, valid for an attribute, may be dependent on the values of some other attribute. Testing of an application would require that all the values valid for an attribute, as well as their meaningful combinations with valid values of other relevant attributes, should be tested.

In manual mode of testing of application software, generally a few important values and their combinations with values of other attributes get tried, due to the lack sufficient time. If the values valid for various attributes and their inter-dependencies can be specified in some well defined format, data for testing the application can be generated automatically. In this way, more extensive testing can be done without much effort.

Several attempts have been made for defining test cases in some generic forms. For application software, test cases can be expressed in terms of sequences of primitive functions available at user interface level and already defined test sequences [McCa85]. Test specification languages based on equivalence classes over input and output domains have also been evolved [Balc89, Sinh89, Ostr88].

The equivalence class concept has proved very useful in identifying important test cases. An equivalence class is defined as a set of values for an attribute, for which the same processing logic is being used in the software. For each attribute, its equivalence classes can be defined. Any one value from each of these

equivalence classes is considered sufficient for testing (for that equivalence class). As boundary values are often used in branch conditions, which is a potential source of errors, it is important to test the software for various boundary values. For this purpose, interior boundary values of the class and the exterior boundary values (i.e., values just outside the boundary of the class) are also important for testing. These outside values should also be properly handled by the software. An attribute may have a set of equivalence classes associated with it. Minimum one value from each equivalence class and all the interior and exterior boundary values of the class should be tested.

A.2 HUTEST : An Automatic testing tool under HUMIS (4GL) Application Development Framework

HUTEST (HUMIS Testing Tool) is a software testing tool, which utilizes the equivalence class concept, for HUMIS [Raja89]. HUMIS is a 4GL application software development environment. In HUMIS, specifications of various menus, submenus, forms for entry / updation of information can be defined on-line and interactively. Their definitions are kept in a database. Appropriate header files are also generated for use in the 'C' application software to be written, containing the application dependent logic. Higher level primitives are provided to do forms based input/output. Input is taken in terms of one form at a time. Field level user triggers (for field validation etc.) are defined in the application software written in 'C'.

HUTEST was developed as an add-on module to the HUMIS form-handler. Each field characteristic is specified using an equivalence class specification language. Inter-field dependencies within same form as well as across forms are also supported. It also has a facility to specify fields characteristics of output forms so that output generated could also be checked against the expected range of values. The equivalence class specifications are pre-processed to create a data base for use at the run time.

Figures A.1a & A.1b show the syntax of the test data specification language. Figure A.2 shows a sample application input and output forms. Figures A.3a & A.3b show the corresponding data definitions using the proposed test data specification language. Figure A.4 depicts the header file generated by the HUMIS system for use by the application program written in 'C' language. Figure A.5 shows the architecture of HUMIS and the associated HUTEST system. The details of HUTEST system are given in [Sinh89].

EQUIVALENCE_CLASS

CLASS

```
INTEGER <class_id> (n1, n2:n3/step1 );  
           % step1 is the increment value  
REAL <class_id> (r1, r2:n3/step1 );  
STRING <class_id> (string1, string2:string3 );
```

CLASS_SET

```
INTEGER <class_set_id> [integer_class_id, integer_class_id, ...  
REAL <class_set_id> [real_class_id, real_class_id, ...] |  
STRING <class_set_id> [string_class_id, string_class_id, ...]
```

FIELD_ASSOCIATION

```
INPUT_FORM <form_id>
```

```
FIELD_ORDER field_id, field_id, ...;
```

```
FIELD <field_id> <exp> | <cl_specs>;
```

```
FIELD <field_id> CASE <log_exp > : <exp> | <cl_specs>;  
                   <log_exp > : <exp> | <cl_specs>;  
                   <OTHERWISE> : <exp> | <cl_specs>;
```

```
END_CASE;
```

```
where <cl_specs> := (n1, n2:n3/step1 ) |  
                  (r1, r2:r3/step1 ) |  
                  (str1, str2:str3 ) |  
                  <class_id> |  
                  <class_set_id> |  
                  [class_id, class_id, ..]
```

Figure A.1a - Test Data Specification Language Syntax

OUTFORM_FORM <form_id>

FIELD <field_id> <exp> | <cl_specs>;

FIELD <field_id> CASE <log_exp > : <exp> | <cl_specs>;
<log_exp > : <exp> | <cl_specs>;
<OTHERWISE> : <exp> | <cl_specs>;

END_CASE;

SPECIFIC_SESSION <session_id>

RANDOM | BOUNDARY | EXHAUSTIVE

INPUT_FORM <form_id>

FIELD_ORDER field_id, field_id, ...;

FIELD <field_id> (<exp> <cl_specs>);

FIELD <field_id> { CASE <log_exp > : <exp> | <cl_specs>;
<log_exp > : <exp> | <cl_specs>;
<OTHERWISE>: <exp> | <cl_specs>;

END_CASE;

}

where { } contains optional clause

Figure A.1b - Test Data Specification Language Syntax

Employee Code :
Designation Code:
Total Salary :

Input
Form

Employee Code :
Designation Code:
Basic Salary :
Total Salary :

Output
Test Form

Figure A.2 - Input & Output Forms

EQUIVALENCE_CLASS

CLASS

```
INTEGER  class_code          (1:10, 101:999/10 );
STRING   class_desgn-mg     ("MG");
STRING   class-desgn_am     ("AM");
STRING   class_desgn_cl     ("CL");
INTEGER  class_basic_mg     (3000:5000/200);
INTEGER  class_basic_am     (2000:4000/100);
INTEGER  class_basic_cl     (1000:3000/50);
INTEGER  class_total_mg     (5000:7000);
INTEGER  class_total_am     (4000:6000);
INTEGER  class_total_cl     (3000:5000);
```

CLASS_SET

```
STRING  class_set_desgn [class_desgn_mg,
                        class_desgn_am, class_desgn_cl];
```

FIELD_ASSOCIATION

INPUT_FORM f1

```
FIELD f1_code  class_code;
FIELD f1_desgn class_set_desgn;
FIELD f1_basic CASE  f1_desgn = "MG" : class_basic_mg
                   f1_desgn = "AM" : class_basic_am
                   f1_desgn = "CL" : class_basic_cl
END_CASE;
```

OUTPUT_FORM f2

```
FIELD f2_code  f1_code;
FIELD f2_desgn f1_desgn;
FIELD f2_basic f1_basic;
FIELD f2_total CASE  f1_desgn = "MG" : class_total_mg
                   f1_desgn = "AM" : class_total_am
                   f1_desgn = "CL" : class_total_cl
END_CASE;
```

Figure A.3a - Test Definition File

SPECIFIC_SESSION 1

EXHAUSTIVE

INPUT_FORM f1 FIELD f1_desgn;

BOUNDARY

INPUT_FORM f1 FIELD f1_basic;

SPECIFIC_SESSION 2

EXHAUSTIVE

INPUT_FORM f1 FIELD f1_desgn class_desgn_mg;

f1 FIELD f1_basic;

SPECIFIC_SESSION 3

EXHAUSTIVE

INPUT_FORM f1 FIELD f1_desgn;

Figure A.3b - Test Definition File

```

struct F1 {  int  par1;
             char par2[3];
             int  par3;
             } f1;

struct F2 {  int  par1;
             char par2 [3];
             int  par3;
             int  par4;
             } f2;

#define F-FORM1 1
#define F-FORM2 2

#define f1_code          f1.par1;
#define f1_desgn        f1.par2;
#define f1_basic        f1.par3;

#define f2_code          f2.par1;
#define f2_desgn        f2.par2;
#define f2_basic        f2.par3;
#define f2_total        f2.par4;

```

Figure A.4 - HUMIS Generated Include File

Man Machine Interface Definition

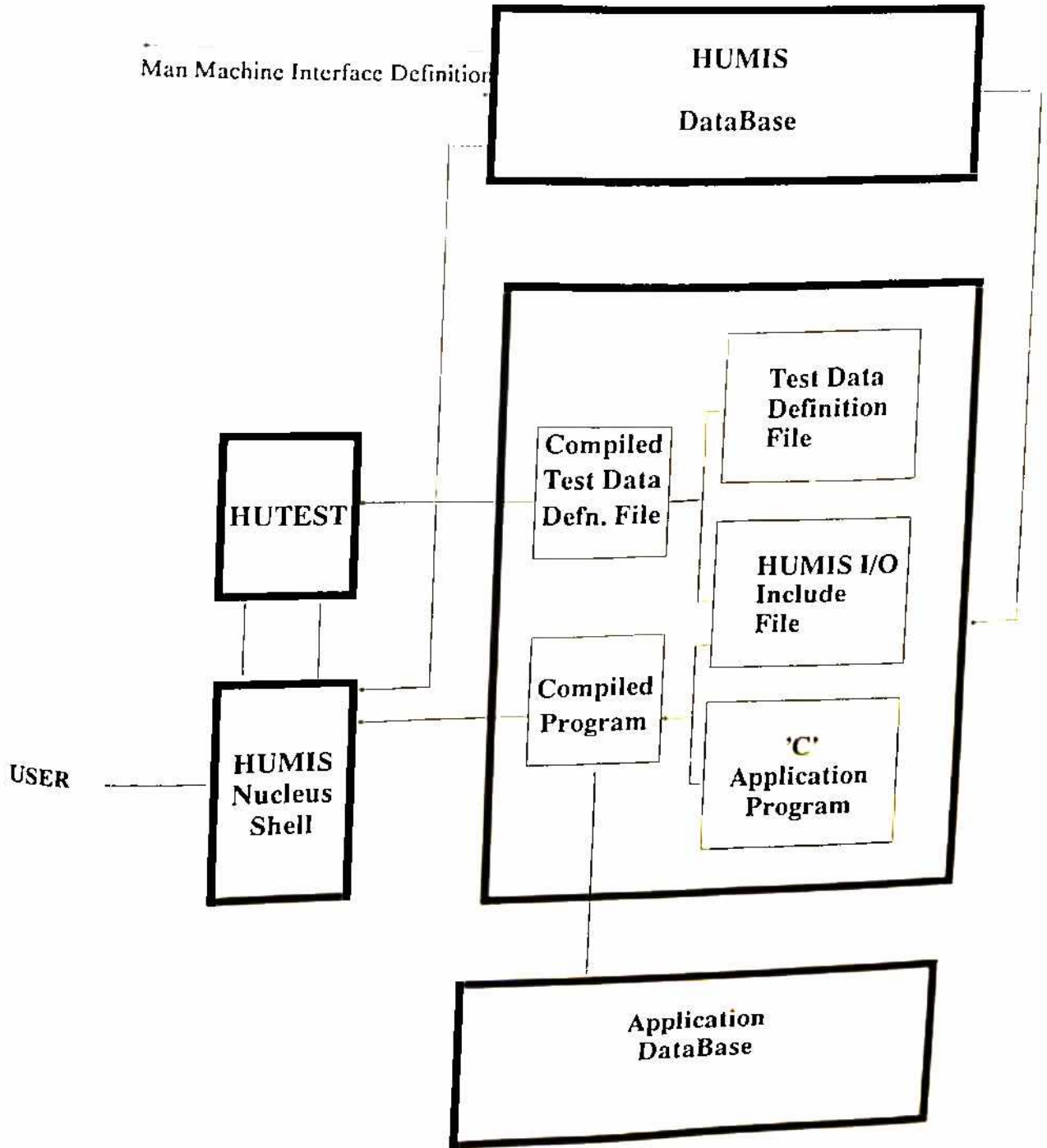


Fig A.5

A.3 Testing of Software using HUTEST

For a testing an application software developed under HUMIS environment, the input forms can be entered by the tester himself. An application software developed under HUMIS environment can be tested in one of the three modes : manual, semi-automatic, and automatic. In manual mode of testing, the tester puts the application software under a test run and fills in all fields of the input forms interactively. On display of an output form, the tester confirms the correct execution of the test, or detects a fault.

The application software can also be tested with the help of HUTEST module, in either semi-automatic mode or automatic mode. In semi-automatic mode of testing, HUTEST behaves as an aid to the tester. When an input form appears on the screen, the tester can fill in the form, or use function keys driven by HUTEST, to generate a valid value for the current field pointed by the cursor, or generate valid values for all the fields of the form. Using the equivalence class definitions specified earlier, HUTEST generates values randomly or in the order specified in the definitions. The values generated by HUTEST could also be modified by the user before processing. Forms values could also be stored for retrieval later for re-use during re-runs or post-analysis.

In automatic mode of testing, the software goes through a set of test runs under HUTEST, where values for all input forms are generated by HUTEST, on-line and without any tester's intervention. In addition, the output forms are checked against the expected values, defined earlier using equivalence classes. Tester is informed whenever any mismatch occurs either through a displayed message or through a record in a log file. Various report generation options can be tested using the data generated as a result of testing in various modes. The automatic mode of testing makes extensive testing of software easier and less time consuming.

A.4 Further Extensions

Functional Coverage : Automatic testing helps in more thorough functional testing which results in greater code coverage. Measuring functionality covered from code coverage tools will be difficult as discussed in Chapter 2 Section 2.3.3. HUTEST can be extended to keep track of the functionality covered during testing.

Extending other 4GL Tools : The concept underlying HUTEST was developed as an extension to HUMIS and can also implemented for other 4GL products, which support form-based application development tools. The form-handler of these tools can be extended to make use of the equivalence class based data definitions for data generation.

END OF APPENDIX A

Appendix B : Test Driver Development

B.1 Test Driver Development

An application software can be exercised more thoroughly using a test driver. Test driver can generate test data using data characterization based on equivalence class definitions. Using test driver, application is exercised without risking the actual environment. Again, the software can be put to test using large number of test suits, without much effort, as data are generated meaningfully and automatically [Mill84, Panz78, Pres92]. Without such a facility, only sketchy testing will be feasible, as the effort required in generating the data manually will be comparatively enormous.

The test driver can be developed in such a fashion that it generates typical volume and frequency of various transactions resembling real life situations. For example, in the case of a library information system, it could generate specified number of books added to library, books issued, returned, users added etc. Test driver developed with suitable control over the number and frequency of various transactions to be generated, can be used to simulate working of the library systems for the desired number of days. The various reports produced will also be realistic in character, and various other options like over-due books, fine option, reminder generation etc., can also be exercised more comprehensively. Sufficient data can be loaded into the database resembling various real life situations, like normal load or overloaded situations etc. Testing can be done in various states of the database, which may reveal many faults, which otherwise would have surfaced during the actual use of the software.

B.2 Test Driver Development for 3GL Based Interactive Applications

Most of the data base applications usually have provisions to update the database and generate various reports. The updation could be off-line in batch mode, or on-line in interactive mode. As computer systems became widely available, interactive applications have become common. The facilities provided in interactive mode are quite large and complex in nature. The application programs have very large component of code devoted to handle the user interface.

Off-line applications take transaction files as an input to update the database in batch mode. To test such applications, one had to simply create transaction files containing various types of transactions. Testing of on-line interactive applications consists of two distinct components : testing of user interface and testing of application processing logic. These two are generally intimately interwoven in the code.

The interface logic, i.e., how the control flows from one field to another, from one form to another etc., can be easily tested by simply exercising the various options available in various menus. There is no need to build any extra observability for testing of user-interface logic, as any anomalous behavior is readily observable.

For testing of the application processing logic, if the meaningful combinations are large, or if one wishes to load the system with reasonable amount of data, manual procedure will be very cumbersome and time consuming. Therefore, driving the application automatically through test driver with meaningfully generated data, will be highly desirable.

The data can be defined to the system using an equivalence class definition language, like described in HUTEST (Appendix A). The form handlers of 4GL products, can be easily extended to make use of these data definitions for automatic data generation, and for

driving the application. In 3GL programming environment, these definitions cannot be utilized very easily. The next section discusses the application development discipline to be followed, to facilitate test driver development in 3GL environment using equivalence class definitions.

B.3 Application Development Discipline Required to Facilitate Test Driver Development

Normally, the application software is one monolithic whole, which has code to take care of interaction with the users, access to database, and processing of application logic. To drive such an application automatically, by providing input data using the indirection mechanism (which is possible in Unix), will be quite difficult, as one will have to anticipate the entire sequence in which the user-software interaction will take place. Some of the sequences of inputs may be dependent on data base state, which is very difficult to anticipate.

To facilitate test driver development in 3 GL environment, it is proposed, that the application software should be developed in such a way that it has the following two well-defined components.

1. User Interface
2. Application Dependent Logic

Normally, code for these two tasks are intermixed, as there is no explicit requirement to keep them separate. Though, application dependent code forms a major part in any software, the user interface code also tends to acquire complexity due to the required user-friendliness and flexibility in operations.

The application dependent logic can be developed independent of the user interface. Any processing to be done on the database e.g., validations, retrieving, appending, updation or computations etc., should be done by the application dependent logic components as

various functions. The user interface component can string these application functions together along with the user interaction, to give shape to the total application.

If the above design discipline is followed, i.e., these two aspects of application are developed as separate modules, writing a test driver to test the application dependent logic component, can become an easy task. The test driver can be developed using the functions for data generation based on equivalence class definitions of data characteristics and the application dependent logic components. Figure B.1 shows the application architecture present in software, developed in a conventional manner. Figure B.2 shows the application architecture in the proposed approach for test driver development.

The test driver can provide user commands for executing the test driver in the desired manner. It can provide options, to generate various types of transactions in required numbers and frequencies. The application reports can be tested using the actual application software itself, as it can use the data created during the test driver execution. The test driver can be used to run the application to reflect the typical volume of daily transactions. The application can be advanced in time and transactions can be processed for the next day. Thus, the application can be executed for number of days in one testing session itself. In this way, one can test time dependent features, e.g., in the case of a library system, overdue books, reminder generation, renewal of journal subscriptions etc. Whatever can be exercised, thoroughly, using the actual application, need not be developed in the test driver. And hence, One should use a mix of actual application execution through its user interface, and the test driver for purpose of comprehensive testing of application testing.

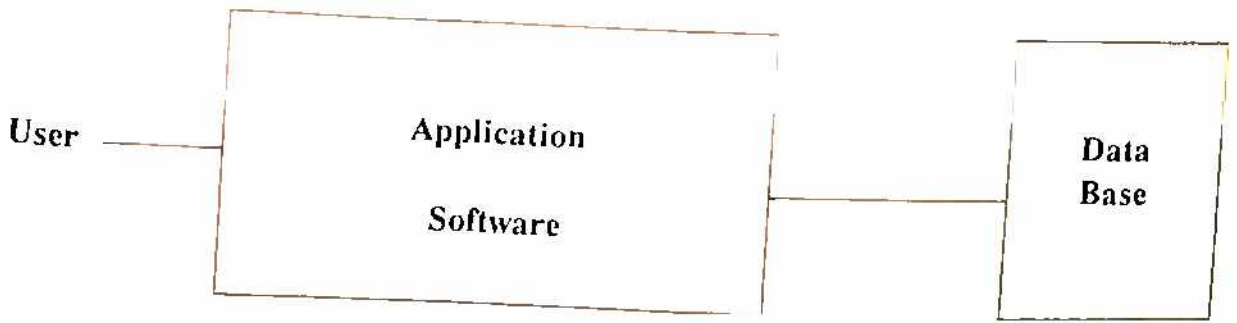


Fig B.1 Conventional Application Software Structure

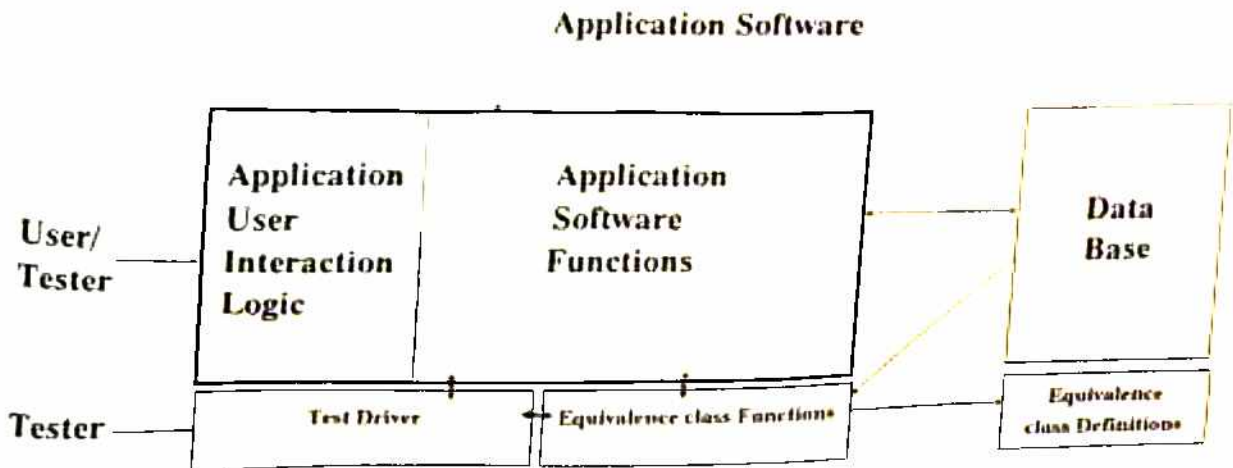


Fig B.2 Proposed Application Software structure to facilitate Test Driver Development in 3GL

For example, in a library circulation system, to issue/return a book, the inputs required are

- user_id
- book_id (or Accn. No.)
- issue/return

After the user_id is input, it has to be checked for validity, i.e., active user, or user having too many books already issued etc.; before the next input is taken. All the processing can be done in the function called

```
validate_user (user_id) which  
returns (name, status, category, number_of_books,...)
```

The call returns with, the name, status, category of user, number of books issued etc. This routine will contain the application dependent logic to interpret these data. Similarly, when book_id is taken as input, call can be made to functions

- display_book (book_id) returns (title, author,...)
- issue_book (book_id, user_id) returns (error_no)

The first function will return some information about the book, which can be displayed for confirmation. On selection of issue option, the function issue_book (book_id, user_id) can be called, which will check whether the book can be issued or not. If it is possible to issue the book, it will perform the required updations in the database, otherwise, it will return the appropriate error number. Here also, interpretation of various states of the book, issued, returned etc., are hidden in this function.

Once the application has been decomposed into separate modules, i.e., application logic and user interface; functions of these modules can be used to write a test driver. The driver can generate the user_id and book_id based on equivalence class definitions, and call the application functions for performing various operations.

In the above examples, we saw, that only providing facility to randomly generate data based on data characteristics is not enough for a valid transaction. The randomly generated value has to be validated against the current state of the database. The value has to be generated again, till a valid value is encountered. Alternatively, a value can be picked up randomly from the appropriate database table. Even, special cases, like serial numbers for accession number, can be handled with very little programming effort.

The control flow logic of a test driver is not complex in nature. The test driver development is also simplified to a great extent, as it uses already developed application logic and the library for test data creation. Thus, the development of the test driver can be done relatively easily. The return of the investment, in test driver development, is very quick, as it saves considerable amount of time and effort required for executing same amount of tests manually.

B.4 Advantages of Design Discipline for Test Driver Development

Separation of application logic from interface logic has several benefits.

This framework provides another very natural way of modularisation. The first level of modularisation is achieved by the menu and sub-menu structuring of the application. Each menu option or a set of menu options, defines a separate program to be developed. This can be considered as the vertical slicing of the software. Separation of interface and the application logic, is the horizontal slicing of the software. Any fault can be either in the interface component, or in the application logic component. Again, whenever, any change in application processing logic takes place, it can be done very easily, without worrying about the interface component.

Again, many of the application's functional units, are usable in multiple places. For example, user_id has to be validated in several options like, book issue, book return etc. As, the above discipline, forces the programmer to create a separate function, any change in validation logic, will have to be made in one place. Thus, even the application logic component, will also be very well structured, and hence, easier to maintain.

B.5 Building Observability in Application Logic

The behavior of the interactive part of the software is visible to the user, and hence, there is no need for building any further observability into it. However, the application logic may require some level of observability to be built-in, especially, where the logic is complex. Generally, most of the application logic is fairly simple, and the result of the processing is recorded in the database, which is accessible to the user through various reports and interactive queries. Only in complex parts of the code where direct observability is desirable, it should be built-in using the probe mechanism. Certain third generation languages, like 'C' (in which considerable application software development is done, specially where efficiency is a major consideration), which are quite error prone, observability can prove to be very useful for fault rectification and performance improvement purposes.

END OF APPENDIX B

References

- [Abbo86] Abbott J., "Software Testing Techniques", NCC Publication, UK, 1986.
- [Agra91] Agrawal H., Demillo R.A. and Spafford E.H., "An Execution - Backtracking Approach to Debugging", IEEE Software, pp 21-26, May 1991.
- [Arak91] Araki K. Furukawa Z. and Cheng J., "A General Framework for Debugging", IEEE Software, pp 14-20, May 1991.
- [Balc89] Balcer M., Hasling W. and Ostrand T., "Automatic Generation of Test Scripts from Formal Test Specifications", Proc. of ACM SIGSOFT'89, 3rd Symposium on Software Testing, Analysis & Verification, pp 210-218, 1989.
- [Basi87] Basili V.R. and Selby R.W., "Comparing the Effectiveness of Software Testing Strategies", IEEE Trans. Software Eng., Vol. SE-13, No. 12, pp 1278-96, December 1987.
- [Bate83] Bates P.C. and Wileden J.C., "High-Level Debugging of Distributed Systems: The Behavioural Abstraction Approach", Journal of Systems and Software 3, pp 205-214, 1983.
- [Beiz90] Beizer B., "Software Testing Techniques", Van Nostrand, 2nd Edition, 1990.
- [Bigg89] Biggerstaff T.J, "Design Recovery for Maintenance and Reuse", IEEE Computer, pp 36-49, July 1989.
- [Boeh81] Boehm B., "Software Engineering Economics", Prentice Hall, EagleWood Cliffs, NJ, 1981.

- [Cheu90] Cheung W.H., Black J.P. and Manning E., "A Framework for Distributed Debugging", IEEE Software, pp 106-115, January 1990.
- [Chik90] Chikofsky E.J. and CrossII J.H., "Reverse Engineering and Design Recovery", IEEE Software, January, 1990.
- [Choi90] Choi S.C. and Scacchi W., "Extracting and Restructuring the Design", IEEE Software, January, 1990.
- [Faga76] Fagan M.E., "Design & Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, Vol. 15, No. 3, pp 182-211, 1976.
- [Faga86] Fagan, M.E., "Advances in Software Inspections", IEEE Trans. Software Eng., SE-12, No.7, pp 744-51, July 1986.
- [Garc84] Garcia-Molina H., German F. JR., and Kohler W.H., "Debugging a Distributed Computing System", IEEE Trans. Software Eng., Vol. SE-10, No. 2, pp 210-219, March 1984.
- [Gibs89] Gibson V.R. and Senn J.A., "System Structure and Software Maintenance Performance", CACM, pp 347-57, March 1989.
- [Gilb88] Gilb T., "Principles of Software Engineering Management", Workingham, England, Addison-Wesley, 1988.
- [Gill191] Gill G.K. and Kemerer C.F., "Cyclomatic Complexity Density and Software Maintenance Productivity", IEEE Trans. Software Eng., Vol. 17, No. 12, pp 1284-88, December 1991.
- [Good75] Goodenough J.B., and Gerhart S.L. "Toward a Theory of Test Data Selection", IEEE Trans. Software Eng., SE-1. NO. 2, pp 156-73, June 1975, .

- [Gupt92] Gupta S. C. and Sinha M. K., "Design Validation Using Probe Mechanism", in Proc. 12th World Computer Congress, IFIP Congress '92, Spain, pp 24-31, September 1992.
- [Gupt91] Gupta S.C. and Sinha M.K., "Testability and Maintainability as criteria for Software Design and Implementation", Proc. Int'l. Conf. on Software Eng., CONSEG '91, Bangalore, India, pp 10-11, October 1991.
- [Ham189] Halmet R., "Theoretical Comparision of Testing Methods", Proc. of ACM SIGSOFT '89, 3rd Symposium on Software Testing, Analysis & Verification, pp 28-37, 1989.
- [Harr82] Harrison W., Magel K., Kluczny R. and DeKock A., "Applying Software Complexity Metrics to Program Maintenance", IEEE Computer, Vol. 15, No. 9, pp 65-79, September 1982.
- [Harr90] Harrison W. and Cook C. , "Insights on Improving the Maintenance Process through Software Measurement", Proc. Conf. Software Maintenance, San Diego, CA, pp 37-44, November 1990.
- [Haye86] Hayes I.J., "Specification Directed Module Testing", IEEE Trans. Software Eng., Vol. SE-12, No. 1, pp 124-133, January 1986.
- [Heat91] Heath M.T. and Etheridge J.A., "Visualizing the Performance of Parallel Programs", IEEE Software, pp 29-39, September 1991.
- [Henr90] Henry S. and Selig C., "Predicting Source-Code Complexity at Design Stage", IEEE Software, pp 36-44, March 1990.
- [Hetz88] Hetzel B., "The complete guide to Software Testing", 2nd Edition, QEP Information Sciences, Inc., 1988.

- [Howd81a] Howden W.E., "A Survey of Static Analysis Methods", IEEE Tutorial on Software Testing & Validation Techniques, Ed. Edward Miller & W.E. Howden, pp 101-115, 1981.
- [Howd81b] Howden W.E., "A Survey of Dynamic Analysis Methods", IEEE Tutorial on Software Testing & Validation Techniques, Ed. Edward Miller & W.E. Howden, pp 209-231, 1981.
- [Huan79] Huang J.C., "Data Flow Anomaly Detection through Program Instrumentation", IEEE Trans. Software Eng., Vol. SE-5, No. 3, pp 226-36, May 1979.
- [Ieee87] _____, IEEE Std., "Software Engineering Standards", IEEE Press, 1987.
- [Ie83TD] _____, IEEE std 829-1983, Standard for Software Test Documentation, IEEE Publication, 1983.
- [Ie87UT] _____, IEEE std 1008-1987, Standard for Software Unit Testing, IEEE Publication, 1987.
- [Ie87VV] _____, IEEE std 1012-1987, Standard for Software Verification and Validation, IEEE Publication, 1987.
- [Ie88RA] _____, IEEE std 1028-1988, Standard for Software Reviews and Audits, IEEE Publication, 1988.
- [Ie89QA] _____, IEEE std 730-1989, Standard for Software Quality Assurance Plans, IEEE Publication, 1989.
- [Ince85] Ince D., "The Validation, Verification and Testing of Software", in Oxford Surveys in Information Technology, Vol. 2, pp 1-40, 1985.

- [Jones87] Jones S.H., Barkan R.H. and Wittie R.D., "Bugnet: A Real Time Distributed Debugging System", Proc. 6th Symposium on Reliability in Distributed Software and Database Systems, IEEE Publication, pp 56-65, 1987.
- [Kafu87] Kafura D. and Reddy G.R., "The Use of Software Complexity Metrics in Software Maintenance", IEEE Trans. Software Eng., 13. No. 3, pp 335-43, 1987.
- [Kore88] Korel B., "PELAS - Program Error-Locating Assistant System", IEEE Trans. Software Eng., Vol. 14, No. 9, pp 1253-60, September 1988.
- [Kore90] Korel B., "Automated Software Test Data Generation", IEEE Trans. Software Eng., Vol. 16, No. 8, pp 870-79, August 1990.
- [LeBl87] LeBlanc T.J. and Mellor-Crummey J.M., "Debugging Parallel Programs with Instant Replay", Proc. of Int'l Conf. on Distributed Computing Systems, 1987.
- [Levi92] Levitt M.E., "ASIC Testing Upgraded", IEEE Spectrum, pp 26-29, May 1992.
- [Lutz90] Lutz. M., et. al., "Testing tools", IEEE Software, pp 53-7, May 1990.
- [McCa85] McCabe T.J., and Schulmeyer G.G., "System Testing Aided by Structured Analysis: A Practical Experience", IEEE Trans. Software Eng., Vol. SE-11, No. 9, pp 917-21, September, 1985.
- [McCa89] McCabe T.J. and Butler C.W., "Design Complexity Measurement and Testing", CACM, Vol. 32, No. 12, pp 1415-25, December 1989.
- [McDo89] McDowell C.E. and Helmbold D.P., "Debugging Concurrent Programs", ACM Computing Surveys, Vol. 21, No. 4, pp 593-622, December 1989.

- [Mill78] Miller E.F. and Howden W.E. (eds & contributors), "Tutorial: Software Testing & Validation Techniques", IEEE Computer Society, 1978.
- [Mill84] Miller E.F., Jr., "Software Testing Technology : An Overview", in Handbook on Software Engineering, Ed. C.R. Vick and C.V. Ramamoorthy, Van Nostrand Reinhold Company Inc., pp 359-378, 1984.
- [Mill88] Miller B.P. and Choi J.D., "A Mechanism for Efficient Debugging of Parallel Programs", Proc. of SIGPLAN'88 Conf., pp 135-144, 1988.
- [Mohe88] Moher T.G., "PROVIDE: A Process Visualization and Debugging Environment", IEEE Trans. Software Eng., Vol. 14, No. 6, pp 849-857, June 1988.
- [Mos193] Mosley D.J., "The Handbook of MIS Application Software Testing", Yourdon Press Computing Serie, New Jersey, 1993.
- [Myer79] Myers G. J., "The Art of Software Testing", John Wiley & Sons, New York, 1979.
- [Neuf93] Neufelder A.M., "Ensuring Software Reliability", Marcel Dekker, Inc., 1993.
- [Ntaf88] Ntafos S.C., "A Comparison of Some Structural Testing Strategies", IEEE Trans. Software Eng., Vol. 14, No. 6, pp 868-874, June, 1988.
- [Olss91] Olsson R.A., Crawford R.H., Ho W.W. and Wee C.E., "Sequential Debugging at High Level of Abstraction", IEEE Software, pp 27-36, May 1991.

- [Oman90a] Oman P. and Cook C.R., "Typographic Style is More than Cosmetic", CACM, Vol. 33, No. 5, pp 506-520, May 1990.
- [Oman90b] Oman P., et.al., "Maintenance Tools", IEEE Software, pp 59-64, May, 1990.
- [Osbo90] Osborne W.M. and Chikofskly E.J., "Fitting Pieces to the Maintenance Puzzle", IEEE Software, January, 1990.
- [Oatr88] Ostrand T.J. and Balcer M.J., "The Category-Partition Method for Specifying and Generating Functional Tests", CACM, Vol. 31, No. 6, pp 676-686, June, 1988.
- [Ould86] Ould M.A. and Unwin C. Ed., "Testing in software Development", British Computer Society, Cambridge University Press, 1986.
- [Parn77] Parnas D.L., "The Influence of Software Structure on Reliability", in Current Trends in Programming Languages Vol. I, Ed. Raymond T.Yeh, Prentice Hall, New Jersey, pp 111-19, 1977.
- [Parn90] Parnas D.L., Schouwen A.J.V, and Kwan S.P., "Evaluation of Safety Critical Software", CACM, Vol. 33, No. 6, pp 636-648, June 1990.
- [Panz78] Panzl D.L., "Automatic Software Test Drivers", Computer, pp 44-50, April 1978.
- [Pres92] Pressman R.S., "Software Engineering - A Practitioner's Approach", 3rd Edition, McGraw-Hill International Edition, 1992.
- [Raja89] Rajaram L.N. and Gupta S.C., "An Integrated Framework for User-Driven Interactive Application Development", in Proc. 11th World Computer Congress, IFIP Congress '89, San Francisco, USA, pp 357-362, August 1989.

- [Rama75] Ramamoorthy C.V. and Ho S.F., "Testing Large Software with Automated Software Evaluation Systems.", in Current Trends in Programming Methodology, Vol. II, Ed. Raymond T. Yeh, Prentice Hall, New Jersey, pp 112-150, 1977.
- [Rama75a] Ramamoorthy C.V. and Kim K.H., "Optimal Placement of Software Monitors Aiding Systematic Testing", IEEE Trans. Software Eng., Vol. SE-1, No. 4, pp 403-410, 1975.
- [Rama76] Ramamoorthy C.V., Ho S.F. and Chen W.T., "On the Automated Generation of Program Test Data", IEEE Trans. Software Eng., Vol. SE-2, No. 4, pp 293-300, December 1976.
- [Rapp85] Rapps S. and Weyuker E.J., "Selecting Software Test Data Using Data Flow Information", IEEE Trans. Software Eng., Vol. SE-11, No. 4, pp 367-75, April 1985.
- [Schn87] Schneidewind N.F., "The State of Software Maintenance", IEEE Trans. Software Eng., Vol. SE-13, No. 3, pp 303-310, March 1987.
- [Schu87] Schultz R.D. and Cardenas A.F., "An Approach and Mechanism for Auditable and Testable Advanced Transaction Processing System", IEEE Trans. Software Eng., Vol. SE-14, No. 6, pp 666-676, June 1987.
- [Selb87] Selby R.W., Basili V.R., and Baker F.T., "Cleanroom Software Development: An Empirical Evaluation", IEEE Trans. Software Eng., Vol. SE-13, No. 9, pp 1027-37, September, 1987.
- [Shim91a] Shimomura T. & Isoda S., "Linked-list Visualization for Debugging", IEEE Software, pp 44-52, May 1991;
- [Shim91b] Shimomura T. and Isoda S. , "CHASE : A Bug-Locating Assistant System", Proc. 15th Annual Int'l Computer Software and Applications Conference, IEEE Publication, pp 412-17, 1991.

- [Sinh89] Sinha M.K., Gupta S.C. and Radhika Ramnath, "HUTEST : An Automatic Testing Tool for An Integrated Application Development Environment", Internal Report, Expert Software Consultants, New Delhi, 1989.
- [Stuc77] Stucki L.G., "New Directions in Automated Tools for Improving Software Quality", in *Current Trends in Programming Methodology*, Vol. II : Program Validation, Ed. Raymond T.Yeh, Prentice Hall, New Jersey, pp 80-111, 1977.
- [Tifa90a] _____ "TIFACLINE Host Software - User Specification Document", TIFAC, Dept of Science & Technology, New Delhi, 1990.
- [Tifa90b] _____ "TIFACLINE Host Software - Detailed Design Document", TIFAC, Dept of Science & Technology, New Delhi, 1990.
- [Tsai90] Tsai J.J.P., Fang K.Y., Chen H.Y., and Bi Y.D., "A Non-Interference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging", *IEEE Trans. Software Eng.*, Vol. 16, No. 8, August 1990.
- [Taub86] Tsubotani H., Monden N., Tanaka M. and Ichikawa T., "A High Level Language-Based Computing Environment to Support Production and Execution of Reliable Programs", *IEEE Trans. Software Eng.*, Vol. SE-12, No. 2, pp 134-146, February 1986.
- [Unix92] _____ "Programmer Guide - ANSI C & Programming Support Tools", AT&T Unix System V Release 4 Manual, Prentice Hall of India, 1992.
- [Whit87] White L.J., "Software Testing and Verification", in *Advances in Computers*, Vol. 26, pp 335-391, 1987.

END OF REFERENCES

List of Publications

- [Gupt92] Gupta S. C. and Sinha M. K., "Design Validation Using Probe Mechanism", in Proc. 12th World Computer Congress, IFIP Congress'92, Spain, pp 24-31, September 1992.
- [Gupt91] Gupta S.C. and Sinha M.K., "Testability and Maintainability as criteria for Software Design and Implementation", Proc. Int'l. Conf. on Software Eng., CONSEG'91, Bangalore, India, pp 10-21, October 1991.
- [Gupt93] Gupta S.C. and Sinha M.K., "Improving Software Maintainability by Observability and Controllability Measures", to be sent for publication, 1993
- [Raja89] Rajaram L.N. and Gupta S.C., "An Integrated Framework for User-Driven Interactive Application Development", in Proc. 11th World Computer Congress, IFIP Congress'89, San Francisco, USA, pp 357-62, August 1989.
- [Sinh89] Sinha M.K., Gupta S.C. and Radhika Ramnath, "HUTEST : An Automatic Testing Tool for An Integrated Application Development Environment", Internal Report, Expert Software Consultants, New Delhi, 1989.

END OF THESIS

7.1.2 Execution time Activation Control

Observability and controllability measures, which have remained attached with the executable software version, can be further controlled at execution time to reduce their overheads. The observability measures (probes) remain deactivated, unless activated by control commands. Thus only activated probes will record information in event history file. Similarly, controllability code, integrated in the main logic or existing as additional functions, will get executed only when activated through control interface commands. Execution time processing overheads can, thus, be reduced to minimum. Deactivation of these measures reduces processing overheads but not memory overheads.

7.1.3 Active Probe Identification

If large number of probes are embedded in the executable version of the software, then certain overheads will be incurred to check whether the probe is active or not. We propose to run software in one of the two modes : "operation" mode and "test" mode. When the software runs in "operation" mode, all observability measures are treated as "no-op". In "test" mode, one would have to check the probe activation status each time a probe is encountered. Checking of activation status of a probe requires checking against a command table, containing entries of all the activation and deactivation commands. After testing against all the entries, the resultant status is used to decide whether the probe is active or not.

To expedite this checking, probes may be divided into broad classes, based on categories of probes (like design, maintenance, monitoring etc.), modules, sub-modules etc. Thus checking can first be done against these major classes. Only if the related class flag is on, then more detail check can be performed. Also the check can be restricted to entries of that class alone. If any activation / deactivation command refer to multiple categories, entries would have to be made in each category table.