

## ABSTRACT

The objective of this project is to survey coverage techniques for software testing and to apply these techniques to the author's working environment, IBM System/390 mainframes. We hope to explore the relationship between coverage and reliability with live testing data and failure statistics.

We review the basic concepts on software testing, and more specifically on the coverage techniques. The theoretical background of control and data flow coverage is presented with the illustration by real-life programming examples. We show that the data flow coverage criteria can identify possible problematic paths that maps to the actual testing semantic required by Y2K compliance software testing.

Then we look into the state-of-the-art coverage testing tool ATAC (Automatic Test Analysis for C) developed by Bellcore in detail. We also survey coverage testing tools available in the mainframe industry. We found that the data flow coverage technique equipped by ATAC had not been applied to any of the mainframe testing tools. In viewing of that, we propose to develop a coverage testing tool ATACOBOL (Automatic Test Analysis for COBOL) for COBOL language on the mainframe platform.

Up to the current implementation, ATACOBOL is able to perform block coverage, decision coverage and all-uses measures. We extend the rules of data flow coverage criteria to adapt data structures that modern high-level languages usually employ. We applied ATACOBOL to measure the control and data flow coverage of some production test cases. Some basic relationships between coverage and reliability is reflected through the measurement.

# TABLE OF CONTENTS

|   |           |
|---|-----------|
| <b>ABSTRACT .....</b>   | <b>1</b>  |
| <b>TABLE OF CONTENTS .....</b>  | <b>2</b>  |
| <b>1. INTRODUCTION .....</b>  | <b>5</b>  |
| 1.1 SOFTWARE TESTING AND ITS IMPORTANCE .....   | 5         |
| 1.2 OVERVIEW OF SOFTWARE TESTING .....  | 5         |
| 1.3 OVERVIEW OF COVERAGE TECHNIQUE.....   | 6         |
| 1.4 BACKGROUND INFORMATION ABOUT THE AUTHOR'S WORKING ENVIRONMENT.....                                    | 8         |
| 1.5 ORGANISATION OF THIS REPORT.....  | 8         |
| <b>2. BACKGROUND THEORY.....</b>  | <b>10</b> |
| 2.1 THE BASIS OF PROGRAM TESTING.....   | 10        |
| 2.2 THE NEED FOR PATH SELECTION CRITERIA.....   | 10        |
| 2.3 CONTROL FLOW COVERAGE.....  | 11        |
| 2.3.1 Block Coverage and Edge Coverage.....   | 11        |
| 2.3.2 A Real-Life Example Demonstrating the Subsumption Relation between Coverage and Edge Coverage ..... | 12        |
| 2.4 DATA FLOW COVERAGE.....   | 13        |
| 2.4.1 The Arise of Data Flow Coverage.....  | 13        |
| 2.4.2 Def/Use Pair.....   | 14        |
| 2.4.3 The Def/Use Graph.....  | 15        |
| 2.4.4 Family of Data Flow Selection Criteria .....  | 18        |
| 2.4.5 Real-Life Y2K Example Demonstrating Advantage of Data Flow Coverage.....                            | 19        |
| 2.4.6 Extented Example that Required All-DU-Paths Criteria .....  | 20        |
| 2.4.6 Global Payments Systems Test.....   | 23        |
| <b>3. SURVEY ON COVERAGE TESTING TOOLS.....</b>   | <b>25</b> |
| 3.1 ATAC (AUTOMATIC TEST ANALYSIS FOR C) .....  | 25        |
| 3.2 SURVEY ON COVERAGE TOOLS FOR COBOL IN MAINFRAME PLATFORM .....  | 26        |
| 3.2.1 Status of COBOL in the Mainframe Platform .....   | 26        |
| 3.2.2 COBOL Coverage Tools on Mainframe .....   | 27        |
| <b>4. IMPLEMENTATION.....</b>   | <b>30</b> |
| 4.1 OVERVIEW .....  | 30        |
| 4.2 ENVIRONMENT SETUP .....   | 30        |
| 4.3 APS COBOL .....   | 31        |
| 4.4 ATACOBOL ARCHITECTURE.....  | 32        |
| 4.5 ATACOBOL CODE PARSER .....  | 33        |
| 4.6 ATACOBOL INSTRUMENTER.....  | 43        |
| 4.7 ATACOBOL RUNTIME ROUTINE .....  | 44        |
| 4.8 ATACOBOL COVERAGE ANALYSER .....  | 45        |
| <b>5. MEASUREMENT .....</b>   | <b>51</b> |
| 5.1 SYSTEM DESCRIPTION .....  | 51        |
| 5.2 NUMBER OF C-USE AND P-USE AGAINST NUMBER OF FAULTS .....  | 52        |
| 5.3 DATA FLOW COVERAGE OF LIVE TEST CASES.....  | 56        |
| <b>6. EVALUATION.....</b>   | <b>58</b> |
| 6.1 EXPERIENCE IN USING ATAC .....  | 58        |
| 6.1.1 The applications of ATAC .....  | 58        |
| 6.1.2 Limitations of ATAC.....  | 60        |
| 6.2 EXPANDING THE DATA FLOW COVERAGE DEFINITION TO C .....  | 63        |

|   |            |
|---|------------|
| 6.2.1 Handling Global Variables .....   | 63         |
| 6.2.2 Handling Array Elements .....   | 64         |
| 6.2.3 Handling Pointers .....   | 66         |
| 6.2.4 Handling Structure .....  | 67         |
| 6.2.4 Handling Union .....  | 68         |
| 6.3 EXPERIENCE IN USING ATACOBOL .....  | 68         |
| 6.3.1 Using ATAC to Measure ATACOBOL .....  | 68         |
| 6.3.2 The Usefulness of Enhanced Rules on Data Structures .....                   | 69         |
| 6.3.3 Comparing ATACOBOL, ATAC and ASSET .....                                    | 70         |
| 6.3.4 Suggested ATACOBOL Further Development .....                                | 71         |
| <b>7. DISCUSSION .....</b>  | <b>73</b>  |
| 7.1 ANOTHER TYPE OF DEF/USE IN ASSEMBLY LANGUAGE — ADDRESSING USE .....           | 73         |
| 7.2 EFFECT OF UNEXECUTABLE PATH TO DATA FLOW COVERAGE CRITERIA .....              | 74         |
| 7.3 COMPLEXITY OF DATA FLOW COVERAGE .....  | 76         |
| 7.3.1 Complexity of Data Flow Coverage Criteria .....                             | 76         |
| 7.3.2 Complexity of Data Flow Coverage Measurement .....                          | 79         |
| 7.4 COMPARISON OF THE FAULT-DETECTING ABILITY OF COVERAGE CRITERIA .....          | 80         |
| 7.4.1 Problem of Empirical Comparison of Effectiveness .....                      | 80         |
| 7.4.2 Problem of Analytical Comparison of Effectiveness .....                     | 81         |
| 7.4.2 PROBBETTER Relation of Coverage Criteria .....                              | 81         |
| 7.5 CODE COVERAGE AND RELIABILITY .....   | 81         |
| 7.5.1 Inadequacy of Operational Profile in Reliability Estimation .....           | 81         |
| 7.5.2 Overestimation of Reliability due to Saturation Effect .....                | 83         |
| 7.6 DEVELOPING AND USING ATACOBOL IN CROSS-PLATFORM APPROACH: PROS AND CONS ..... | 85         |
| 7.7 INCORPORATION OF COVERAGE METRICS TO ARMOR .....                              | 87         |
| 7.8 POTENTIAL BY-PRODUCTS OF ATACOBOL .....                                       | 88         |
| 7.9 VISUAL-AID FOR COVERAGE ANALYSIS .....  | 88         |
| 7.10 COVERAGE ANALYSIS FOR PROGRAM VERSION CHANGES .....                          | 90         |
| 7.11 DATA FLOW COVERAGE: TO USE OR NOT TO USE? .....                              | 91         |
| <b>8. SCHEDULE .....</b>  | <b>94</b>  |
| 8.1 PROJECT IMPLEMENTATION SCHEDULE .....   | 94         |
| 8.2 RESOURCES .....   | 94         |
| 8.2.1 Hardware .....  | 94         |
| 8.2.2 Software .....  | 95         |
| 8.2.3 Human Resources .....   | 95         |
| <b>9. CONCLUSION .....</b>  | <b>96</b>  |
| <b>10. REFERENCE .....</b>  | <b>97</b>  |
| <b>APPENDIX A ATACOBOL PROGRAM SPECIFICATIONS .....</b>                           | <b>101</b> |
| A.1 MODULE SPECIFICATIONS .....   | 101        |
| A.1.1 Normaliser .....  | 101        |
| A.1.2 Router .....  | 101        |
| A.1.3 Variable Table Builder .....  | 102        |
| A.1.4 Data Flow Graph Builder .....   | 102        |
| A.1.5 Def-Use Path Searcher .....   | 102        |
| A.1.6 Coverage Analyser .....   | 103        |
| A.2 FILE LAYOUTS .....  | 103        |
| A.2.1 Program Primitive Structure .....   | 103        |
| A.2.2 Control Flow Information File .....   | 104        |
| A.2.3 Variable Table File .....   | 104        |
| A.2.4 Data Flow Information File (Def) .....                                      | 104        |
| A.2.5 Data Flow Information File (C-Use) .....                                    | 104        |
| A.2.6 Data Flow Information File (P-Use) .....                                    | 105        |
| A.2.7 Data Flow Information File (DCU) .....                                      | 105        |
| A.2.8 Data Flow Information File (DPU) .....                                      | 105        |

---

**APPENDIX B ATACOBOL TUTORIAL..... 106**

# 1. INTRODUCTION

## 1.1 Software Testing and Its Importance

The product of any engineering activity must be shown to be correct according to its requirements throughout its development. Software testing is the dynamic verification and validation of developed software system. Software is to experiment with the behaviour of the system. Verification answers the question: “Are we building the product right?” while validation answers the question “Are we building the right product?”.

Software testing shows the presence of bugs and to locate errors, so as to assure the correctness and reliability of a software system. For many modern systems, including networking and telecommunication systems, banking and finance systems, and consumer products, software plays a crucial role in their daily control, communication and operations. Software failure in the new airport project of Hong Kong SAR has caused chaos and great financial lost. More seriously, software failure kills people.

Unfortunately, testing software is perhaps more difficult and costly than testing other engineering products. It is estimated that over 50% of the development cost will be spent on testing.

## 1.2 Overview of Software Testing

Software testing can be divided into categories according to the objective, method of testing and development phase. From software development perspective, we can have unit test, integration test and product test from one phase to the other. The unit test is the coding of small software components, the integration tests the integrated large

component, and the product test software in its final form. In the re-release of a modified software, the process to test the new software with test cases from the old version is called regression test.

According to the test methods, we have *white-box testing* and *black-box testing* [Ghe97], [Som96]. Any of these methods might be applied during the test phases mentioned above.

White-box, or *coverage* testing is program structure-based. This approach employs information about the internal structure to derive test cases and it focus on what a program does.

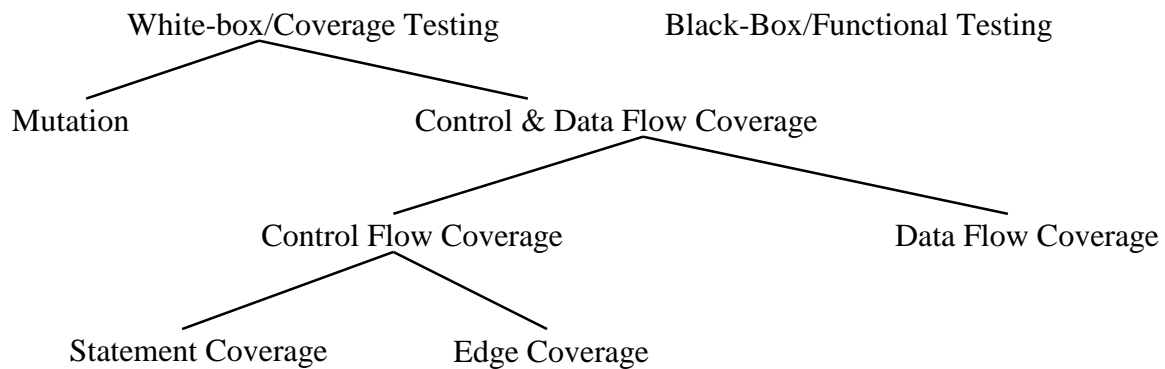
Black box, or *functional* testing, on the other hand, is requirement based. We derive test cases purely based on the specification and test what a program is supposed to do.

Software testing aims to verify and validate the program developed, and therefore, on the believe to achieve higher software reliability.

Nevertheless, testing is an expensive active in software development. From another point of view, software testing offers lots of research opportunities. “When to stop testing?” is a challenging question towards all software testers. It is prosperous to develop methods for considering minimal test sets. Automatic test case generation has been an active research area.

### 1.3 Overview of Coverage Technique

There are two streams of coverage testing methods: *Data and control flow testing*, and *mutation testing*. Control flow testing and data flow testing are the sub-categories of control and data flow testing. Control flow can further divided into sub-streams of *statement coverage* and *edge coverage* (Figure 1.1).



**Figure 1.1** Software Testing Methods

- Statement coverage testing directs the tester to construct test cases such that each statement or a basic block of code is executed at least once.
- Edge coverage testing directs the tester to construct test cases such that each decision edge in the program is covered at least once.
- Data flow coverage testing directs the tester to construct test cases such that all the *def-use pairs* are covered. (The theoretical background of data flow coverage will be discussed in more detail in chapter 2).

Mutation testing helps a tester design test cases from an approach very different from that of the path-oriented testing strategies mentioned above. Mutation testing generates several syntactically correct *mutants* of a given program. A mutant is obtained from making a change in the original program in accordance with a set of rules. Mutation testing requires a tester to generate test data that distinguish all non-equivalent mutants of the program [Chio89].

In this project, we only concentrate on the application of control and data flow coverage methods.

#### 1.4 Background Information about the Author's Working Environment

This project targets at the application of code coverage techniques on the author's working environment.

The author is a mainframe programmer of the Hong Kong and Shanghai Banking Corporation Ltd (HSBC). The production mainframe serves HSBC as well as the Hang Seng Banking Corporation Ltd (HASE) of the HSBC Bank Group. Most corporate and customer banking systems are running on the production mainframe. They includes the Credit Card System, Current Account System, Savings Account System, Foreign Exchange System, Remittance System, Bank Fund Transfer System, Instalment Loan System, ... etc.

HSBC and HASE plays the leading role and shares a large proportion of the local banking business. The two banks serve over than 60% of the current and savings accounts of Hong Kong SAR. It reflects that the mainframe program development environment of HSBC can have high representative of the local banking industry.

#### 1.5 Organisation of this Report

In this project, we survey code coverage techniques for software testing. Based on the survey, we proposed a code coverage tool for COBOL in mainframe platform. Here briefs the content of each report chapter.

- Chapter 1 gives an introduction of this project. It reviews the current status of software testing and, in particular, describes the coverage methods. The working environment of the author is also described.
- Chapter 2 presents the theoretical background of the control and data flow coverage methods. The data flow selection criteria proposed by Weyuker *et al* [Rap85], [Fra88] is described in details with the illustration of real-life examples. It illustrates



data flow coverage would be particularly useful in software testing related to Y2K problem.

- Chapter 3 surveys the state-of-the-art coverage testing tool. The coverage testing tool for C language, ATAC, developed by Bellcore is focused. Coverage tools for COBOL in the mainframe industry is explored.
- Chapter 4 details the implementation of a coverage testing tool, ATACOBOL, proposed for COBOL language on the mainframe platform. Additional rules catering data structure is described.
- Chapter 5 provides detail evaluation of features and limitations of ATAC. It reviews experience in using ATAC and ATACOBOL, and evaluates the abilities of these tools.
- Chapter 6 shows the measurement arrangement and results in applying ATACOBOL measure to production test cases.
- Chapter 7 elaborates and discusses various topics related to the above chapters.
- Chapter 8 describes the schedule and resources of this project.
- Chapter 9 concludes the research result of this project.
- Chapter 10 provides a list of reference.
- Appendix A lists the module specifications of ATACOBOL, their functions, limitations, and file layouts.
- Appendix B gives a tutorial in applying ATACOBOL to a simple APS COBOL program.

## 2. BACKGROUND THEORY

### 2.1 The Basis of Program Testing

In this chapter we describe the theoretical background of code coverage methods. Real-life example from the author's programming experience will be cited as example for illustration. Program in this context is referred as programs written in procedural language as distinguished from logic programming language. Program testing is the most commonly used method for demonstrating that a program accomplishes its intended purpose. It involves selecting elements from the program's input domain  $D$ , executing the program  $P$  on these test cases  $T$ , and comparing the actual output with the expected output. For temporal related programs like on-line transaction keying by bank tellers,  $D$  involves the temporal domain. On this base, we assume the existence of an so-called *oracle*, i.e., some methods to determine whether or not the output produced by a program is correct.

### 2.2 The Need for Path Selection Criteria

While testing all possible inputs values would provide the most complete picture of a program's behaviour, the input domain is usually too large for exhaustive testing to be practical. On another point of view,  $T$  is generally associated with a set  $\Pi$  of paths through  $P$ 's flow graph. It means that we usually cannot exhaust all possible paths of program  $P$ . The usual procedure is to select a relatively small  $T$  which in some sense representative of entire  $D$  or implicitly all paths  $\Pi$ . Observation of the program on this subset is then used to predict its behaviour in general. Unfortunately, discovering such an ideal set of test data is in general an impossible task [Clar89].

Numbers of path selection criteria  $C$  have been proposed. The most well know of these criteria are the statement coverage and edge coverage. Weyuker *et al* proposed an family of path selection criteria that includes the control flow coverage criteria and an additional set of data flow selection criteria in terms the def-use pairs [Rap85], [Fra88].

## 2.3 Control Flow Coverage

### 2.3.1 Block Coverage and Edge Coverage

A program is a finite sequence of legal statements  $\langle s_1, \dots, s_n \rangle$ . A program can be uniquely decomposed into a set of disjoint blocks having the property that whenever the first statement of the block is executed, the other statements are executed in the given order. Formally, a *block* is a maximal set of ordered statements  $b = \langle s_1, \dots, s_b \rangle$ .

The program flow graph  $G$  (figure 2.1) representing a program  $P$  consists of one *node*  $i$  corresponding to each block  $b_i$  of  $P$  and an *edge* from to node  $j$  to node  $k$  denoted  $(j, k)$ . In the following discussion, the term *block* and *node* is of the same sense of meaning. A path  $\pi$  is a finite sequence of nodes  $\langle n_1, \dots, n_k \rangle, k \geq 2$ , such that there is an edge from  $n_i$  to  $n_{i+1}$ .

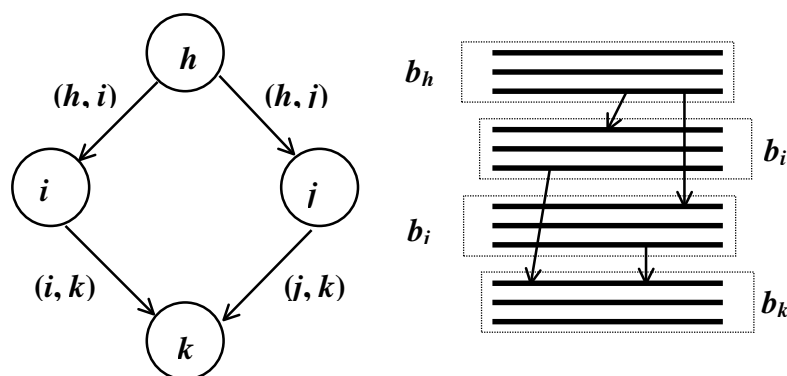
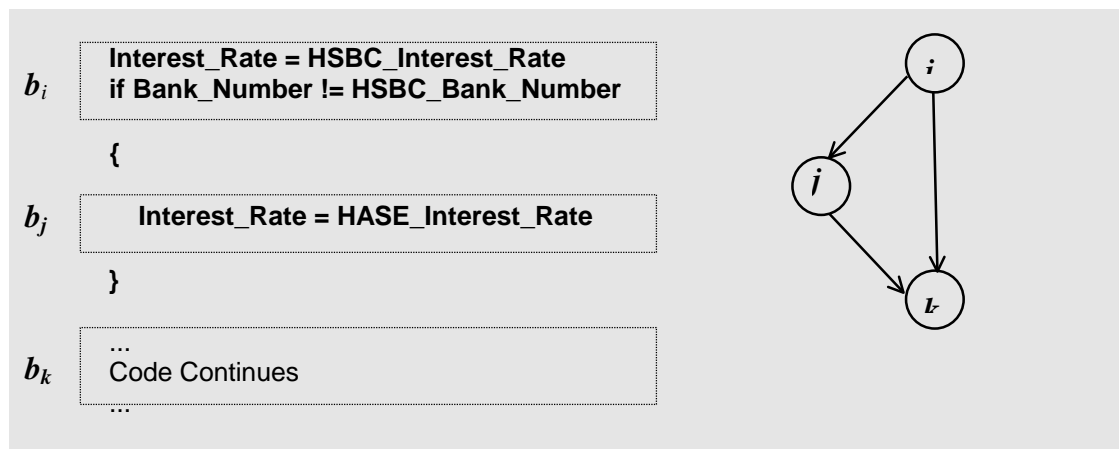


Figure 2.1 A Typical Program and Program Flow Graph

A set of paths  $\Pi$  fulfils block coverage criteria if all the nodes of the control flow graph of a program is included in  $\Pi$ .  $\Pi$  fulfils edge coverage criteria if all the edges of the control flow graph of a program is included in  $\Pi$ .

Obviously, edge coverage is a stronger criteria than statement coverage. It is because if all edges are covered, the nodes connected by the edges will also be covered. Meanwhile, the reverse is not always true, that is, all nodes covered does not implies all edges covered. We will prove that by a real-life programming example in the next section.

### 2.3.2 A Real-Life Example Demonstrating the Subsumption Relation between Coverage and Edge Coverage



**Figure 2.2** Example Program and Flow Graph for Edge Coverage

In Hong Kong SAR, the Hong Kong and Shanghai Banking Corporation Limited (HSBC), bank number 004, and Hang Seng Banking Corporation Limited (HASE), bank number 024, are all under the HSBC Group. Teller's banking terminals and Auto Teller Machines (ATMs) of both banks are all hosted by the same mainframe and mostly handled by the same program operated by HSBC. Therefore, programs usually

involves the logic to distinguish one bank from the other, so that bank specific value can be assigned to certain attributes (like interest rates, overdrawn limits, service charges, etc.) In some systems, since the transaction volume of HSBC is higher than HASE, the HSBC's corresponding value is set up as the default value for certain attributes. Program fragments like figure 2.2 occurs frequently in in-house programs.

In this program fragment:

Block coverage,  $\Pi = \{ i, j, k \}$

Edge coverage,  $\Pi = \{ (i, j), (j, k), (i, k) \}$

If we just perform the test with HASE inputs only (i.e. `Bank_Number != HSBC_Bank_Number`), block coverage is met but edge coverage is not met (i.e., only  $(i, j)$  and  $(j, k)$  are covered). However, it is uncertain whether the `Interest_Rate` is set up correctly for HSBC. To take care of both banks, we should meet the edge coverage criteria by testing also with HSBC inputs.

Criterion  $C_1$  includes criterion  $C_2$  if and only if, for every program  $P$  and any test set  $T$  which satisfies  $C_1$  also satisfies  $C_2$ . This relation is denoted by  $C_1 \Rightarrow C_2$ . This is called the notion of subsumption. Therefore, we can denote the relation between block coverage and edge coverage by: edge coverage  $\Rightarrow$  block coverage .

## 2.4 Data Flow Coverage

### 2.4.1 The Arise of Data Flow Coverage

Block and edge coverage are the well-accepted basic path selection criteria as they captures the essence of program control structure. Unluckily, these control flow coverage measures can fail to expose many common errors. We reason that, even if every statement and branch had been executed, if the result of some computation had

never be used, one would have little evidence that the intended computation had been performed. Several path selection criteria which are based on data flow analysis on the hope to “bridge the gap” between edge coverage and the almost impossible all paths coverage. Rather than selecting program paths based solely on the control structure of a program, the data flow criteria track variables through a program. A number of path selection criteria have been proposed in the literature [Las83], [Nta84], [Rap85], [Fra89]. Each of them captures some important aspect of a program’s structure. The family of data flow criteria proposed by Weyuker *et al* is considered as the most complete and systematic one [Fra93].

#### 2.4.2 Def/Use Pair

The path selection criteria proposed by Weyuker, Rapps and Frankl [Rapp85], [Fran89] are based on an investigation of the ways in which values are associated with variables, and how these associations can affect the execution of a program. This analysis focuses on the occurrences of variables within a program. Each variable occurrence is classified as being a *definitional*, *computation-use* or *predicate-use* occurrence. They are referred as *def*, *c-use*, and *p-use*, respectively.

In a program, a variable is either defined or used in some way. When a variable is assignment by certain value. We called it a def of the variable. Among uses, we can recognise two substantially different types of uses. The first directly affects the computation being performed or allows one to see the result of some earlier definition. Such type of use is called a c-use. A c-use may indirectly affect the flow of control through the program. On the other hand, the second type of use directly affects the flow of control through the program, and thereby may indirectly affect the computations performed. Such type of use is called a p-use.

### 2.4.3 The Def/Use Graph

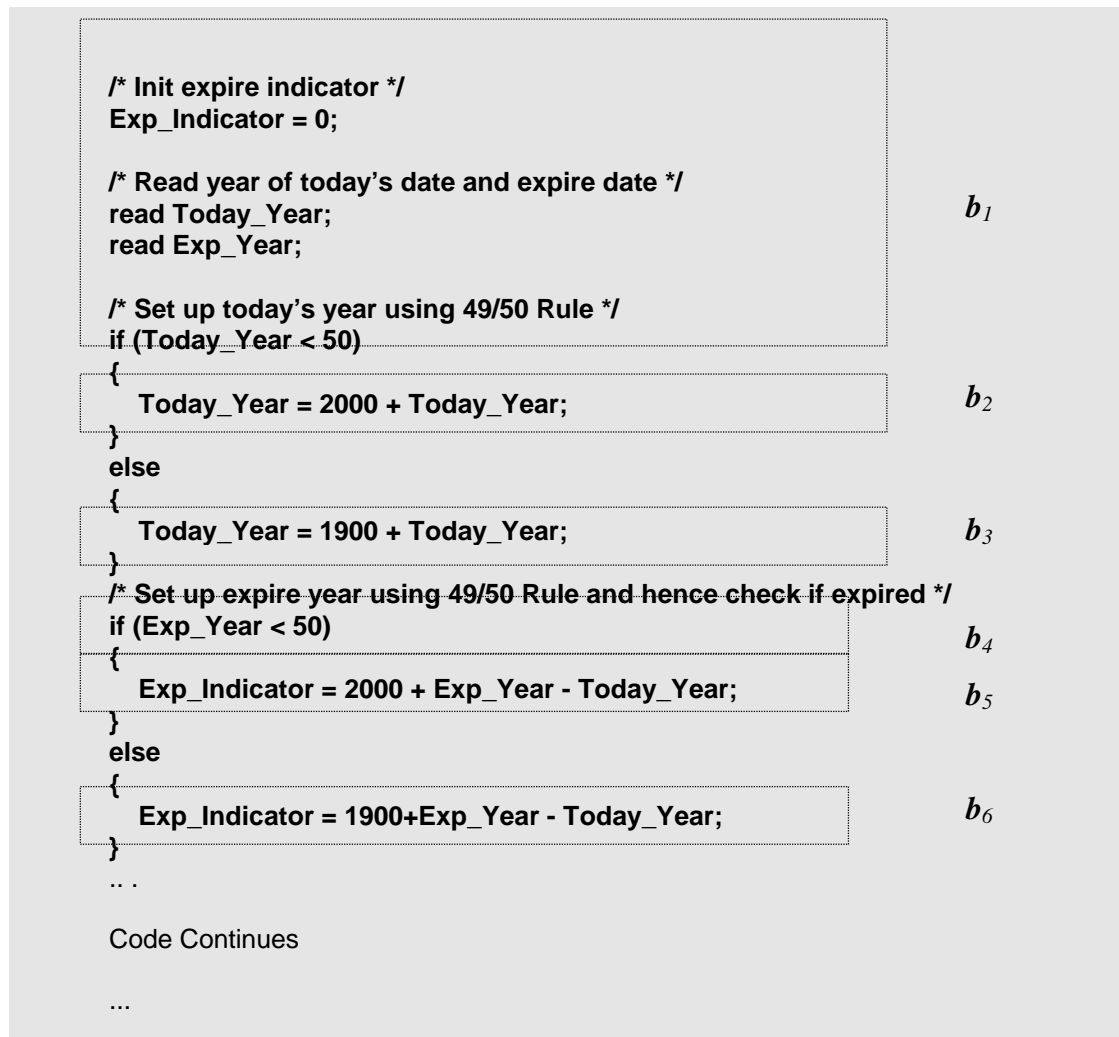
Only global definition and use are considered. Since we are interested in tracing the flow of data between nodes, any definition which is used only within the node in which the definition occurs is of little importance. A def of a variable is global if it is the last def of that variable in a node. A c-use of a variable is a global c-use, provided there is no def of that variable proceeding the c-use within the node. C-use associates with nodes. That is, a node may contain c-use of certain variables. In contrast, p-use associate with edges. P-use is by-default global since it always associates with the transfer of control between blocks.

Weyuker *et al* defines a *def/use graph* that can be created from a program by associating a set with each edge and two sets with each node.

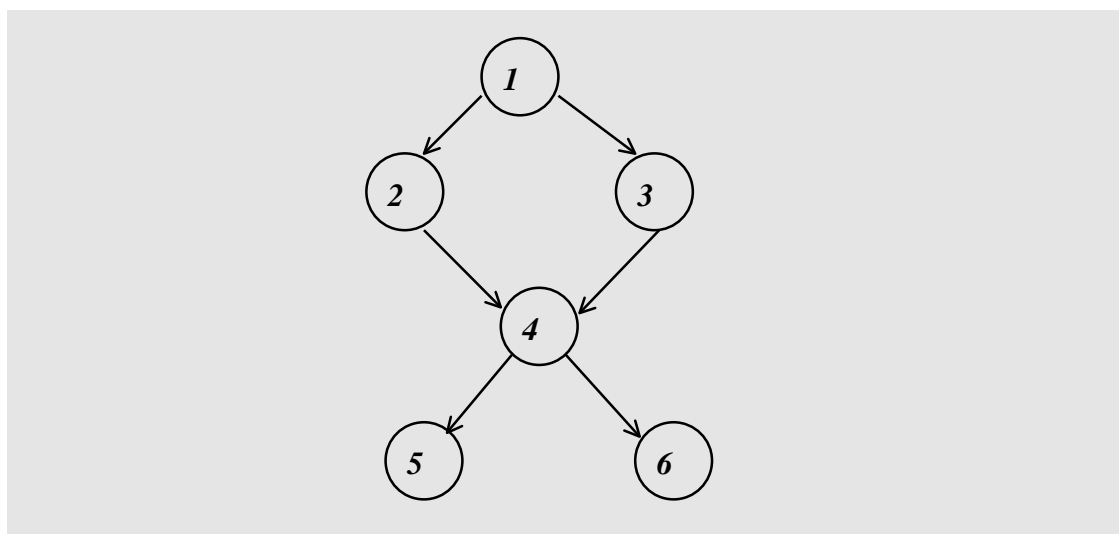
The sets of the def/use graph are as follows:

- $\text{def}(i)$  is the set of variables for which node  $i$  contains a global def.
- $\text{c-use}(i)$  is the set of variables for which node  $i$  contains a global c-use.
- $\text{p-use}(i, j)$  is the set of variables for which edge  $(i, j)$  contains a p-use.

Let us illustrate the construction of the def/use graph by another real-life example. This example is inspired from Year 2000 testing in the banking business. Some accounts opened with the banks are temporary, for example, loan accounts. These temporary account records carry expire dates. Before the manipulation of a temporary account, the account should be verified if it had been expired. Since most of the date fields in files carries only 2 digits, to cater for Y2K problem, one way is to apply the 49/50 rule to determine the century. Program like figure 2.3 may occurs. The control flow graph of this program is shown on figure 2.4.



**Figure 2.3** Example Coding to Demonstrate Data Flow Coverage



**Figure 2.4** Flow Graph of Example Coding

The def/use graph for this program fragment is as follows:



| node | c-use                  | def                                     | edge   | p-use      |
|------|------------------------|---|--------|------------|
| 1    | $\emptyset$            | Exp_indicator<br>Today_Year<br>Exp_Year | (1, 2) | Today_Year |
| 2    | Today_Year             | Today_Year                              | (1, 3) | Today_Year |
| 3    | Today_Year             | Today_Year                              | (4, 5) | Exp_Year   |
| 4    | $\emptyset$            | $\emptyset$                             | (4, 6) | Exp_Year   |
| 5    | Today_Year<br>Exp_Year | Exp_Indicator                           |        |            |
| 6    | Today_Year<br>Exp_Year | Exp_Indicator                           |        |            |

The def/use graph can be considered as the data flow graph in contrast to control flow graph. Based on this graph, two def/use pair sets is established:

Let  $i$  be any node and  $x$  any any variable such that  $x \in \text{def}(i)$ .

A path  $(i, n_1, \dots, n_m, j)$ ,  $m \geq 0$ , contains no defs of  $x$  in nodes  $n_1, \dots, n_m$ , is called a def-clear path w.r.t.  $x$  from node  $i$  to node  $j$ .

A path  $(i, n_1, \dots, n_m, j, k)$ ,  $m \geq 0$ , contains no defs of  $x$  in nodes  $n_1, \dots, n_m$ , is called a def-clear path w.r.t.  $x$  from node  $i$  to edge  $(j, k)$ .

- $\text{dcu}(x, i)$  is the set of all nodes  $j$  such that  $x \in \text{c-use}(j)$  and for which there is a def-clear path w.r.t.  $x$  from  $i$  to  $j$ .
- $\text{dpu}(x, i)$  is the set of all edges  $(j, k)$  such that  $x \in \text{p-use}(j, k)$  and for which there is a def-clear path w.r.t.  $x$  from  $i$  to  $j$ .

The dcu and dpu sets from the previous table of our Y2K example are:

$$\text{dcu}(\text{Exp\_Indicator}, 1) = \{5, 6\}$$

$$\text{dpu}(\text{Exp\_Indicator}, 1) = \{\emptyset\}$$

$$\text{dcu}(\text{Today\_Year}, 1) = \{2, 3\}$$

$$\text{dpu}(\text{Today\_Year}, 1) = \{(1, 2), (1, 3)\}$$

$$\text{dcu}(\text{Exp\_Year}, 1) = \{5, 6\}$$

$$\text{dpu}(\text{Exp\_Year}, 1) = \{(4,5), (4,6)\}$$

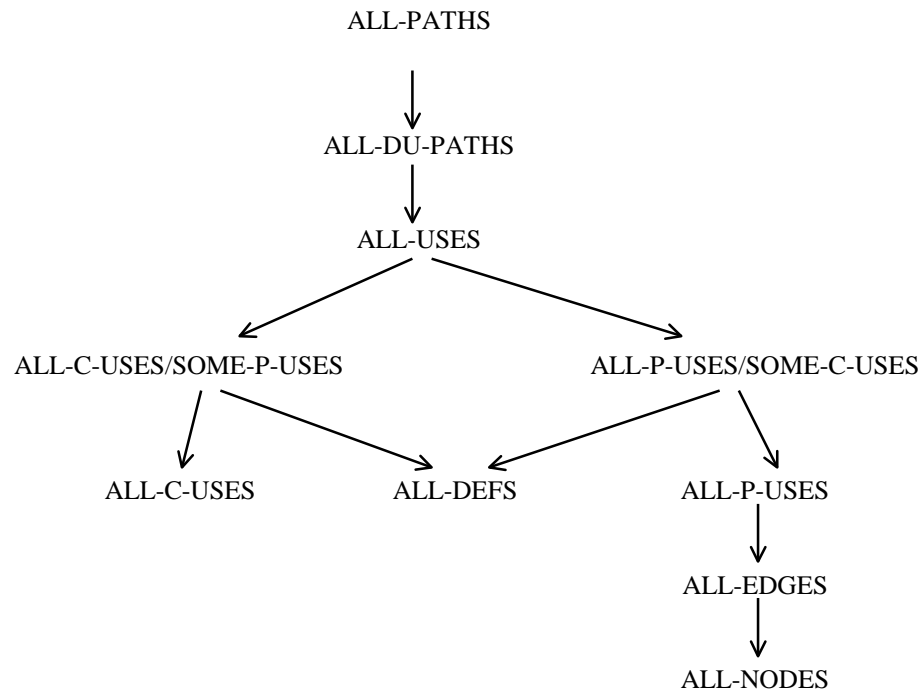
$$\begin{aligned} \text{dcu}(\text{Today\_Year}, 2) &= \{5, 6\} \\ \text{dpu}(\text{Today\_Year}, 2) &= \{\emptyset\} \\ \text{dcu}(\text{Today\_Year}, 3) &= \{5, 6\} \\ \text{dpu}(\text{Today\_Year}, 3) &= \{\emptyset\} \end{aligned}$$

#### 2.4.4 Family of Data Flow Selection Criteria

With these dcu and cpu definitions, Weyuker presents the family of data flow coverage criteria:

| Criterion              | Associations Required   |
|------------------------|---|
| All-defs               | Some $(i, j, x)$ s.t. $j \in \text{dcu}(x, i)$ or<br>some $(i, (j,k), x)$ s.t. $(j, k) \in \text{dpu}(x, i)$ .  |
| All-c-uses             | All $(i, j, x)$ s.t. $j \in \text{dcu}(x, i)$ .   |
| All-p-uses             | All $(i, (j,k), x)$ s.t. $(j, k) \in \text{dpu}(x, i)$ .  |
| All-p-uses/some-c-uses | All $(i, (j,k), x)$ s.t. $(j, k) \in \text{dpu}(x, i)$ .<br>In addition, if $\text{dpu}(x, i) = \emptyset$ then some $(i, j, x)$<br>s.t. $j \in \text{dcu}(x, i)$ .                   |
| All-c-uses/some-p-uses | All $(i, j, x)$ s.t. $j \in \text{dcu}(x, i)$ .<br>In addition, if $\text{dcu}(x, i) = \emptyset$ then some $(i, (j,k), x)$<br>s.t. $(j, k) \in \text{dpu}(x, i)$ .                   |
| All-uses               | All $(i, j, x)$ s.t. $j \in \text{dcu}(x, i)$ and<br>all $(i, (j,k), x)$ s.t. $(j, k) \in \text{dpu}(x, i)$ .   |
| All-du-paths           | All du-paths from $i$ to $j$ with respect to $x$ for each $j \in \text{dpu}(x, i)$ and all du-paths from $i$ to $(j, k)$ with respect to $x$ for each $(j, k) \in \text{dpu}(x, i)$ . |

Rapps and Weyuker [Rap84] proves the subsumption relationship of these family of data flow criteria together with control flow criteria as shown in figure 2.5.



**Figure 2.5** Control and Data Flow Coverage Subsumption

#### 2.4.5 Real-Life Y2K Example Demonstrating Advantage of Data Flow Coverage

Interested readers please refer to [Raps84] for the details of the proofs. Here we just use the Y2K example to illustrate how the all-uses criteria can be a stronger criteria than all-edges and capable to detect more faults in practice.

In the Y2K examples,

Edge coverage,  $\Pi = \{ (1, 2), (1, 3), (4, 5), (4, 6) \}$

Two complete paths  $\{1, 2, 4, 5\}$  and  $\{1, 3, 4, 6\}$  can have already fulfil the edge coverage criteria.

Semantically,  $\{1, 2, 4, 5\}$  tests the case:

**Case 1: Both Today's Date and Expire Date in 19XX.**

$\{1, 3, 4, 6\}$  tests the case:

**Case 2: Both Today's Date and Expire Date in 20XX.**

Unfortunately, two cross-century cases are not tested, they are:

**Case 3: Today's Date in 19XX and Expire Date in 20XX.****Case 4: Today's Date in 20XX and Expire Date in 19XX.**

These 2 cases should indeed be focus by the Y2K compliance test but missed out by the all-edge criteria.

On the other hand, consider the All-Uses (or just All-C-Uses in this case) criteria, for every node  $i$  and every  $x \in \text{def}(i)$ , the selected paths should include a def-clear path w.r.t.  $x$  from  $i$  to all elements of  $\text{dcu}(x, i)$ . Review the  $\text{dcu}$  of our example:

$$\text{dcu}(\text{Today\_Year}, 2) = \{5, 6\}$$

$$\text{dcu}(\text{Today\_Year}, 3) = \{5, 6\}$$

Paths from node 2 to node 5 and node also 6 is required. Paths from node 3 to node 5 and also node 6 is required. Therefore, to satisfy the All-Uses criteria, despite paths  $\{1, 2, 4, 5\}$  and  $\{1, 3, 4, 6\}$ ,  $\{1, 2, 4, 6\}$  and  $\{1, 3, 4, 5\}$  may be included. Not just the stronger of the criteria c-use is, but it really reveals the semantic of the real-world testing requirements of test cases construction:

**Case 1: Both Today's Date and Expire Date in 19XX.****Case 2: Both Today's Date and Expire Date in 20XX.****Case 3: Today's Date in 19XX and Expire Date in 20XX.****Case 4: Today's Date in 20XX and Expire Date in 19XX.**2.4.6 Extended Example that Required All-DU-Paths Criteria

The previous example requires all-uses criteria to fulfil the semantic requirement. In this example, we re-arrange the coding as shown in figure 2.6.

In this case, instead of using variable `Today_Year` in separate nodes 5 and 6, the date comparison is made centralised at node 7. In this case the all-uses criteria for both

variables Today\_Year and Expire\_Year not necessary covers all the four Y2K test case

```

/* Init expire indicator */
Exp_Indicator = 0;

/* Read year of today's date and expire date */
read Today_Year;
read Exp_Year;

/* Set up today's year using 49/50 Rule */
if (Today_Year < 50)
{
    Today_Year = 2000 + Today_Year;
}
else
{
    Today_Year = 1900 + Today_Year;
}
/* Set up expire year using 49/50 Rule and hence check if expired */
if (Exp_Year < 50)
{
    Exp_Year = 2000 + Exp_Year;
}
else
{
    Exp_Year = 1900+Exp_Year;
}

Exp_Indicator = Exp_Year - Today_Year;

...

Code Continues

...

```

*b*<sub>1</sub>

*b*<sub>2</sub>

*b*<sub>3</sub>

*b*<sub>4</sub>

*b*<sub>5</sub>

*b*<sub>6</sub>

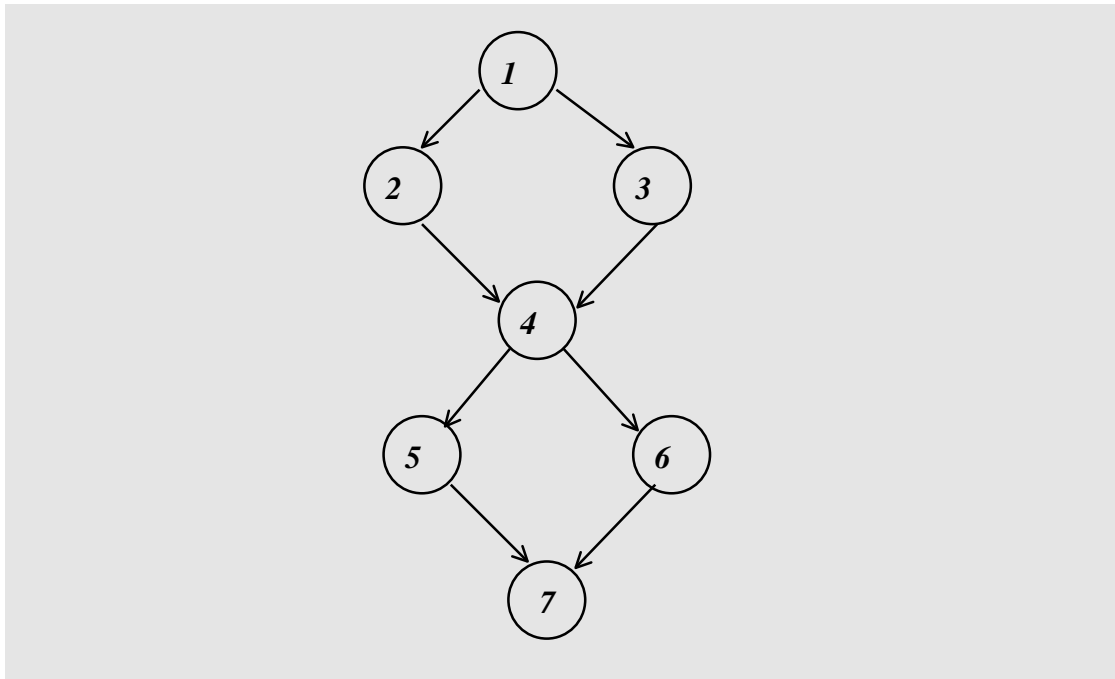
*b*<sub>7</sub>

**Figure 2.6** Another Example Coding to Demonstrate All-DU-Paths Criteria

The def/use graph for this program fragment is as follows:

| node | c-use      | def                                     | edge   | p-use      |
|------|------------|---|--------|------------|
| 1    | ∅          | Exp_indicator<br>Today_Year<br>Exp_Year | (1, 2) | Today_Year |
| 2    | Today_Year | Today_Year                              | (1, 3) | Today_Year |
| 3    | Today_Year | Today_Year                              | (4, 5) | Exp_Year   |
| 4    | ∅          | ∅                                       | (4, 6) | Exp_Year   |
| 5    | Exp_Year   | Exp_Year                                |        |            |
| 6    | Exp_Year   | Exp_Year                                |        |            |
| 7    | Today_Year | Exp_indicator                           |        |            |

|          |  |  |  |
|----------|--|--|--|
| Exp_year |  |  |  |
|----------|--|--|--|



**Figure 2.7** Flow Graph of Another Example Coding

The dcu and dpu sets for this Y2K example are:

|  |   |                      |
|--|---|----------------------|
| $\text{dcu}(\text{Exp\_Indicator}, 1)$ | = | $\{7\}$              |
| $\text{dpu}(\text{Exp\_Indicator}, 1)$ | = | $\{\emptyset\}$      |
| $\text{dcu}(\text{Today\_Year}, 1)$    | = | $\{2, 3\}$           |
| $\text{dpu}(\text{Today\_Year}, 1)$    | = | $\{(1, 2), (1, 3)\}$ |
| $\text{dcu}(\text{Exp\_Year}, 1)$      | = | $\{5, 6\}$           |
| $\text{dpu}(\text{Exp\_Year}, 1)$      | = | $\{(4, 5), (4, 6)\}$ |
| $\text{dcu}(\text{Today\_Year}, 2)$    | = | $\{7\}$              |
| $\text{dpu}(\text{Today\_Year}, 2)$    | = | $\{\emptyset\}$      |
| $\text{dcu}(\text{Today\_Year}, 3)$    | = | $\{7\}$              |
| $\text{dpu}(\text{Today\_Year}, 3)$    | = | $\{\emptyset\}$      |
| $\text{dcu}(\text{Exp\_Year}, 5)$      | = | $\{7\}$              |
| $\text{dpu}(\text{Exp\_Year}, 5)$      | = | $\{\emptyset\}$      |

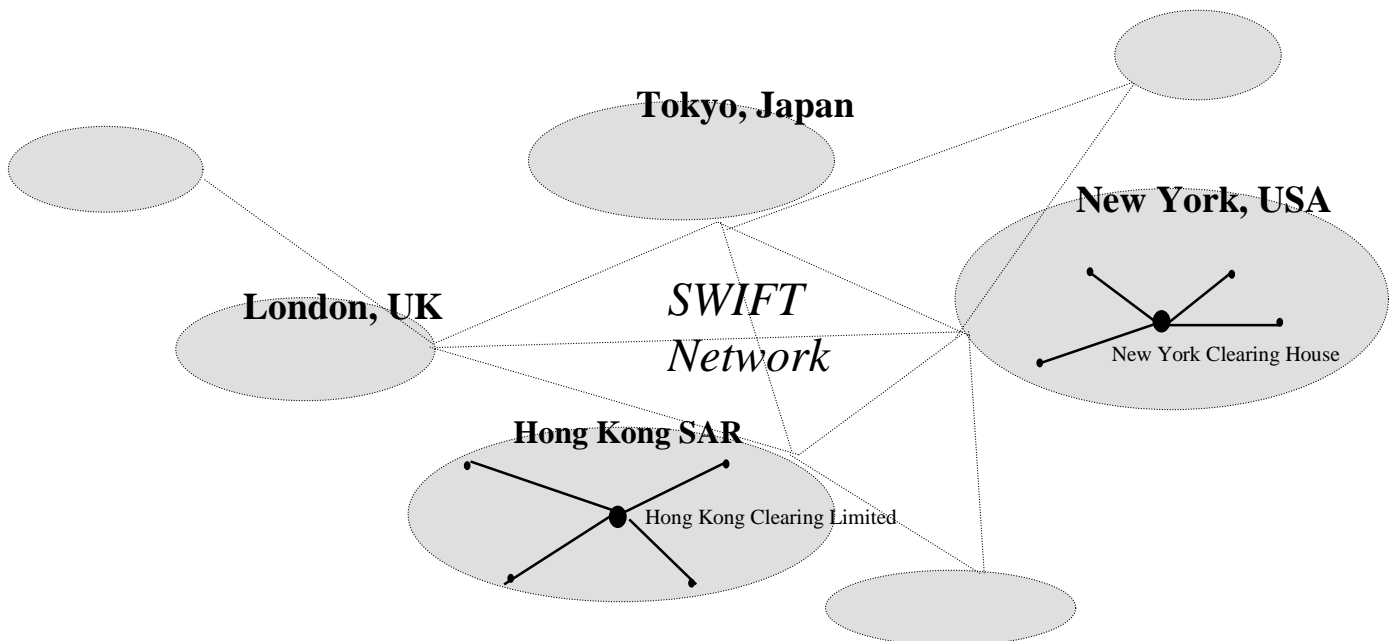
$$\text{dcu}(\text{Exp\_Year}, 6) = \{7\}$$

$$\text{dpu}(\text{Exp\_Year}, 6) = \{\emptyset\}$$

Consider the all-uses criteria, Today\_Year defined in node 2 and node 3 only need to be used in node 7 through any path, and Exp\_Year defined in node 5 and 6 also only need to be used in node 7. If these two paths coincides, only two test cases out of the four Y2K real-world test cases is executed to fulfil all-uses criterion. Here, it remains the all-du-paths criterion as the only promising criterion to enable all four cases to be tested in this example. The all-du-paths criteria requires all paths between the def-use of a variable to be executed at least once.

#### 2.4.6 Global Payments Systems Test

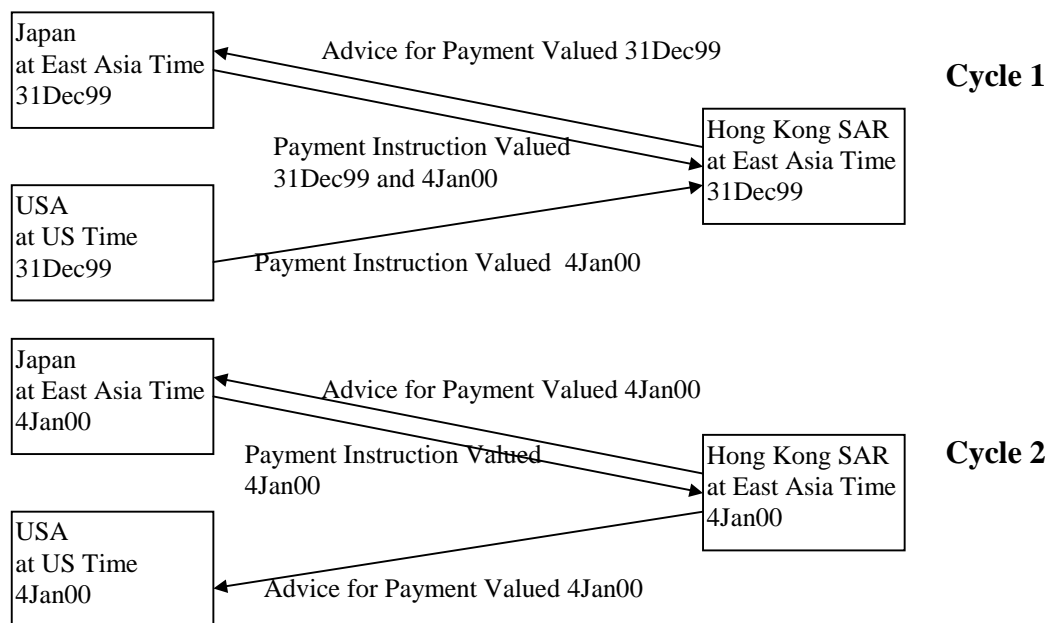
From the author's experience in a global banking industry wide Y2K test, it shows the Y2K coverage criteria discussed previously would possibly applicable.



**Figure 2.8** Global Payments Systems Test Simplified Connection Scenario

The Global Payments Systems Test for Y2K was being organised by the New York Clearing House. This test serves as the global financial industry Year 2000 test to validate the payment systems of banking sectors across markets. 190 financial institutes of 20 markets/countries participated in the test. Participant may send and receive payment instructions with its partners within same time zone or cross time zone. Hong Kong SAR participated the test on 12 June 1999 (Cycle 1) and 13 June 1999 (Cycle 2) with the cycle system date set to 31 December 1999 and 4 January 2000 respectively [ICL99].

The purpose of the testing is to draw the attention of international community on the importance of the Y2K issue and make sure the payment infrastructure will be functioning properly through 2000. The test involves date comparison handling mechanism of payment instructions, it aligns with the four test cases that can be detected by the data flow coverage criteria discussed previously. The payment instruction handling scenario of Hong Kong SAR is illustrated in figure 2.9.



**Figure 2.9** Global Payments Systems Test scenario for Hong Kong SAR



### 3. SURVEY ON COVERAGE TESTING TOOLS

We survey currently available coverage testing tools. A state-of-the-art coverage testing tool called ATAC is studied and evaluated. In specific, we also surveys coverage tools available for COBOL language in IBM mainframe platforms. Comparison is made between these coverage tools. From this survey, we found that the mainframe industry lacks the coverage tools that is comparable to ATAC.

#### 3.1 ATAC (Automatic Test Analysis for C)

ATAC was developed and used as a research instrument at Purdue University and Bellcore [Hor90]. It has been applied in two real-world projects called the University of Iowa/Rockwell Joint Project and the Bellcore Project described in [Hor94]. ATAC was commercialised as  $\chi$ ATAC in the Software Understanding System package, *xSuds*. *xSuds* technology can be licensed from Bellcore or from IBM. IBM sells the technology as the IBM C and C++ Maintenance and Test Toolsuite. The supported platforms covers HPUX, AIX, Solaris, Linux and Windows 95/NT with IBM's VisualAge C++ and Windows 95/NT with Microsoft Visual C++.

In this project, ATAC release 3.3.13 was compiled and executed on SUN Solaris 2.5.1. We can identify the following application of ATAC in the software testing process:

- measuring test set completeness by control and data flow coverage;
- displaying non-covered code to aid in test cases creation;
- reducing regression test set size by determining minimal test set out of tested cases.

Potentially, it can be applied to select effective randomly generated test cases. Using ATAC, the tester is required to compile the program to be tested by the ATAC pre-processor. The testing process can then be carried out as normal. The coverage measurement process is nearly transparent for the tester.

At any time of the testing, ATAC can display summary of the coverage and uncovered codes. It can also determine minimal test set size for optimal coverage. Therefore, the selected minimal test set can be used for regression test to minimise test cost. Detail evaluation and experience of using ATAC is presented on chapter 6.

## **3.2 Survey on Coverage Tools for COBOL in Mainframe Platform**

### 3.2.1 Status of COBOL in the Mainframe Platform

COBOL is the major high level language employed in IBM OS/390, MVS, VM mainframe environment. COBOL language is still an strategic and supported product of IBM mainframes. Versions of COBOL emerges continuously in the mainframe industry. In HSBC mainframe environment, more than 80% of in-house programs are developed by COBOL. Many organisations, especially the business sector, have millions of dollars invested in COBOL-based systems and in the COBOL programmers who create and maintain the applications. COBOL applications are performing mission critical applications in the business world that the users don't really want to retire them.

A large number of users/programmer are very pleased with their COBOL applications, except that they simply want to move them to open systems or client/server architectures.

In many cases, rewrite programs in other languages is costly and risky. Modernising a COBOL application is often the alternative with the least cost, least lead time, and least risk. Many COBOL developer like IBM, CA, ACUCORP and Intersolv has invented new versions of COBOL, modernising to a COBOL that supports an open system, client/server configuration might take relatively little time and effort.

In conclusion, we believe that COBOL will last long in the mainframe platform. Renew and to import new technologies to the mainframe COBOL programming environment is necessary and rewarding.

### 3.2.2 COBOL Coverage Tools on Mainframe

In this survey, COBOL coverage products of four major software vendors of the mainframe industry is selected for evaluation. Table 3.1 compares the features of these tools. *Paragraph* in COBOL is similar to *function* in C language. Paragraph coverage directs the tester to construct test cases that each paragraph in COBOL to be covered at least once.

From the table, we can identify that the IBM Code Assistant possesses the most complete features. Additional features like visual aid and tracing of specified coding various from one products to the other. CA-TestCoverage and IBM Code Assistant executes program under normal execution environment to take coverage measurement while SMARTTEST and XPEDITER requires the measurement to be taken on dedicated debugging environment.

In comparison with ATAC, IBM Code Assistant is able to provide advanced features like test set minimisation. XPEDITER of Compuware makes use of PC platform to present the result a graphical and user-friendly way. In common, none of them supports data flow coverage measurement.

From this product survey, we notice that the mainframe industry still lacks a software testing tools making use of the data coverage technique. In view of that, we propose to design and implement a coverage testing tool ATACOBOL (Automatic Test Analysis for COBOL) for the mainframe COBOL program development similar to ATAC.

|  |  |  |  |  |
|--|--|--|--|--|
| <b>Vendor</b>                            | Computer Associate (CA)  | VIASOFT  | Compuware  | IBM  |
| <b>Product Name</b>                      | CA-TestCoverage / 2000   | VIA/SmartTest with Test Coverage Analysis (TCA) option       | XPEDITER/ Code Coverage  | Coverage Assistant (CA)  |
| <b>Execution Platform</b>                | MVS, OS/390  | MVS, OS/390  | MVS, OS/390 and Microsoft Windows 3.X /95 (for viewing result)   | MVS, OS/390  |
| <b>Coverage Measurement Environment</b>  | Program compiled by specific compiler and executed normally                    | Testing program loaded under the debug environment SmartTest | Testing program loaded under the debug environment XPEDITER  | Program compiled by specific compiler and executed normally                                    |
| <b>Block Coverage Measurement</b>        | Paragraph coverage   | Statement coverage, paragraph coverage                       | Statement coverage, paragraph coverage   | Statement coverage   |
| <b>Control Flow Coverage Measurement</b> | N/A  | N/A  | Edge coverage  | Edge coverage  |
| <b>Data Flow Coverage Measurement</b>    | N/A  | N/A  | N/A  | N/A  |
| <b>Code execution count</b>              | Present  | Present  | Present  | N/A  |
| <b>Test Set Minimization</b>             | N/A  | N/A  | N/A  | Present by complementary using Distillation Assistance (DA) under the same software package    |
| <b>Test cost evaluation</b>              | N/A  | N/A  | N/A  | Execution time measured is considered as the cost  |
| <b>Trace changed coding</b>              | N/A  | N/A  | Present manually: Segments of program can be highlighted to trace  | Present by complementary using of Source Audit Assistant (SAA) under the same software package |
| <b>Visual aid</b>                        | - Summary statistics report<br><br>-Source Map Report (Indexed program source) | -Summary statistics report                                   | -High-level, system-level graphical structural chart for IT manager<br><br>-Colored code to indicate branches and complexity | -Summary statistics report<br><br>-Targeted Coverage Report<br><br>-Annotated Listing Report   |
| <b>Other features</b>                    | Compare difference of execution counts for different test cases                | N/A  | Advise risk degree of a program based on the coverage, execution count, verb types and McCabe metric                         | Execution time is measured   |

Table 3.1 Mainframe COBOL Coverage Testing Tools

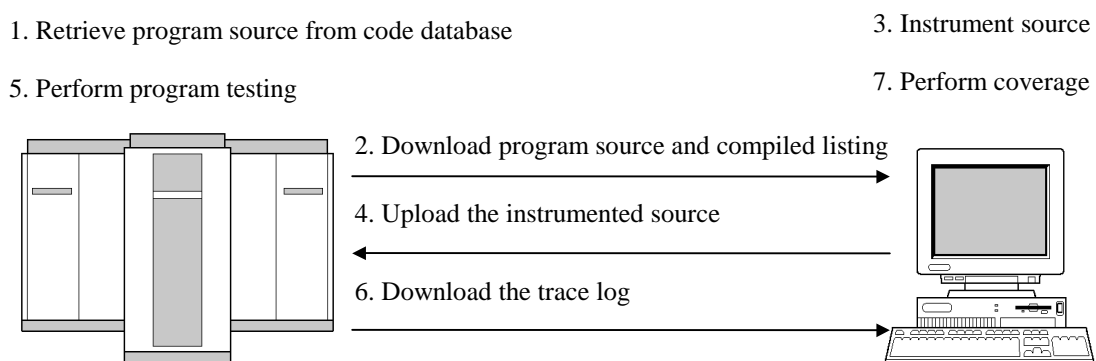
## 4. IMPLEMENTATION

### 4.1 Overview

The instrumentation, coverage measurement and analysis of ATACOBOL is implemented across mainframe and PC platforms. The ATACOBOL instrumentation and analysis program tools are written in C language using the Microsoft Visual C++ 6.0 Compiler. A version of COBOL language called the S-COBOL (structured COBOL) is selected as the target language for analysis. The S-COBOL is introduced to HSBC since 1986 and all batch programs since then are recommended to be written in S-COBOL.

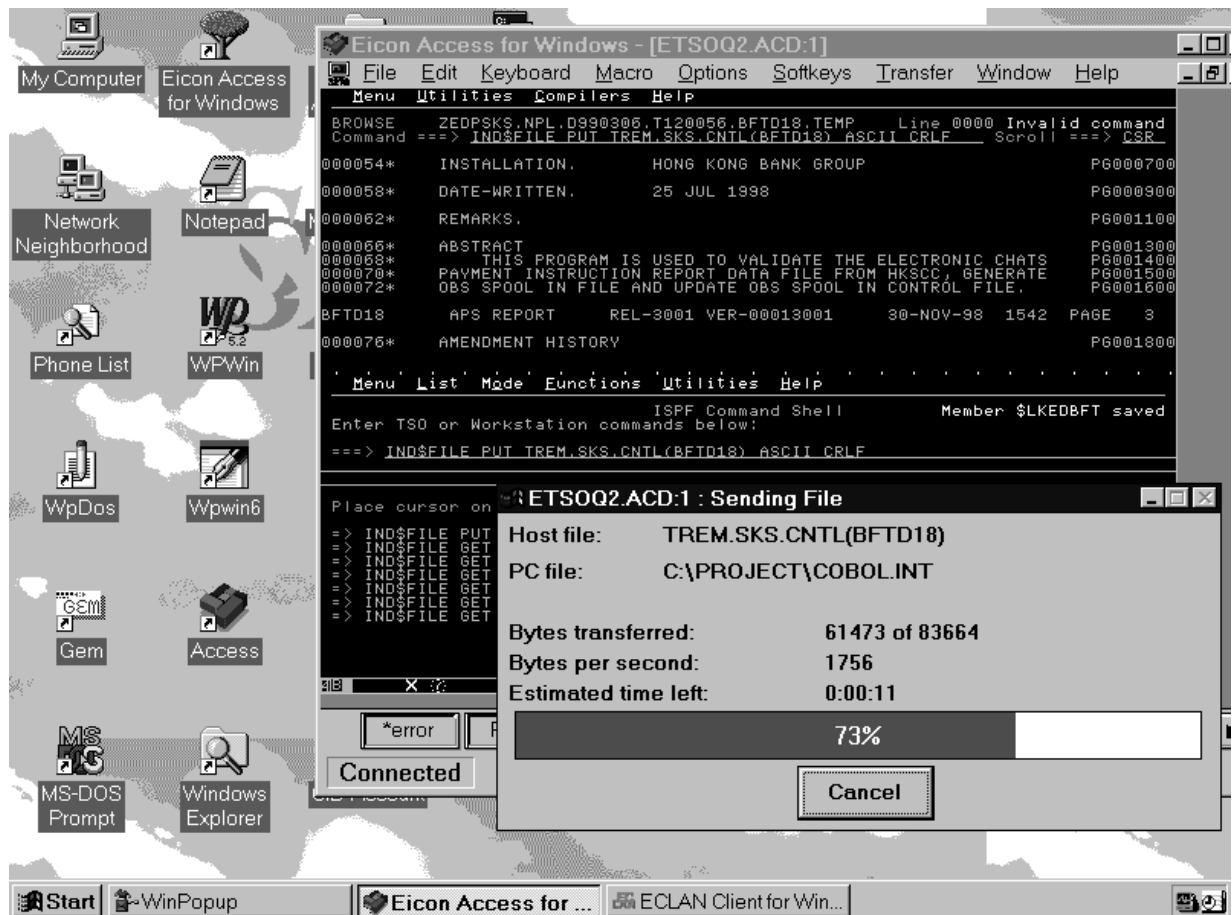
### 4.2 Environment Setup

Coverage measurement of ATACOBOL is currently achieved across IBM OS/390 and Microsoft Windows 95 by the aid of file transfer. The process is illustrated in figure 4.1 and screen capture of the file transfer is shown in figure 4.2.



**Figure 4.1** Environment Setup for ATACOBOL

In the HSBC mainframe, all in-house program sources are stored in program version control system, Edevour. We can retrieve any version of the program form Edevour.



**Figure 4.2** Screen Capture of File Transfer

### 4.3 APS COBOL

ATACOBOL supports APS (Application Productivity System) Development Centre is a cobol code generator developed and supported by Intersolv (previously called SAGE software) in USA and represented by Sumisho Electronics Limited (Japan) in Asia. The product was first delivered in 1984.

APS Development Centre is widely applied in HSBC mainframe program development. It supports generation of ANSI COBOL (or COBOL II) codes for VSAM, CICS, IDMS and DB2. This project only concentrates on pure batch programs.

The COBOL language used in APS, namely S-COBOL (Structured COBOL), is an extension of ANSI COBOL with the following changes:

- Indentation, not punctuation, controls program logic;
- One verb per line;
- Entry to and exit from program code occurs only in the root paragraph;
- Disallow falling through across paragraph.
- “GOTO”, “ALTER” and “PERFORM ... THRU” statements are not supported, that is, all S-COBOL performs are implicitly PERFORM-THRU.

In short, it avoids the irregularity of native COBOL and enforces structured programming.

#### **4.4 ATACOBOL Architecture**

The use of ATACOBOL involves 3 phases consequently:

- **Instrumentation Phase**  
The S-COBOL source is instrumented according to the structural information extracted from the source and compiled listing.
- **Testing Phase**  
The instrumented source is compiled and testing is carried out as usual. Program execution is traced automatically.
- **Analysis Phase**  
The trace log is analysed to take coverage measurement.

ATACOBOL is composed by 4 major components:

- Code Parser
- Code Instrumenter



- Coverage Analyser
- Runtime Trace Module

These components work co-operatively to perform coverage measurement as illustrated in Figure 4.5.

**Program Source**

```

/* *****
/* SUBROUTINE - CHECK WHOLE BCAMFI ROUTINE
SKIP1
/* ENTRY POINT - CHK-WHOLE-BCAMFI-RTN
SKIP1
/* CHECK ALL RECORDS IN BCAMFI AND PRINT BFT ELECTRONIC
/* CHATS PAYMENT INSTRUCTION FILE REPORT
/* *****
SKIP3
PARA CHK-WHOLE-BCASFI-RTN
SKIP1
OPEN INPUT BCASFI
SKIP1
INITIATE PRINT1-REPORT
SKIP1
PERFORM READ-BCASFI-RTN
SKIP1
IF BCASFI=EOF
UTL167-TIME = TIME-OF-DAY
UTL167-TEXT = MSG-1
DISPLAY MSG-1 UPON SYSOUT
CALL 'UTL167' USING UTL167-MSG
CALL 'CBCANCEL'
SKIP1
PERFORM EDIT-BCASFI-CNTL-RTN
SKIP1
REPEAT
PERFORM READ-BCASFI-RTN
UNTIL BCASFI=EOF
PERFORM VALIDATE-BCASFI-REC-RTN
SKIP1
TERMINATE PRINT1-REPORT
SKIP1
IF BCASFI-INVALID
UTL167-TIME = TIME-OF-DAY
UTL167-TEXT = MSG-2
DISPLAY MSG-2 UPON SYSOUT
CALL 'UTL167' USING UTL167-MSG
CALL 'CBCANCEL'
SKIP1
IF BCASFI-TRLR-NOT-READ
UTL167-TIME = TIME-OF-DAY
UTL167-TEXT = MSG-3
DISPLAY MSG-3 UPON SYSOUT
CALL 'UTL167' USING UTL167-MSG
CALL 'CBCANCEL'
SKIP1
CLOSE BCASFI
EJECT
    
```

**Control Flow Information File**

| Para Num | Block Num | Level Num | Block Type | Source Line Num |
|----------|-----------|-----------|------------|-----------------|
| 5        | 1         | 1         | P          | 601             |
| 5        | 2         | 2         | M          | 603             |
| 5        | 3         | 2         | I          | 609             |
| 5        | 4         | 3         | M          | 610             |
| 5        | 5         | 2         | M          | 616             |
| 5        | 6         | 2         | R          | 618             |
| 5        | 7         | 3         | M          | 619             |
| 5        | 8         | 2         | U          | 620             |
| 5        | 9         | 3         | M          | 621             |
| 5        | 10        | 2         | M          | 623             |
| 5        | 11        | 2         | I          | 625             |
| 5        | 12        | 3         | M          | 626             |
| 5        | 13        | 2         | I          | 632             |
| 5        | 14        | 3         | M          | 633             |
| 5        | 15        | 2         | M          | 639             |
| 5        | 16        | 1         | Q          | 649             |

**Remark:**  
 For Block Type, 'P' - start of paragraph  
 'Q' - end of paragraph  
 'M' - normal statement  
 'I' - IF statement  
 'R' - REPEAT  
 'U' - UNTIL

**Figure 4.3** S-COBOL Source and its Corresponding Record in Control Flow File

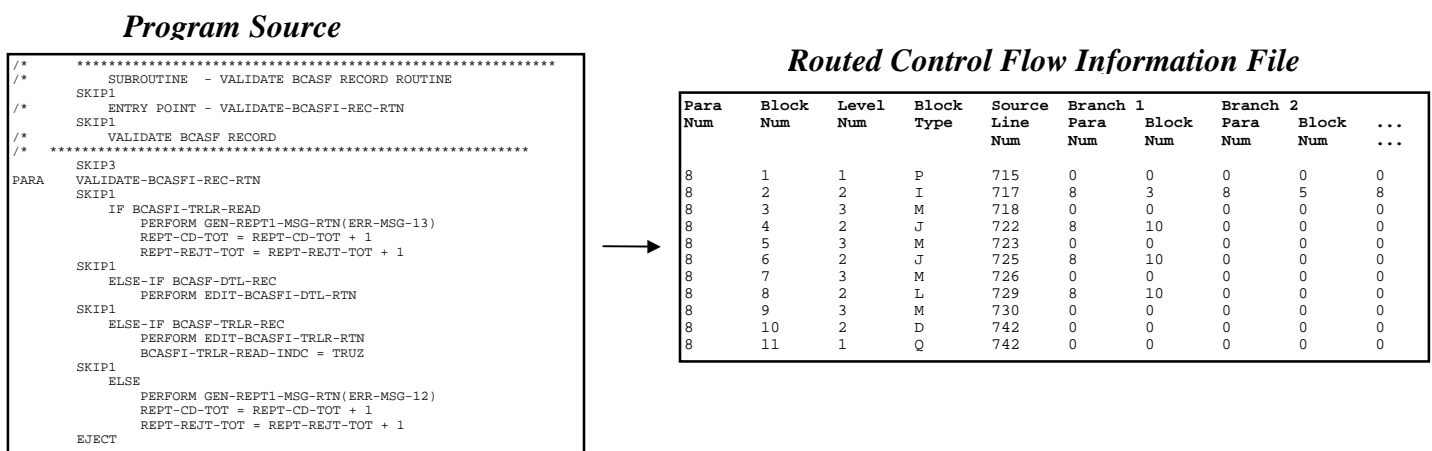
**4.5 ATACOBOL Code Parser**

The ATACOBOL Code Parser analyses S-COBOL source code and produces two files: **Control Flow Information File** and **Data Flow Information File**. In developing the parser, some techniques used by classic compilers [Aho86] are applied while shortcuts that takes advantages from the features of S-COBOL is also

considered. S-COBOL uses indentation, not punctuation, controls program logic, therefore, blocks can be parsed easily by the indentation of statements. Control Flow Information File contains control flow information about the source program for use by the ATACOBOL Instrumenter. Both files are employed in the analysis phase.

To build the Control Flow Information File, the S-COBOL source is firstly parsed into blocks. Each record in the file exactly represents a block. A record also contains additional information about the characteristics of that block including the block type and the cross reference to line number in the source.

For S-COBOL, the program flow is represented by indentation instead of punctuation and thus making it very easy to parse the source code into block. Figure 4.3 shows how a typical paragraph of S-COBOL program source is parsed into primitive records in the control flow information file. Subsequently, the records are scanned to create links from a node to the other to represent decision branches. As a result, the source program is digested to a node-link structure stored in the Control Flow Information File. Figure 4.4 shows an typical example of part of the routed Control Flow Information File extracted form a source program.



**Figure 4.5** Program Source and Routed Record in Control Flow Information File

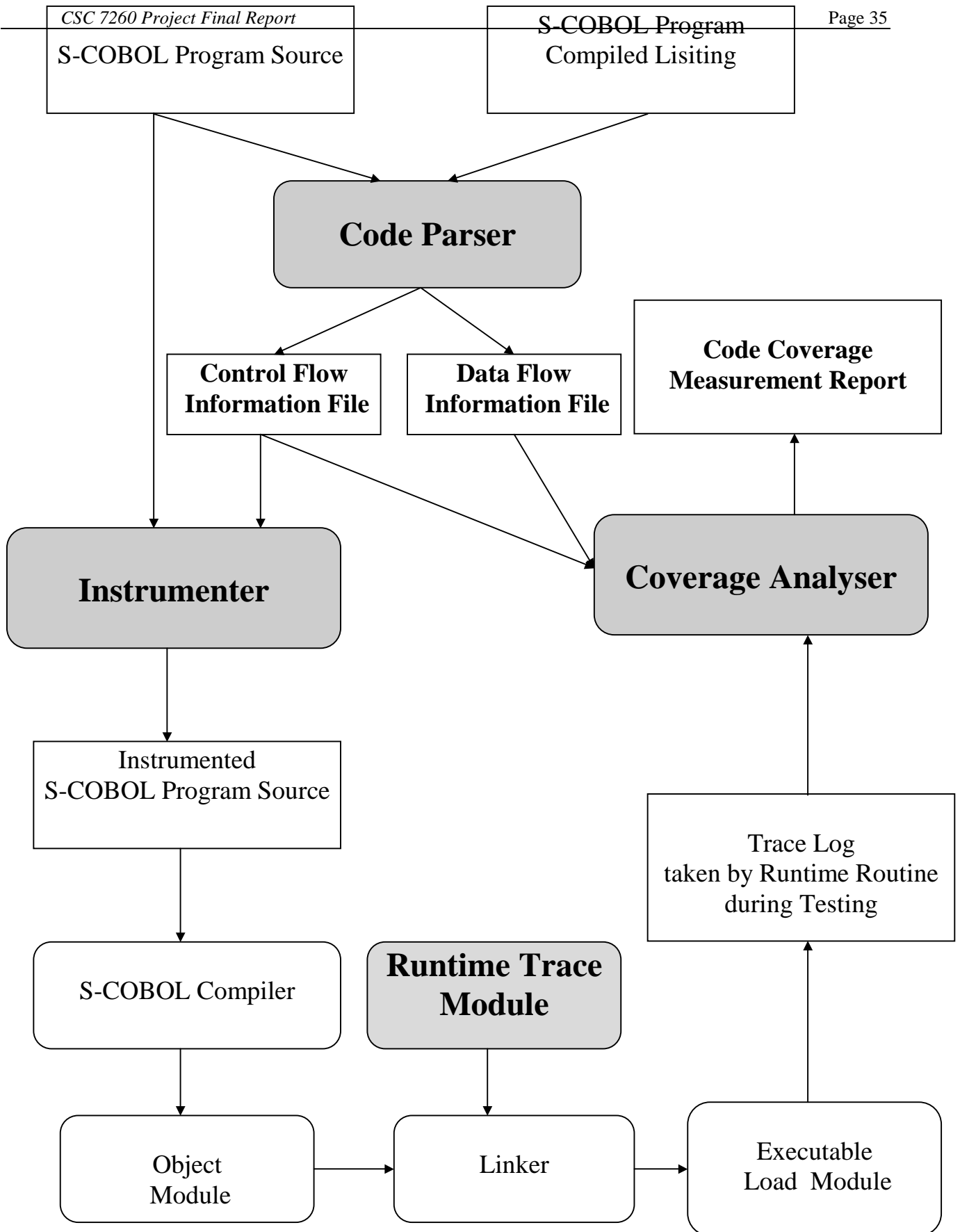
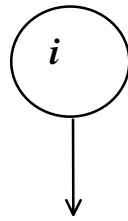
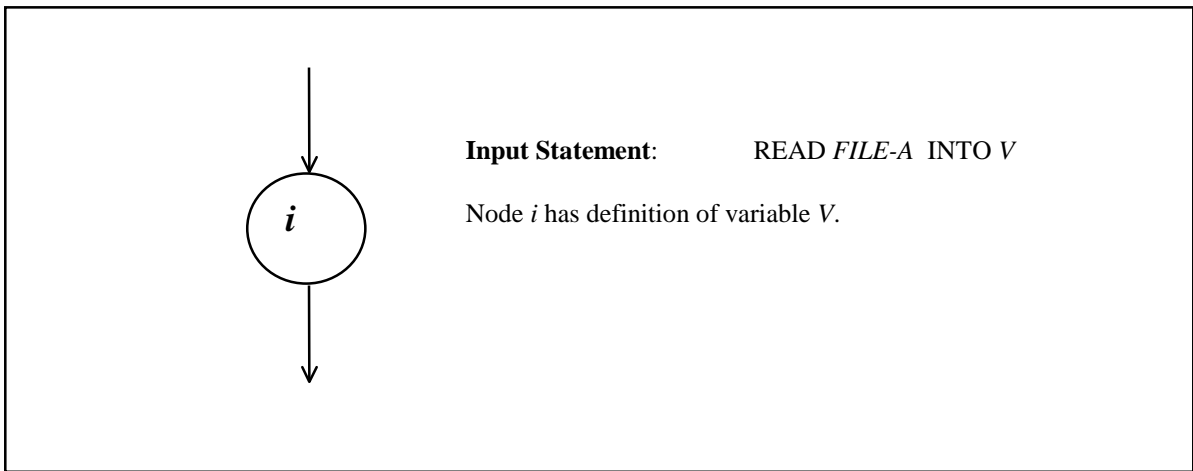


Figure 4.5 ATACOBOL Architecture

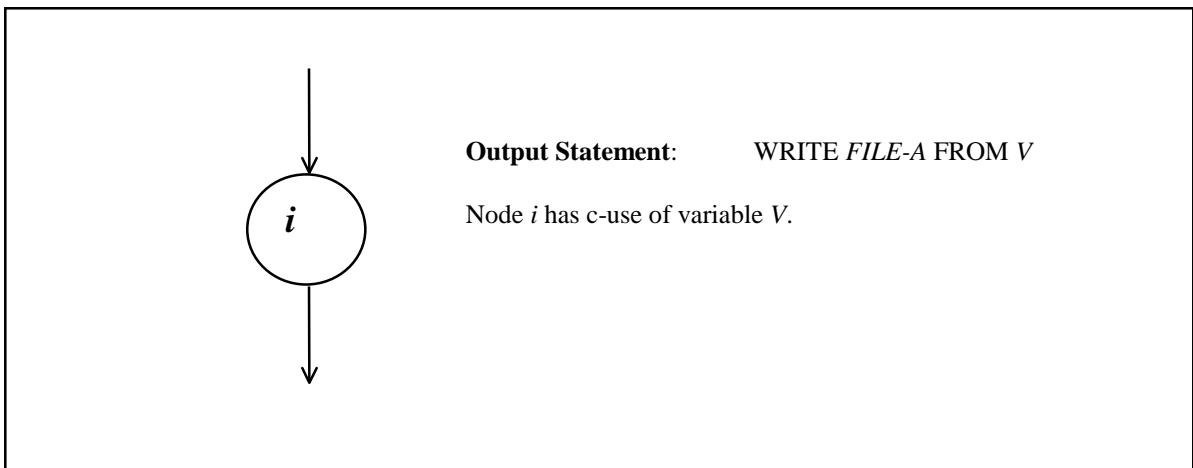
Assignment Statement:  $V = Expression$  OR  
MOVE A to V



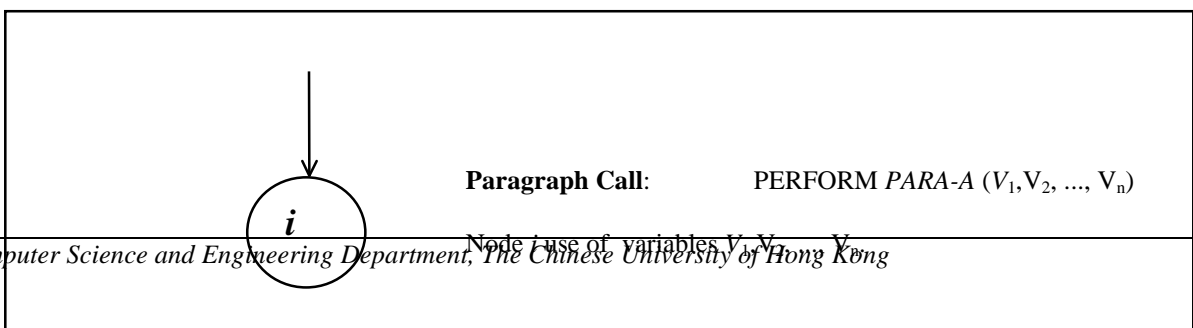
**Figure 4.6a** Assignment Statement



**Figure 4.6b** Input Statement

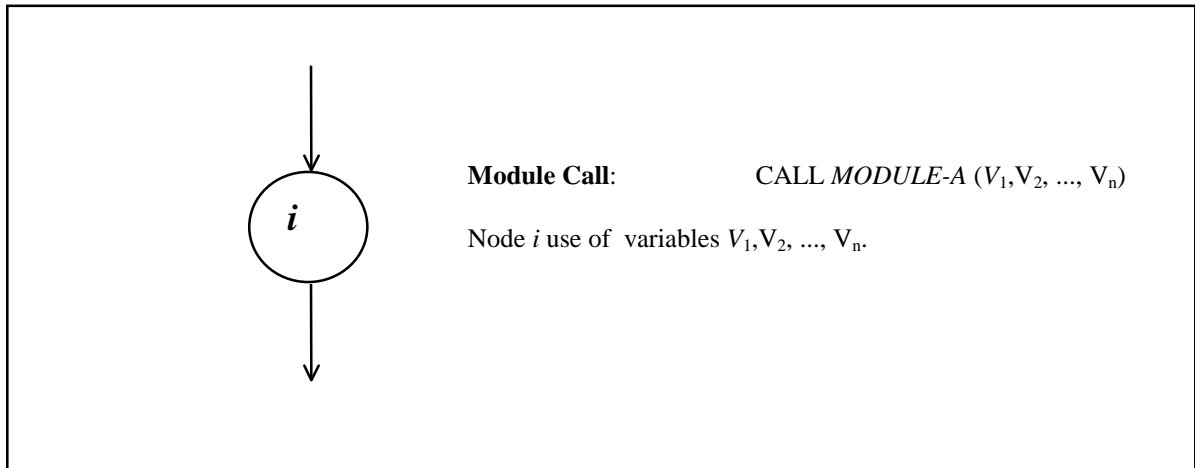


**Figure 4.6c** Output Statement

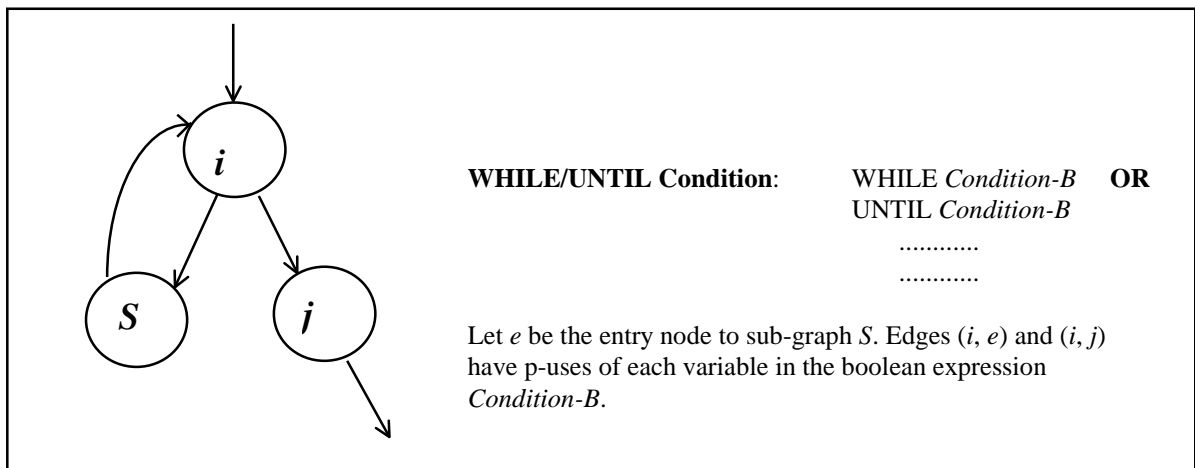




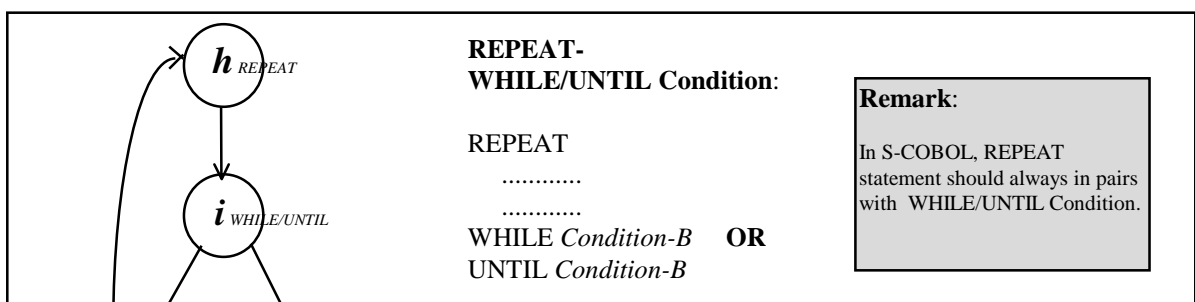
**Figure 4.6d** Paragraph Call

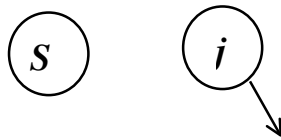


**Figure 4.6e** Module Call

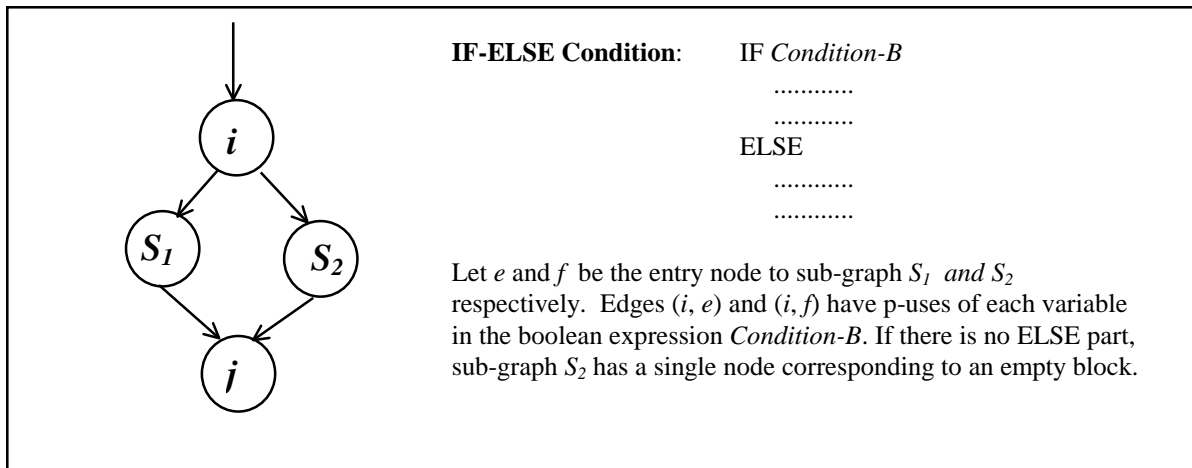


**Figure 4.6f** WHILE/UNTIL Condition

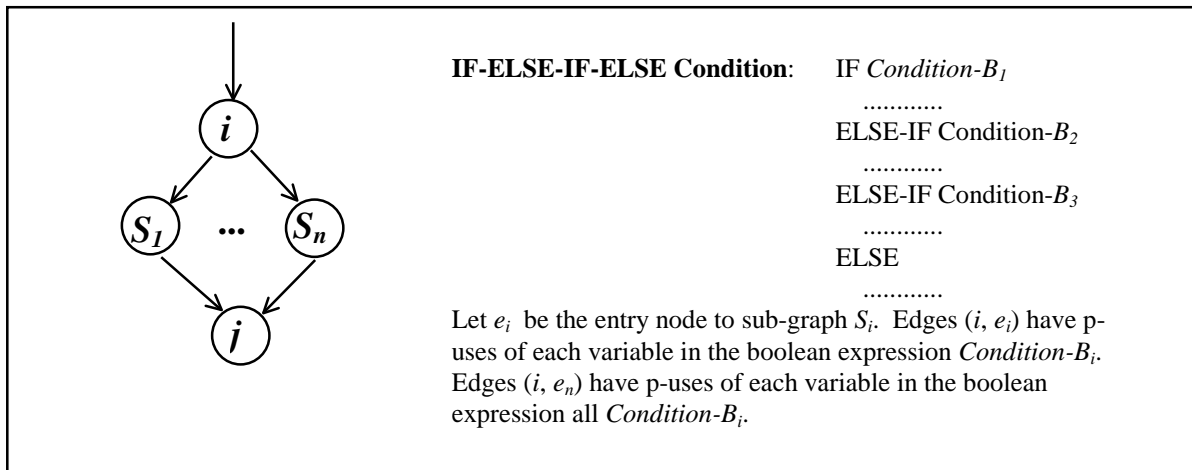




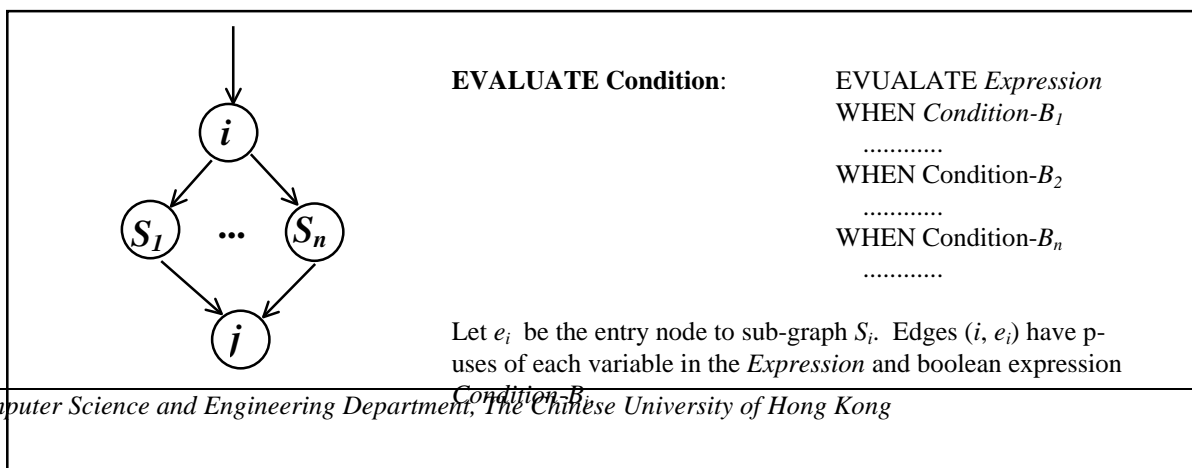
**Figure 4.6g** REPEAT-WHILE/UNTIL Condition



**Figure 4.6f** IF Condition



**Figure 4.6g** IF-ELSE-IF-ELSE Condition



**Figure 4.6h** EVALUATE Condition

**Figure 4.6** Control Flow and Data Flow Relationship for S-COBOL Language

To build the Data Flow Information File, macros and variable defined in the program listing is parsed to form a variable table as illustrated in figure 4.7.

**Program Compiled Listing**

```

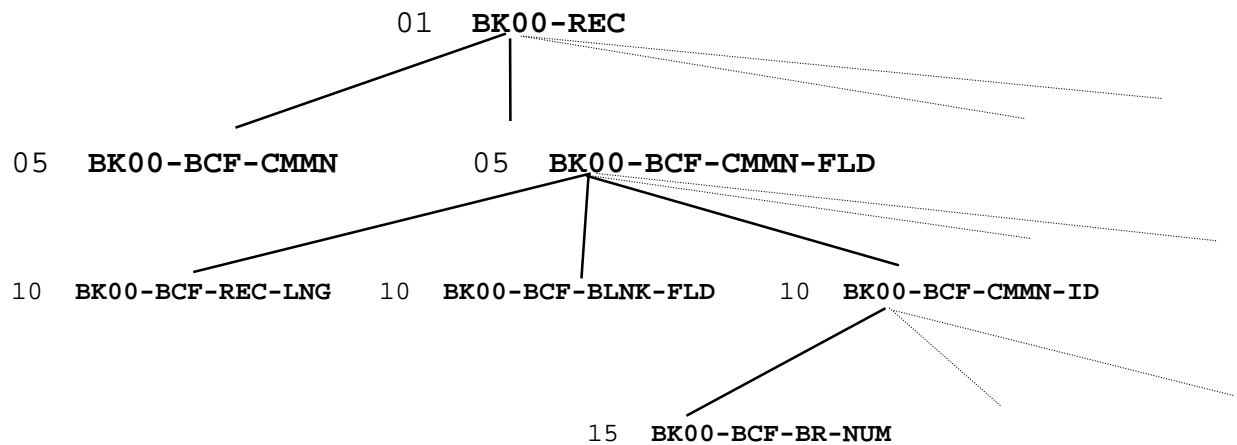
000570*01 BK00-REC          @COPY BK00W.          PG022600
01 BK00-REC.
*          BK00          BK00          BNK00          00000010
*          BANK CONTROL RECORD 00          00000020
*          DMR          00000030
*          05 BK00-BCF-CMMN          PIC          X(11).          00000040
*          CIF-BCF-CMMN          00000050
*          CIF BCF COMMON FIELDS          00000060
*          DMR          00000070
*          05 BK00-BCF-CMMN-FLD          REDEFINES          BK00-BCF-CMMN.          00000080
*          MACRO/DSECT          BCRCOM          BCTLDM          00000090
*          CIF-BCF-CMMN-FLD          00000100
*          COMMON FIELDS          00000110
*          DMR          00000120
*          10 BK00-BCF-REC-LNG          PIC          9(4)          COMP.          00000130
*          CIF-BCF-REC-LNG          00000140
*          RECORD LENGTH          00000150
*          DMR          00000160
*          10 BK00-BCF-BLNK-FLD          PIC          X(2).          00000170
*          CIF-BCF-BLNK-FLD          00000180
*          BLANK FIELD          00000190
*          DMR          00000200
*          10 BK00-BCF-CMMN-ID.          CIF-BCF-CMMN-ID          00000210
*          COMMON KEY          00000220
*          DMR          00000230
*          15 BK00-BCF-BR-NUM          PIC          X(2).          00000240
*          CIF-BCF-BR-NUM          00000250
*          0000 - BANK GROUP RECORD          00000260
*          0A00 - BANK RECORD          DMR          00000270
*          0BBB - BRANCH RECORD          00000280
*          A          BEING 1ST DIGIT          00000290
*          OF BRANCH NO.          00000300
*          BBB BEING BRANCH NO.          00000310
*          00000320
*          00000330
*          00000340
*          00000350
    
```

**Variable Table**

| Identifier |         |         |         |                   |
|------------|---------|---------|---------|-------------------|
| Level 1    | Level 2 | Level 3 | Level 4 | Variable Name     |
| ..         | ..      | ..      | ..      | ..                |
| ..         | ..      | ..      | ..      | ..                |
| 4          | 0       | 0       | 0       | BK00-REC          |
| 4          | 1       | 0       | 0       | BK00-BCF-CMMN     |
| 4          | 2       | 0       | 0       | BK00-BCF-CMMN-FLD |
| 4          | 2       | 1       | 0       | BK00-BCF-REC-LNG  |
| 4          | 2       | 2       | 0       | BK00-BCF-BLNK-FLD |
| 4          | 2       | 3       | 0       | BK00-BCF-CMMN-ID  |
| 4          | 2       | 3       | 1       | BK00-BCF-BR-NUM   |
| ..         | ..      | ..      | ..      | ..                |
| ..         | ..      | ..      | ..      | ..                |

**Figure 4.7** Part of Variable Table Extracted from Program Compiled Listing

In the variable table, each variable is assigned with a unique identifier with different levels. The '01', '05', '10' ... codeword of the S-COBOL program is a structural definition like *struct* in C language. Unlike ATAC (refer to 6.2.4), ATACOBOL consider each element in a structure at same level as different. The hierarchical upper level element (parent) in a structure contains the lower level elements (children). This kind of data structure is not considered in the data flow criteria proposed previously.



**Figure 4.8** Hierarchical Representation of Structure of the Example in Figure 4.7

At this point, we need to define additional rules to the selection of def-use pairs. That is, for a variable in the hierarchical structure is defined, any uses of that variable must be the same variable or a variable within the same hierarchical path (i.e. either the parent/grandparent or children/grand-child of the defined variable). To site an example, a full record (upper level variable) is read from a file at the very beginning of a program, and the breakdowns of this record (lower level variable) would be used in subsequent parts of the program. The use of these breakdowns should be considered as the use of the full record. Moreover, breakdowns of a record may be defined in various block of a program, and finally the full record is written to a file. On the other hand, each unrelated breakdowns may represent an individual attribute. Therefore, defines and uses of non-hierarchical individual variables is unlikely to require association.

A formal definition is given as follows:



Let  $i, j$  be a variables in a structure,  $\text{CompStruct}(i, j)$  is a function that returns the hierarchical relationship between  $i$  and  $j$ .

$\text{CompStruct}(i, j) = \text{TRUE}$     if  $i = j$ , or  $i$  is the parent/grandparent  $j$ , or  $j$  is the parent/grandparent  $i$

$\text{CompStruct}(i, j) = \text{FALSE}$     otherwise

Now, we re-define the two functions proposed by Weyuker *et al*:

- $\text{dcu}(x, i)$  is the set of all nodes  $j$  such that  $x \in \text{c-use}(j)$  and for which there is a def-clear path w.r.t.  $x$  from  $i$  to  $j$ , given  $\text{CompStruct}(i, j) = \text{TRUE}$ .
- $\text{dpu}(x, i)$  is the set of all edges  $(j, k)$  such that  $x \in \text{p-use}(j, k)$  and for which there is a def-clear path w.r.t.  $x$  from  $i$  to  $j$ , given  $\text{CompStruct}(i, j) = \text{TRUE}$ .

For each block, variables are scanned to check assign them to the Def/Use graph as described in Chapter 2. The classification of a variable as def/use in constructs of the S-COBOL context is defined in figures 4.6a-4.6h. The format of def/c-use/p-use information stored in file is shown in figure 4.9. Variables stored in these files are represented by the identifier associated with the variable table described beforehand.

```

/* *****
/*      SUBROUTINE - VALIDATE BCASF RECORD ROUTINE
/*      SKIP1
/*      ENTRY POINT - VALIDATE-BCASFI-REC-RTN
/*      SKIP1
/*      VALIDATE BCASF RECORD
/*      *****
/*      SKIP3
/*      PARA VALIDATE-BCASFI-REC-RTN
/*      SKIP1
/*      IF BCASFI-TRLR-READ
/*          PERFORM GEN-REPT1-MSG-RTN(ERR-MSG-13)
/*          REPT-CD-TOT = REPT-CD-TOT + 1
/*          REPT-REJT-TOT = REPT-REJT-TOT + 1
/*      SKIP1
/*      ELSE-IF BCASF-DTL-REC
/*          PERFORM EDIT-BCASFI-DTL-RTN
/*      SKIP1
/*      ELSE-IF BCASF-TRLR-REC
/*          PERFORM EDIT-BCASFI-TRLR-RTN
/*          BCASFI-TRLR-READ-INDC = TRUZ
/*      SKIP1
/*      ELSE
/*          PERFORM GEN-REPT1-MSG-RTN(ERR-MSG-12)
/*          REPT-CD-TOT = REPT-CD-TOT + 1
/*          REPT-REJT-TOT = REPT-REJT-TOT + 1
/*      EJECT
    
```

*Def*

*Data Flow Information File*  
*C-Use*

*P-Use*

| Para Num | Block Num | Variable |
|----------|-----------|----------|
| ..       | ..        |          |
| 6        | 2         | 6 2 7 0  |
| 6        | 2         | 6 2 3 0  |
| 6        | 4         | 4 8 0 0  |
| 6        | 5         | 6 2 7 0  |
| 6        | 5         | 6 2 3 0  |
| ..       | ..        |          |

| Para Num | Block Num | Variable  |
|----------|-----------|-----------|
| ..       | ..        |           |
| 6        | 2         | 6 10 13 0 |
| 6        | 2         | 6 2 7 0   |
| 6        | 2         | 6 2 3 0   |
| 6        | 2         | 6 2 12 0  |
| 6        | 5         | 6 2 7 0   |
| 6        | 5         | 6 2 3 0   |
| ..       | ..        |           |

| Para Num | Edge       |          |         | Variable |
|----------|------------|----------|---------|----------|
|          | From Block | To Block |         |          |
| ..       | ..         |          |         |          |
| 6        | 1          | 2        | 4 7 1 0 |          |
| 6        | 1          | 3        | 2 5 7 0 |          |
| 6        | 1          | 4        | 2 5 5 0 |          |
| 6        | 1          | 5        | 4 7 1 0 |          |
| 6        | 1          | 5        | 2 5 7 0 |          |
| 6        | 1          | 5        | 2 5 5 0 |          |
| ..       | ..         |          |         |          |

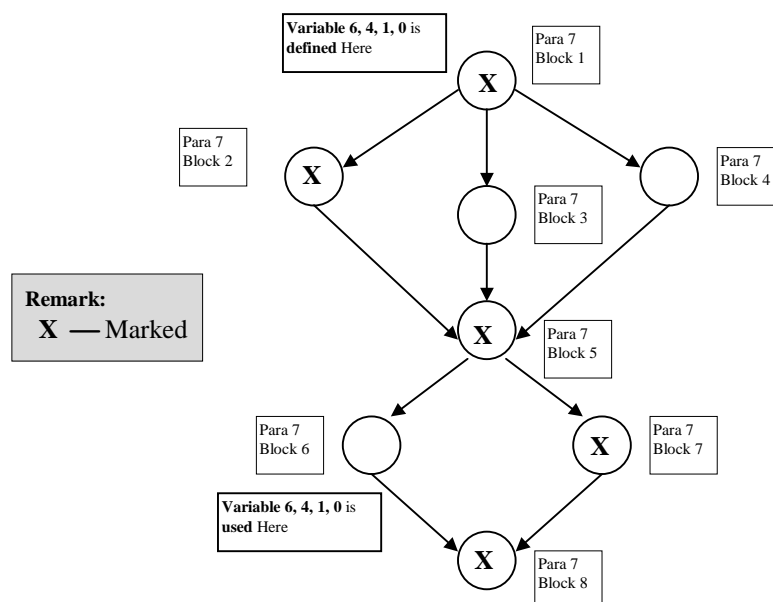
**Figure 4.9** Record Format of Def/C-Use/P-Use in Data Flow Information File

Def-use pairs are extracted from the Def/Use graph. This is achieved by exhaustive and search over the control flow graph. For every searched nodes within a searching path, the node is marked and will not be searched again. As a result, the Data Flow Information File contains the dcu and dpu sets described in Chapter 2.

## 4.6 ATACOBOL Instrumenter

From the Control Flow Information File, ATACOBOL Instrumenter gets information about blocks and their corresponding positions in the S-COBOL source file. Then ATACOBOL insert codes with correct alignment to the source. The purpose of the inserted codes is to call the ATACOBOL Runtime Module passing with an unique block identifier as the parameter. This unique block identifier is composed by the paragraph number and the node number. Screen capture of the instrumented code uploaded to OS/390 is shown on figure 4.11.

When a block is executed during testing, the Runtime Module logs down the paragraph number and block number as identifier. As a result, the execution path can be traced.



**Figure 4.10** Illustration of Def-Use Pair Searching

```

SDSF OUTPUT DISPLAY ZEDPSK1 JOB04665 DSID 121 LINE 1,769 COLUMNS 02- 81
COMMAND INPUT ==> SCROLL ==> CSR
001482 DISPLAY '4 3' PG
001484 CHK-WHOLE-BCASFI-RTN PG057300
001485 DISPLAY '5 1' PG
001490 DISPLAY '5 2' PG
001492 OPEN INPUT BCASFI PG057500
001496 INITIATE PRINT1-REPORT PG057700
001500 PERFORM READ-BCASFI-RTN PG057900
001504 DISPLAY '5 3' PG
001506 IF BCASFI-EOF PG058100
001508 DISPLAY '5 4' PG
001510 UTL167-TIME = TIME-OF-DAY PG058200
001512 UTL167-TEXT = MSG-1 PG058300
001514 DISPLAY MSG-1 UPON SYSOUT PG058400
001516 CALL 'UTL167' USING UTL167-MSG PG058500
001518 CALL 'CBCANCEL' PG058600
001522 DISPLAY '5 5' PG
001524 PERFORM EDIT-BCASFI-CNTL-RTN PG058800
001528 DISPLAY '5 6' PG
001530 REPEAT PG059000
001532 DISPLAY '5 7' PG
001534 PERFORM READ-BCASFI-RTN PG059100
001536 DISPLAY '5 8' PG
001538 UNTIL BCASFI-EOF PG059200
001540 DISPLAY '5 9' PG
001542 PERFORM VALIDATE-BCASFI-REC-RTN PG059300
001546 DISPLAY '5 10' PG
001548 TERMINATE PRINT1-REPORT PG059500
001552 DISPLAY '5 11' PG
001554 IF BCASFI-INVALID PG059700
001556 DISPLAY '5 12' PG
001558 UTL167-TIME = TIME-OF-DAY PG059800
001560 UTL167-TEXT = MSG-2 PG059900
4E 02/21

```

**Figure 4.11** Instrumented S-COBOL Source

## 4.7 ATACOBOL Runtime Routine

The function of ATACOBOL Runtime Routine has been discussed in section 4.6. Up to current implementation, the COBOL system call “DISPLAY” is employed as the Runtime Routine. It outputs the trace log to the SYSOUT (System Output) of OS/390 JES2 job held queue (Figure 4.12). Its function is similar to a output file [IBM97]. The SYSOUT is captured after testing as the input to ATACOBOL Coverage Analyser. For further development, a discrete Runtime Module can be written in IBM 370 Assembly Language and writes the output to user defined trace log files. It would then be able to support specific function in the customised runtime module.

```

SDSF OUTPUT DISPLAY ZEDPSKSA JOB09524 DSID 102 LINE 0 COLUMNS 02- 81
COMMAND INPUT ==> SCROLL ==> CSR
***** TOP OF DATA *****
1
1 2
2 1
2 2
15 1
15 2
15 3
15 4
15 5
15 6
2 3
3 1
3 2
3 3
3 4
3 5
3 6
3 7
3 8

SDSF JOB DATA SET DISPLAY - JOB ZEDPSKSA (JOB09524) DATA SET DISPLAYED
COMMAND INPUT ==> SCROLL ==> CSR
NP DDNAME STEPNAME PROCSTEP DSID OWNER C DEST REC-CNT PAGE
JESMSGLG JES2 2 ZEDPSKS X LOCAL 40
JESJCL JES2 3 ZEDPSKS X LOCAL 51
JESYSMSG JES2 4 ZEDPSKS X LOCAL 73
PRINT BFTD18 101 ZEDPSKS X LOCAL 14
S- SYSOUT BFTD18 102 ZEDPSKS X LOCAL 159
CAIPRINT BFTD18 107 ZEDPSKS X LOCAL 1, 166
CAIPRNT1 BFTD18 108 ZEDPSKS X LOCAL 43
CAIPRNT2 BFTD18 109 ZEDPSKS X LOCAL 1

```

Figure 4.12 OS/390 SYSOUT

#### 4.8 ATACOBOL Coverage Analyser

The ATACOBOL Coverage Analyser takes the Control Flow Information File, Data Flow Information File and Trace Log as inputs. It performs several levels of coverage measurements:

- **Block Coverage:**

From the Control Flow Information File, every node identifier is compared with the Trace Log. (See figure 4.13)

- **Decision Coverage:**

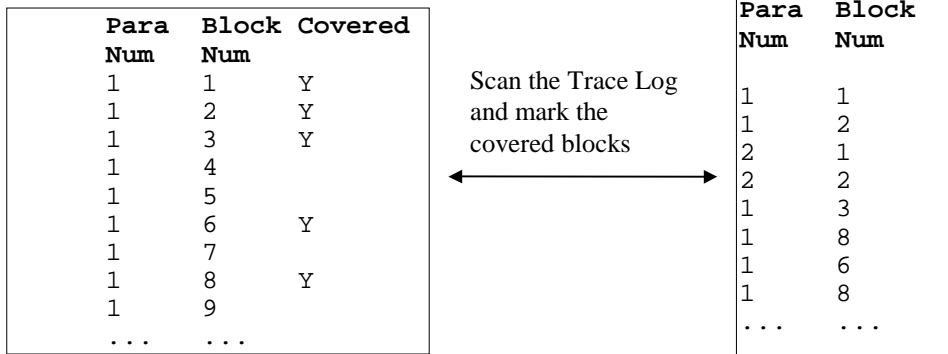
Form the Control Flow Information File, every node pairs with a path between them is extracted to compare with the Trace Log. (see figure 4.14)

- **C-Uses and P-Uses Data Flow Coverage:**

From the Data Flow Information File's Def-Use graph, C-Uses and P-Uses is checked against the Trace Log. (see figure 4.15)

**Decision Coverage Criteria extracted  
Control Flow Information File  
Log**

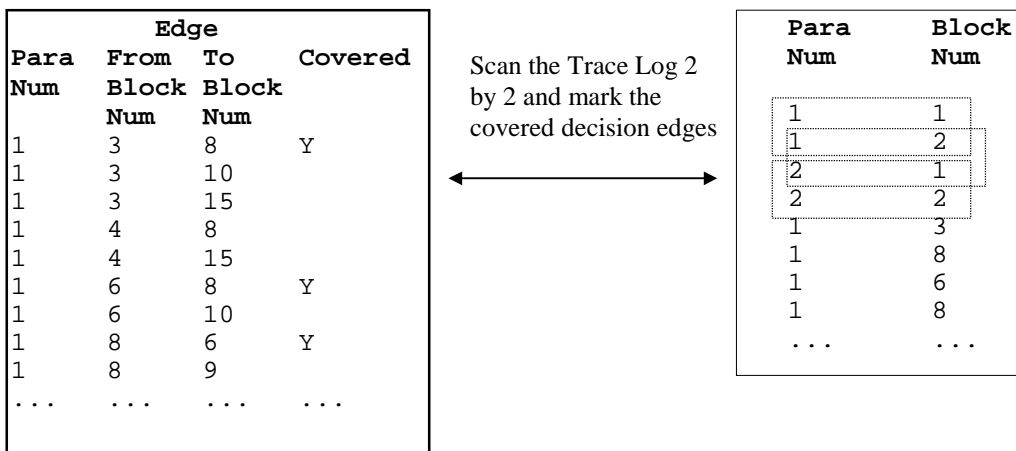
**Trace**



**Figure 4.13** Analysing Block Coverage

**Block Coverage Criteria extracted  
Control Flow Information File  
Log**

**Trace**



**Figure 4.14** Analysing Decision Coverage

***C-Use Def-Use Pair extracted from Control Flow Information File and Data Flow Information File***

| Para Num | Variable | Def Block Num | Use Block Num | Covered |
|----------|----------|---------------|---------------|---------|
| 1        | 2 2 3    | 1             | 2             | Y       |
| 1        | 2 2 7    | 2             | 12            | S       |
| 1        | 2 2 7    | 4             | 13            | S       |
| 1        | 4 0 0    | 3             | 5             |         |
| 1        | 4 1 0    | 4             | 6             |         |
| 1        | 4 1 0    | 6             | 7             |         |
| 1        | 5 0 0    | 6             | 5             |         |
| 1        | 9 1 1    | 6             | 6             |         |
| 1        | 9 1 3    | 9             | 13            |         |
| ...      | ...      | ...           | ...           |         |

- Whenever a block is found executed in the Trace Log, the corresponding def-use pair defined in that block will be marked as “S” (Tracing Started).
- Upon the use of the variable, the def-use pair will be marked as covered.
- However, if another same variable is defined before the use of the started tracing variable, the tracing will be cancelled. And, that other variable will be marked as “S” instead.

***Trace Log***

| Para Num | Block Num |
|----------|-----------|
| 1        | 1         |
| 1        | 2         |
| 2        | 1         |
| 2        | 2         |
| 1        | 3         |
| 1        | 8         |
| 1        | 6         |
| 1        | 8         |
| ...      | ...       |

***P-Use Def-Use Pair extracted from Control Flow Information File and Data Flow Information File***

| Para Num | Variable | Def Block Num | Use Edge From Block Num | Use Edge To Block Num | Covered |
|----------|----------|---------------|-------------------------|-----------------------|---------|
| 1        | 3 1 0    | 2             | 6                       | 8                     | S       |
| 1        | 3 1 0    | 2             | 6                       | 10                    | S       |
| 1        | 3 3 0    | 2             | 3                       | 8                     | S       |
| 1        | 3 3 0    | 2             | 3                       | 10                    | S       |
| 1        | 3 3 0    | 2             | 3                       | 15                    | S       |
| 1        | 6 2 0    | 7             | 4                       | 8                     |         |
| 1        | 6 2 0    | 7             | 4                       | 15                    |         |
| 1        | 6 4 1    | 2             | 11                      | 15                    | S       |
| 1        | 6 4 1    | 2             | 11                      | 16                    | S       |
| ...      | ...      | ...           | ...                     | ...                   |         |

**Figure 4.15** Analysing Data Flow Coverage

ATACOBOL Analyser finally outputs reports about the coverage measurement, including a summary report of the percentage of coverage per paragraph and uncovered blocks (Figure 4.17), decision edges (Figure 4.18), c-use (Figure 4.19) or p-use (Figure 4.20) for individual coverage measures. Figure 4.16 shows an typical summary report layout:

| Total Para Number=16 |       |        |          |       |       |       |       |       |
|----------------------|-------|--------|----------|-------|-------|-------|-------|-------|
| Para #               | Block | Cover  | Decision | Cover | C-Use | Cover | P-Use | Cover |
| 1                    | 1/1   | (100%) | 0/0      | (-%)  | 0/0   | (-%)  | 0/0   | (-%)  |
| 2                    | 1/1   | (100%) | 0/0      | (-%)  | 0/0   | (-%)  | 0/0   | (-%)  |
| 3                    | 0/4   | (0%)   | 0/1      | (0%)  | 0/13  | (0%)  | 0/0   | (-%)  |
| 4                    | 1/1   | (100%) | 0/0      | (-%)  | 0/0   | (-%)  | 0/0   | (-%)  |
| 5                    | 5/13  | (38%)  | 2/5      | (40%) | 0/0   | (-%)  | 0/0   | (-%)  |
| 6                    | 0/5   | (0%)   | 2/2      | (0%)  | 4/6   | (0%)  | 3/5   | (0%)  |
| 7                    | 0/5   | (0%)   | 0/1      | (0%)  | 0/7   | (0%)  | 0/4   | (0%)  |
| 8                    | 0/6   | (0%)   | 0/4      | (0%)  | 0/1   | (0%)  | 0/0   | (-%)  |
| 9                    | 0/4   | (0%)   | 0/1      | (0%)  | 0/9   | (0%)  | 0/4   | (0%)  |
| 10                   | 0/4   | (0%)   | 0/1      | (0%)  | 0/11  | (0%)  | 0/0   | (-%)  |
| 11                   | 0/1   | (0%)   | 0/0      | (-%)  | 0/6   | (0%)  | 0/0   | (-%)  |
| 12                   | 0/21  | (0%)   | 0/11     | (0%)  | 0/3   | (0%)  | 0/0   | (-%)  |
| 13                   | 0/19  | (0%)   | 0/8      | (0%)  | 0/15  | (0%)  | 0/10  | (0%)  |
| 14                   | 0/14  | (0%)   | 0/6      | (0%)  | 0/18  | (0%)  | 0/8   | (0%)  |
| 15                   | 0/8   | (0%)   | 0/4      | (0%)  | 0/10  | (0%)  | 0/5   | (0%)  |
| 16                   | 0/1   | (0%)   | 0/0      | (-%)  | 0/23  | (0%)  | 0/10  | (0%)  |
| Total                | 8/108 | (7%)   | 4/44     | (9%)  | 4/122 | (3%)  | 3/46  | (7%)  |

**Figure 4.16** A typical coverage measurement summary report

```

Report of Uncovered Blocks
=====
Uncovered Block Number

Paragraph #1
-----

Paragraph #2
-----

Paragraph #3
-----

2
3
4
5
...

```

**Figure 4.17** A typical part of report on uncovered block



```

Report of Uncovered Decision Edges
=====

Uncovered Decision Edge (From Block Number  --  To Block Number)

Paragraph #1
-----

Paragraph #2
-----

Paragraph #3
-----
3      --      4

Paragraph #4
-----

Paragraph #5
-----
3      --      4
8      --      9
8      --     10

Paragraph #6
-----
5      --      6
5      --      7
...

```

**Figure 4.18** A typical part of report on uncovered decision edge

```

Report of Uncovered Def-C-Uses
=====

Paragraph #1
-----

Paragraph #2
-----

Paragraph #3
-----
Var:   6      16      0      DEF:   0      C-USE:  2
Var:   6      21      9      DEF:   0      C-USE:  2
Var:   6      21     10      DEF:   0      C-USE:  2
Var:   6      21     11      DEF:   0      C-USE:  2
Var:   6      14      2      DEF:   0      C-USE:  2
Var:   6      11      1      DEF:   0      C-USE:  2
Var:   6      1      1      DEF:   0      C-USE:  2
Var:   6      1      1      DEF:   0      C-USE:  2
Var:   6      1      3      DEF:   0      C-USE:  2
Var:   6      17      1      DEF:   0      C-USE:  2
Var:   6      17      2      DEF:   0      C-USE:  2
Var:   6      17      4      DEF:   0      C-USE:  2
Var:   6      17      3      DEF:   0      C-USE:  2

Paragraph #4
-----
...

```

**Figure 4.19** A typical part of report on uncovered C-Use

```

Report of Uncovered Def-P-Uses
=====

Paragraph #1
-----

Paragraph #2
-----

Paragraph #3
-----

Paragraph #4
-----

Paragraph #5
-----

Paragraph #6
-----
Var:   6       5       2       DEF:   0       P-USE:  3       --       4
Var:   6       5       2       DEF:   0       P-USE:  8       --       9
Var:   6       5       2       DEF:   0       P-USE:  8       --      10
Var:   6       5       3       DEF:   0       P-USE: 11       --      12
Var:   6       5       1       DEF:   0       P-USE: 13       --      14

Paragraph #7
-----
Var:   6       5       2       DEF:   2       P-USE:  5       --       6
Var:   6       5       2       DEF:   2       P-USE:  5       --       7
Var:   1       1       0       DEF:   0       P-USE:  5       --       6
Var:   1       1       0       DEF:   0       P-USE:  5       --       7

Paragraph #8
-----
...

```

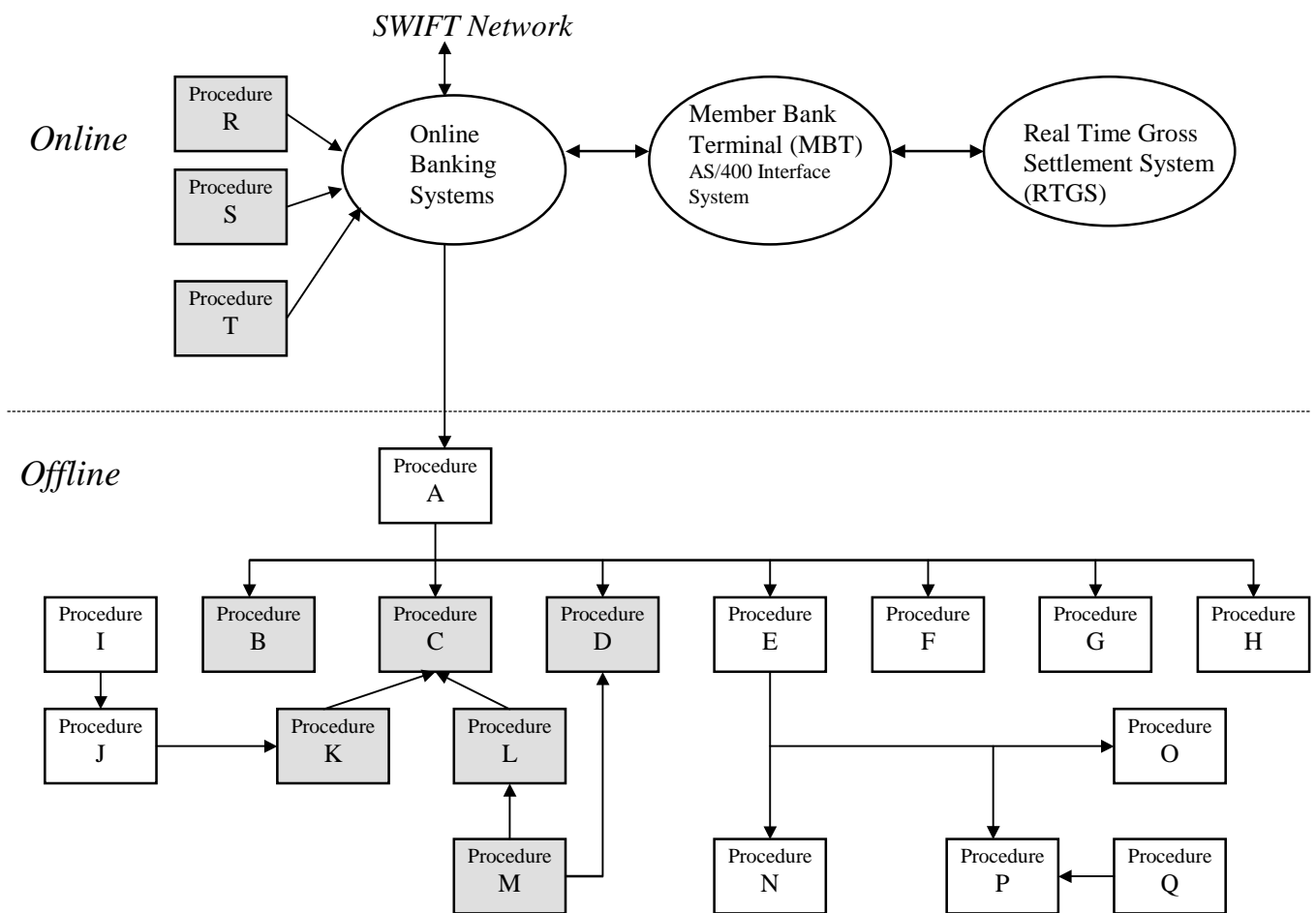
**Figure 4.20** A typical part of report on uncovered P-Use

## 5. MEASUREMENT

### 5.1 System Description

In this chapter, ATACOBOL is applied with live production programs and test cases.

The system under measurement is the batch programs of interface system of HSBC mainframe application BFT to RTGS.



**Figure 5.1** System Overview

The Real Time Gross Settlement System (RTGS) was introduced in December 1996.

Basically, it is a local central clearing system in Hong Kong SAR. Similar systems are

operating over the world. For SWIFT payments that require local settlement, the RTGS would be employed. It is one of the most advanced in the Asia Pacific Region [Bee98]. The system is in full compliance with international standards and has greatly enhanced the robustness of the inter-bank payment system by reducing settlement risk. Under this system, all licensed banks in Hong Kong open and maintain settlement accounts with HKMA (Hong Kong Monetary Authority), and have access to the system. Inter-bank payments settled across the books of HKMA are final and irrevocable.

Therefore, each licensed bank would have their computer system connected to RTGS. The host application BFT (Bank Fund Transfer) Systems that interfaces HSBC and RTGS is employed in this measurement. Figure 5.1 shows the overview this system. The Online Banking Systems are developed in IBM 370 Assembly Language. A procedure consists of 1 to 5 modules, mostly are written in APS COBOL language for offline execution. There are 3 online procedure that creates real-time spools to the Online Banking Systems.

## **5.2 Number of C-Use and P-Use Against Number of Faults**

Shaded procedures in Figure 5.1 are selected for this measurement. There are totally 21 modules (say module M1 to M21) developed in APS COBOL in the selected procedures. The module history and source can be retrieved from the version control system of the development environment (Table 5.1). Problem/Changes Reports during March 1998 to February 1999 are also collected (Table 5.2). The information is plotted in Graph 5.1 and Graph 5.2.

From the graphs, we find two peaks in the program size changes. This two peaks, July and January, reflects two major release at that time. Refer to Graph 5.2, during the

first release, major the complexity of the modules: number of block, edges, C-Uses and P-Uses increased as the number of faults reported also increases. The number of faults reduced as program fixes released. In September, as the number of transactions handled by the modules released in July increased. New problem broke out. It accounts for the higher fault rate in September. Overall, the graph shows that except the number of C-Uses and P-Uses, other factors affects the reliability of a software system.

However, the number of blocks, number of C-Uses and No. of P-Uses increased is highly correlated as reflected in Graph 5.1. That makes the determination of the impact of the data flow to software reliability difficult. It demands extensive measurement to collect more statistics until we can get a clear picture of the effects of the amount of C-Uses and P-Uses to the software reliability.

**Table 5.1 Module Amendment Statistics**

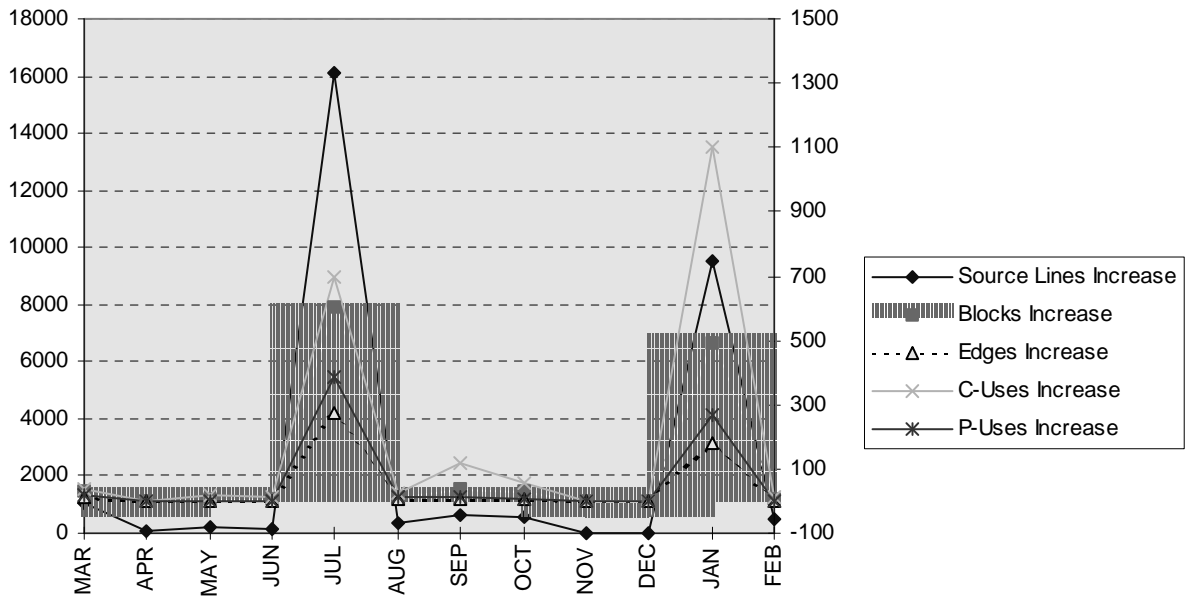
| Date            | Affected Modules       | No. of Source Lines Increased | No. of Blocks Increased | No. of Edges Increased | No. of C-Uses Increased | No. of P-Uses Increased |
|-----------------|------------------------|-------------------------------|-------------------------|------------------------|-------------------------|-------------------------|
| Original Number | M1-M21                 | 172,641                       | 29,586                  | 18,622                 | 51,877                  | 22,357                  |
| 1998 March      | M1-M4                  | 1,032                         | 28                      | 12                     | 36                      | 20                      |
| 1998 April      | M5-M7                  | 42                            | 0                       | 0                      | 2                       | 0                       |
| 1998 May        | M2, M13                | 220                           | 6                       | 2                      | 21                      | 2                       |
| 1998 June       | M11                    | 165                           | 4                       | 0                      | 9                       | 0                       |
| 1998 July       | M2-M6, M8-M14, M18-M21 | 16,088                        | 603                     | 275                    | 698                     | 384                     |
| 1998 August     | M2                     | 323                           | 11                      | 4                      | 27                      | 13                      |
| 1998 September  | M3, M8-M9, M18-M21,    | 647                           | 39                      | 5                      | 120                     | 10                      |

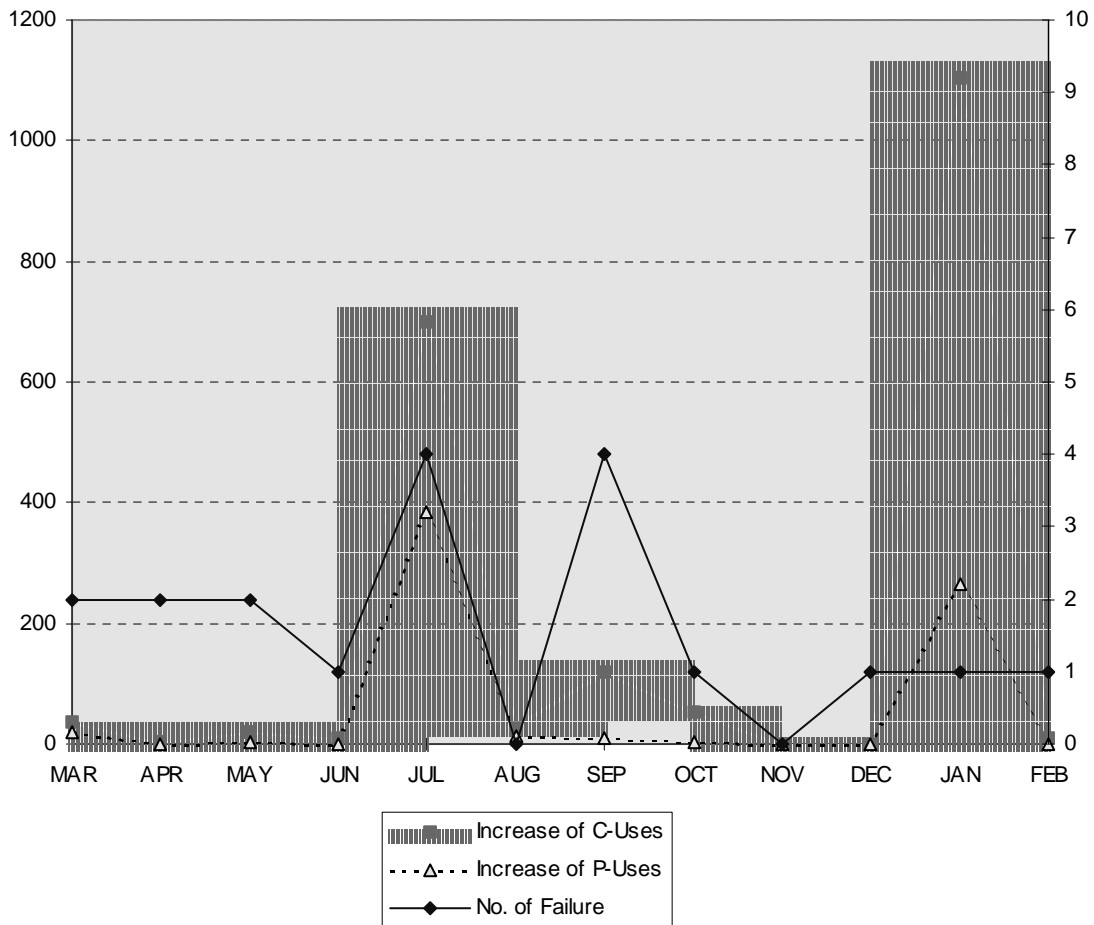
|                  |                             |       |     |     |       |     |
|------------------|-----------------------------|-------|-----|-----|-------|-----|
| 1998<br>October  | M6-M7,<br>M14,M15           | 544   | 28  | 4   | 54    | 4   |
| 1998<br>November | N/A                         | 0     | 0   | 0   | 0     | 0   |
| 1998<br>December | N/A                         | 0     | 0   | 0   | 0     | 0   |
| 1999<br>January  | M2-M6,<br>M7-M9,<br>M13-M17 | 9,556 | 490 | 182 | 1,103 | 266 |
| 1999<br>February | M1,<br>M6-M7                | 461   | 8   | 0   | 9     | 0   |

**Table 5.2 Module Failure Statistics**

| Month          | No. of Problem Reported |
|----------------|-------------------------|
| 1998 March     | 2                       |
| 1998 April     | 2                       |
| 1998 May       | 2                       |
| 1998 June      | 1                       |
| 1998 July      | 4                       |
| 1998 August    | 0                       |
| 1998 September | 4                       |
| 1998 October   | 1                       |
| 1998 November  | 0                       |
| 1998 December  | 1                       |
| 1999 January   | 1                       |
| 1999 February  | 1                       |

**Graph 5.1 Growth of Modules**



**Graph 5.2 No. of Failure with C&P-Uses Changes**

### 5.3 Data Flow Coverage of Live Test Cases

Three modules (say O1, O2 and O3) are selected for coverage measurement with the system test and user acceptance test cases before their last release.

| Module                               | O1    | O2    | O3     |
|--------------------------------------|-------|-------|--------|
| Number of Source Lines               | 4,276 | 7,225 | 15,044 |
| Number of Test cases                 | 51    | 12    | 5      |
| Number of Blocks                     | 831   | 1,473 | 2,206  |
| Percentage of Block Coverage         | 100%  | 42%   | 18%    |
| Number of Decision Edges             | 504   | 999   | 1,430  |
| Percentage of Decision Edge Coverage | 98%   | 33%   | 7%     |
| Number of C-Uses                     | 2,012 | 2,604 | 3,695  |
| Percentage of C-Uses Coverage        | 94%   | 22%   | 5%     |
| Number of P-Uses                     | 838   | 1,317 | 2,121  |
| Percentage of P-Uses Coverage        | 95%   | 28%   | 6%     |



The above measurement demonstrates ATACOBOL's ability to measure production scale modules. O1 is a newly created module. In the system test and user acceptance test, the functionality is tested thoroughly. On the other hand, O2 and O3 are enhanced versions, only the enhanced features is tested and few basic re-tested with representative regression test case. The coverage for O2 and O3, is therefore, relatively low. It would be useful if the measurement tool can focus only on the affected parts of a program enhancement. Please refer to 7.10. for further discussions on this issue.

For further measurement, we could measure the increase in number of test cases against the percentage of coverage. The behaviour of growth of coverage relates to the organisation of the program. If the program has evenly distributed coding on various functions, the growth curve would be evenly increase. If the program has a large piece of common mainline, the growth curve is expected to be concave.

## 6. EVALUATION

### 6.1 Experience in Using ATAC

#### 6.1.1 The applications of ATAC

After using ATAC, we can identify the following applications of ATAC in the software testing process:

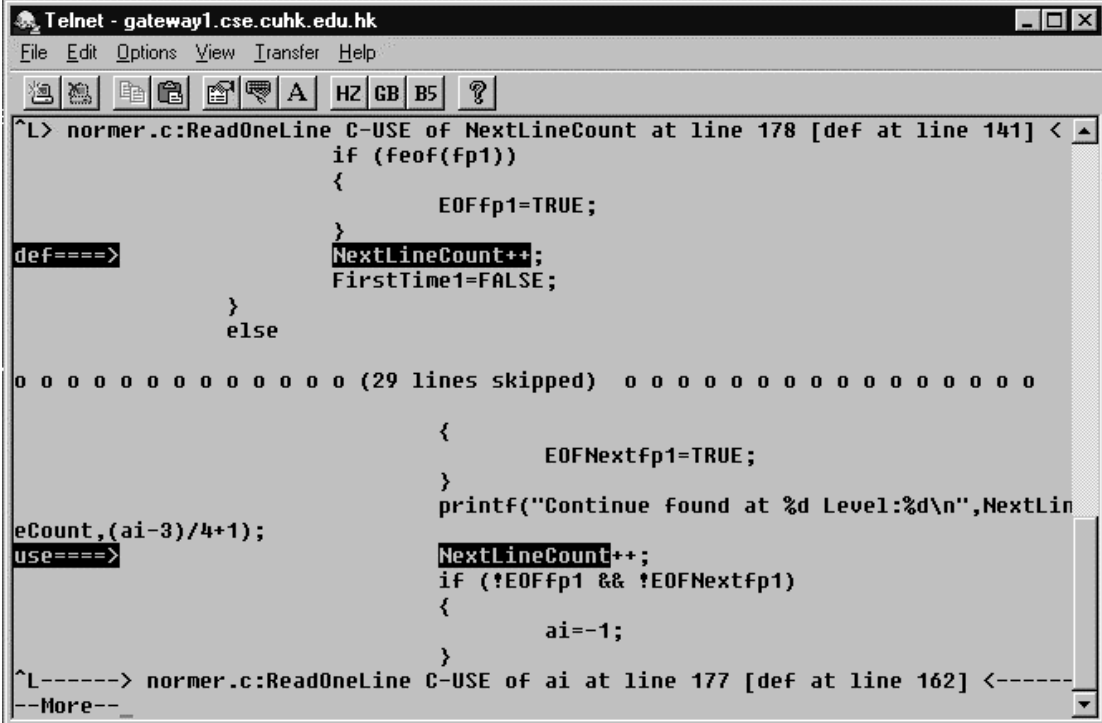
- measuring test set completeness by control and data flow coverage;
- displaying non-covered code to aid in test cases creation;
- reducing regression test set size by determining minimal test set out of tested cases.

In section 3.1, we have described these applications briefly. We would further examine these purposes in more detail.

The first purpose, measuring test completeness, gives an objective measure of how completely a program or routine has been tested. This measure is useful in evaluating the quality of the testing procedure being used, and in establishing a level of confidence in the quality of tested programs. A low coverage score indicates that the tests do not effectively exercise the program. A high coverage score establishes confidence that the program, in passing the tests, works correctly.

The second purpose, displaying code not covered, is a programmer's aid for unit testing. Since a thoroughly-done unit testing job can vastly reduce the overall cost of testing a software system, a programmer can use the coverage displays to reveal particular code constructs that have not been covered by unit testing. By examining the code, the programmer can discover tests that will cause these, as yet not covered, constructs to be covered. After running these additional tests, the programmer can

check which constructs are newly covered, and examine the remaining non-covered constructs.



```
Telnet - gateway1.cse.cuhk.edu.hk
File Edit Options View Transfer Help
[Icons] HZ GB B5 ?
^L> normer.c:ReadOneLine C-USE of NextLineCount at line 178 [def at line 141] <
    if (feof(fp1))
    {
        EOFfp1=TRUE;
    }
def====>    NextLineCount++;
            FirstTime1=FALSE;
    }
    else
o o o o o o o o o o o o o o o (29 lines skipped) o o o o o o o o o o o o o o o
    {
        EOFNextfp1=TRUE;
    }
    printf("Continue found at %d Level:%d\\n",NextLin
eCount,(ai-3)/4+1);
use====>    NextLineCount++;
            if (!EOFfp1 && !EOFNextfp1)
            {
                ai=-1;
            }
^L-----> normer.c:ReadOneLine C-USE of ai at line 177 [def at line 162] <-----
--More--
```

**Figure 6.1** Uncovered construct shown by ATAC (Measuring ATACOBOL)

The tests run over the life of a program are often collected together to form a regression test set. The regression test set is re-run each time the program is modified to verify that the modifications have not adversely affected the behaviour of the program. At some point a regression test set may grow large enough that it is not practical to run the whole set of tests after small program modifications. Hence the third purpose of coverage testing uses the coverage measure to select a subset of the regression tests which together achieve a high level of coverage. This technique may identify tests that add no coverage at all to the regression tests, and are therefore candidates for deletion.

### 6.1.2 Limitations of ATAC

In using ATAC release 3.3.13, we have encountered that ATAC does not apply the all-C-use path selection criteria according to [Rap84] for a simple case. We make use of three simple programs to illustrate the findings by ATAC sessions. The first program's (test1.c) c-uses paths are selected correctly by ATAC while p-uses paths are not selected. The second one (test2.c) is modified from test1.c to enable p-uses to be detectable by ATAC. Again, the third program test3.c is modified from test2.c, ATAC. In this program, ATAC fails to identify the c-uses paths

```
solar1.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atac -v
ATAC release 3.3.13 Sep 26, 1994.
Copyright (c) 1993 Bell Communications Research, Inc. (Bellcore)
Send comments or questions to atac@bellcore.com.
solar1.cs.cuhk.hk:/uac/ptmsc/kssze/sample>
solar1.cs.cuhk.hk:/uac/ptmsc/kssze/sample> cat test1.c
#include <stdio.h>

main()
{
    int x;
    int y;
    int z;

    printf("Enter x:\n");
    scanf("%d",&x);
    printf("Enter y:\n");
    scanf("%d",&y);

    if (x>0)
    {
        z=1;
    }
    else
    {
        z=2;
    }
    if (y>0)
    {
        z=z+1;
    }
    else
    {
        z=z+2;
    }
}
```

This small program takes 2 inputs x, y and depending on the sign of x and y, subsequent path is executed. This program has the flow graph as shown in figure 6.2.

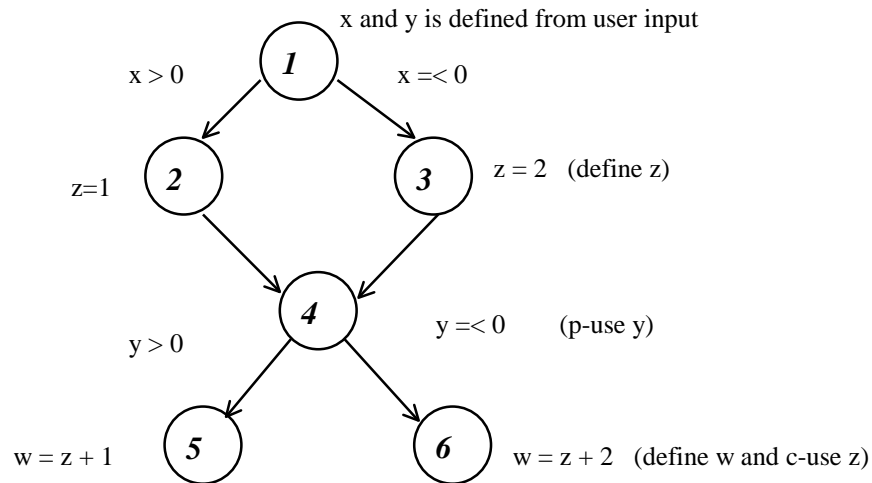


Figure 6.1

According to the path selection criteria by [Rap84] and as discussed in Chapter 2, for all uses, there should be totally 4 c-def/use pairs: (2, 5), (2, 6), (3, 5), (3, 6) and 4 p-def/use pairs (1, (1,2)), (1, (1,3)), (1, (4,5)), (1, (4,6)).

Now, this small program is compiled by ATAC. Then, the program is executed for path (1, 2, 4, 5) only. Note that for ATAC to display the coverage summary, the program under test should be executed at least once.

```

solar1.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atacCC -g -c test1.c
solar1.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atacCC -g -o test1 test1.o
solar1.cs.cuhk.hk:/uac/ptmsc/kssze/sample> test1
Enter x:
1
Enter y:
1
sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atac -s -f test1.trace test1.atac
% blocks      % decisions  % C-Uses    % P-Uses    function
-----
82(9/11)      50(2/4)      25(1/4)     100(0)      main
sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample>

```

However, ATAC reported that there are 4 c-uses and no p-uses is defined. The amount for c-uses is correct while p-use is not found by ATAC. We revised the program as follows:

```

solar1.cs.cuhk.hk:/uac/ptmsc/kssze/sample> cat test2.c
#include <stdio.h>

main()
{
    int w;
    int x;
    int y;
    int z;

```

```

x=1;
y=1;

if (x>0)
{
    z=1;
}
else
{
    z=2;
}
if (y>0)
{
    w=z+1;
}
else
{
    w=z+2;
}
}

```

In this case, variable x and y is not defined from users input through parameters &x and &y of function scanf. Instead, x and y is defined directly by assignment statement.

After a path is executed, ATAC reports the following result:

```

sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atac -s -f test2.trace test2.atac
% blocks      % decisions  % C-Uses    % P-Uses    function
-----
71(5/7)       50(2/4)     25(1/4)     100(4)      main

```

ATAC reports the amount of c-uses and p-uses as expected. Comparing test1.c and test2.c, it is clear that ATAC is unable to recognise the input to parameter &x and &y as the definition of variable x and y. It may be because all parameters passed in a function are all considered as outputs instead of inputs by ATAC in the C language context.

Another program test3.c is analysed by ATAC as follows:

```

solar1.cs.cuhk.hk:/uac/ptmsc/kssze/sample> cat test3.c
#include <stdio.h>

main()
{
    int x;
    int y;
    int z;

    x=1;
    y=1;

    if (x>0)
    {
        z=1;
    }
    else
    {
        z=2;
    }
    if (y>0)

```

```

    {
      z=z+1;
    }
    else
    {
      z=z+2;
    }
  }
}

```

This program is same as test2.c except node 5 and 6 are revised. Variable is both defined and used in node 5 and 6. After the program is executed for once, we obtain the following result:

```

sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atac -s -f test4.trace test4.atac
% blocks      % decisions  % C-Uses    % P-Uses    function
-----
71(5/7)      50(2/4)     100(0)      100(4)      main

```

ATAC becomes unable to detect any c-uses. Although variable *z* is both defined and used in node 5 and 6. However, its uses related to the definition in node 2 and 3 should not be ignored. Here we have identified two limitations of ATAC release 3.3.13.

## 6.2 Expanding the Data Flow Coverage Definition to C

The literature defines the data flow coverage criteria based on an idealised programming language. In expanding the definitions to C, new data types and structures should be catered. However, none of the enhanced definition is presented in the publications about ATAC [Hor90], [Hor94], [Lyu96].

In the following sub-sections, we will investigate how ATAC treats *global variable*, *array*, *pointer*, *struct*, and *union* of C language one by one.

### 6.2.1 Handling Global Variables

ATAC is routine based, that means, ATAC only analysis data flow coverage within functions locally but not globally. Therefore, even a global variable is defined and used in different functions, ATAC cannot recognise it (refer to the following ATAC

session). This can also be considered as another limitation of ATAC. In [Rap85], the data flow coverage is only defined for single procedure program only. To apply the definitions to C language, ATAC seems only choose to analyse intra-function data flow coverage in order to reduce complexity.

```
#include <stdio.h>

void show(void);

int x;
int y;

main()
{
    printf("Enter x:\n");
    scanf("%d",&x);
    if (x > 0)
    {
        y=1;
    }
    else
    {
        y=2;
    }
    show();
}

void show()
{
    printf("y is %d",y);
}
```

Variable `y` is defined at two nodes of `main ()` and c-used in `show()`. Globally, there should be 2 c-uses. However, ATAC only analysis coverage within functions:

```
sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atac -s -f testglob.trace testglob.atac
% blocks      % decisions  % C-Uses    % P-Uses    function
-----
86(6/7)       50(1/2)     100(0)      50(1/2)     main
100(2)        100(0)     100(1)      100(0)     show
89(8/9)       50(1/2)     100(1)      50(1/2)     == total ==
```

## 6.2.2 Handling Array Elements

An array contains different elements, we will consider whether these element as the same or different by the following ATAC experiments. Let us perform a control experiment first, this control will also serve for the next sub-sessions.

```
sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> cat control.c
#include <stdio.h>

main()
{
    int x;
    int y;
    int z;

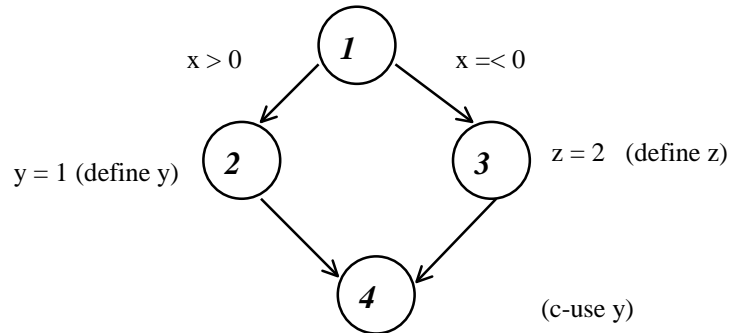
    printf("Enter x:\n");
    scanf("%d",&x);
    if (x > 0)
```



```

{
  y=1;
}
else
{
  z=2;
}
printf("y is %d",y);
}

```



**Figure 6.3**

This program defines  $y$  and  $z$  in two nodes separately and only  $y$  is c-used in the final node. Therefore, only one c-use is defined. On the other hand, if  $z$  is defined also in node 2, there are 2 c-uses. It will be used as the measure to see whether ATAC treats variables on the two side the same or different.

```

sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atac -s -f control.trace control.atac
% blocks      % decisions  % C-Uses    % P-Uses    function
-----
86(6/7)      50(1/2)     100(1)     100(0)     main

```

We perform the experiment to investigate how ATAC treats different elements of an array now:

```

sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> cat testarray.c
#include <stdio.h>

main()
{
  int x;
  int s[3];

  printf("Enter x:\n");
  scanf("%d",&x);
  if (x > 0)
  {
    s[0]=1;
  }
  else
  {
    s[1]=3;
  }
  printf("s is %d",s[2]);
}

```

Refer to figure 6.3, we put different array elements on the two side (node 2 and node 3). The result reveals that ATAC considers different array elements as the same as it recognise 2 c-uses.

```
sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atac -s -f testarray.trace testarray.trace
% blocks      % decisions  % C-Uses    % P-Uses    function
-----
86(6/7)       50(1/2)     50(1/2)     100(0)      main
```

### 6.2.3 Handling Pointers

We continue to see how ATAC treats pointers by similar experiment. In experimenting arrays, we identified ATAC treats pointers def/use in 2 cases. The first case is as follows:

```
sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> cat testptr1.c
#include <stdio.h>

main()
{
    int x;
    int *z;

    printf("Enter x:\n");
    scanf("%d",&x);
    if (x > 0)
    {
        *z=1;
    }
    else
    {
        *z=3;
    }
    printf("z is %d",z);
}
```

In this case, the field pointed by the pointer is defined in both nodes (node 2 and node 3 in figure 6.3) and the pointer is used for display. ATAC recognise the pointer and the field pointed as different variables. So, no c-use is identified by ATAC.

```
sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atac -s -f testptr1.trace testptr1.atac
% blocks      % decisions  % C-Uses    % P-Uses    function
-----
71(5/7)       50(1/2)     100(0)     100(0)      main
```

The second case is as follows:

```
sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> cat testptr2.c
#include <stdio.h>

main()
{
    int x;
    int *z;

    printf("Enter x:\n");
    scanf("%d",&x);
```

```

if (x > 0)
{
    z++;
}
else
{
    z+=2;
}
printf("z is %d",*z);

```

In this case, the pointer is defined in both nodes (node 2 and node 3 in figure 6.2) and the field pointed is used. ATAC recognise the pointer and the field pointed as the same. So, 2 c-use is identified by ATAC.

```

sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atac -s -f testptr1.trace testptr1.atac
% blocks      % decisions  % C-Uses    % P-Uses    function
-----
86(6/7)       50(1/2)      50(1/2)     100(0)      main

```

### 6.2.4 Handling Structure

The following ATAC session investigates how ATAC treats different elements within a structure. Variable name and weight are defined separately in node 2 and 3 of figure 6.2.

```

sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> cat teststruct.c
#include <stdio.h>

main()
{
    struct
    {
        char name[2];
        int weight;
    } y;

    int x;

    printf("Enter x:\n");
    scanf("%d",&x);
    if (x > 0)
    {
        y.name[0]='A';
    }
    else
    {
        y.weight=175;
    }
    printf("weight is %d",y.weight);
}

```

The result shows that 2 c-uses is recognised by ATAC. Therefore, it reveals that ATAC recognise different element within a structure as the same variable. This may not be fair in case the elements are indeed defined and used for individual purposes.

```

sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atac -s -f teststruct.trace
teststruct.atac
% blocks      % decisions  % C-Uses    % P-Uses    function

```

```
-----
86(6/7)      50(1/2)      50(1/2)      100(0)      main
```

### 6.2.4 Handling Union

The investigation for union is just about the same for structure:

```
sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> cat testunion.c
#include <stdio.h>

main()
{
    union
    {
        char name[2];
        int  weight;
    } y;

    int x;

    printf("Enter x:\n");
    scanf("%d",&x);
    if (x > 0)
    {
        y.name[0]='A';
    }
    else
    {
        y.weight=175;
    }
    printf("weight is %d",y.weight);
}
```

And the result is also the same:

```
sparc53.cs.cuhk.hk:/uac/ptmsc/kssze/sample> atac -s -f testunion.trace testunion.atac
% blocks      % decisions  % C-Uses    % P-Uses    function
-----
86(6/7)      50(1/2)      50(1/2)      100(0)      main
```

## 6.3 Experience in Using ATACOBOL

After using ATACOBOL to carry out measurement, though the presentation is still primitive, ATACOBOL shows its capability in:

- measuring test set completeness by control and data flow coverage;
- displaying non-covered code to aid in test cases creation;

### 6.3.1 Using ATAC to Measure ATACOBOL

In the development of ATACOBOL, ATAC is applied to measure the coverage of ATACOBOL. On one side, we can gain more experience in using ATAC, and on the

other side, to guarantee the test completeness of ATACOBOL. The following table shows the final coverage percentage. For detail descriptions of the modules, please refer to Appendix A.

| Module Name | Block Coverage | Decision Coverage | C-Use Coverage | P-Use Coverage |
|-------------|----------------|-------------------|----------------|----------------|
| NORMER      | 97%            | 95%               | 79%            | 63%            |
| ROUTER      | N/A            | N/A               | N/A            | N/A            |
| INSTRUER    | 93%            | 89%               | 72%            | 77%            |
| BVAR        | 100%           | 91%               | 86%            | 85%            |
| BDEFUSE     | N/A            | N/A               | N/A            | N/A            |
| PSEARCH     | N/A            | N/A               | N/A            | N/A            |
| ANALYSER    | N/A            | N/A               | N/A            | N/A            |

The testing of ATACOBOL achieved high coverage rate in general. Uncovered blocks are coding that handle exception or extreme conditions, these usually consume extra effort to make the test cases to fulfil the coverage requirement. Some modules are unable to be measured by ATAC due to nested data structure syntax supported by Visual C++ 6.0 while not supported by the SunOS 5.6 SunOS/BSD Compatibility Package C Compiler.

### 6.3.2 The Usefulness of Enhanced Rules on Data Structures

In the current ATACOBOL implementation, it supports 3 hierarchical levels of data structure representation as described in section 4.5. The module O2 described in section 5.3 is used again to compare the difference if elements of a data structure are not distinguished from each other. This experiment is achieved by modifying the variable table to wipe away level 2 and level 3 identifier of a variable.

Figure 6.4 shows a section of live data definition in APS COBOL. COBOL programmers used to collect variables under the same data structure label for documentation reason rather than any intrinsic relationship among the variables. Each

variable is an individual counter, they are collected under the data (variable) label 'COUNTERS'.

```

05  COUNTERS.
    SKIP1
    10  LINE-CNT          PIC 9(2)    VALUE 80.
        88  PAGE-OVERFLOW      VALUE 76 THRU 80.
    10  LINE-CNT1        PIC 9(2)    VALUE 80.
        88  PAGE-OVERFLOW1     VALUE 76 THRU 80.
    10  LINE-CNT2        PIC 9(2)    VALUE 80.
        88  PAGE-OVERFLOW2     VALUE 76 THRU 80.
    10  PAGE-CNT         PIC 9(2)    VALUE 0.
    10  PAGE-CNT1        PIC 9(2)    VALUE 0.
    10  PAGE-CNT2        PIC 9(2)    VALUE 0.

```

**Figure 6.4** Live APS COBOL structural data definition

The experiment result shown in the table reveals that nearly doubled amount of C-Uses and P-Uses are identified by ATACOBOL if the enhanced rules are not applied. That means nearly the same amount of C-Uses and P-Uses are incorrectly defined.

|                     | All elements in a data structure are considered as the same | Using enhanced rules as described in section 4.5 |
|---------------------|---|--|
| No. of C-Uses found | 5073  | 3156   |
| No. of P-Uses found | 3066  | 1589   |

### 6.3.3 Comparing ATACOBOL, ATAC and ASSET

The following table compares currently available coverage tools, including ATACOBOL for COBOL language described in this report, ATAC for C language, and ASSET (A System to Select and Evaluate Tests) for PASCAL as described in [Fra88]. All-defs is a weak criteria and it is still uncertain about its application. All-du-paths is the most demanding criteria, however impose unreasonable measurement complexity.

| Tools                                   | ATACOBOL                           | ATAC  | ASSET                              |
|---|------------------------------------|---|------------------------------------|
| Measure Block Coverage                  | ✓                                  | ✓   | ✓                                  |
| Measure Edge Coverage                   | ✓                                  | ✓   | ✓                                  |
| Measure All-C-Uses Coverage             | ✓                                  | ✓   | ✓                                  |
| Measure All-P-Uses Coverage             | ✓                                  | ✓   | ✓                                  |
| Measure All-Defs                        | ×                                  | ×   | ✓                                  |
| Measure All-DU-Paths                    | ×                                  | ×   | ✓                                  |
| Measurement Range                       | Within Paragraphs                  | Within Functions                              | No limitation                      |
| Display uncovered construct             | ✓<br>(only block no. is displayed) | ✓<br>(Automatic display of source statements) | ✓<br>(only block no. is displayed) |
| Graphical display of Control Flow graph | ×                                  | ×   | ✓                                  |

#### 6.3.4 Suggested ATACOBOL Further Development

From our experience and the comparison of the last section, we could consolidate some aspects of ATACOBOL that demands improvements.

- The display of uncovered construct by ATACOBOL is relative primitive, users have to follow line numbers stated in the files to trace the source statement. Therefore, ATACOBOL should be enhanced to display the source statement automatically. In the current implementation, the source line numbers have been already stored with the block numbers in control flow information file. It makes the system readily to this enhancement.
- The identification of def-use pair is limited within discrete paragraphs. Some uses of global variables may not be found as illustrated in 6.2.1. Therefore, to measure the coverage thoroughly over the whole program, ATACOBOL should be enhanced to identify def-use pairs across paragraphs.
- Support graphical presentation of control flow graphs and data flow graphs to facilitate test cases construction. Please refer to 7.9 for further discussions.

- The processing algorithms applied in the current ATACOBOL implementations is primitive. Effective algorithms can be applied to increase the processing effectiveness of ATACOBOL.
- To incorporate ATACOBOL to automatic testing tools. A common practice of Y2K problem testing is to inject a large amount of production data into programs with date fields aged by automatic tools, and observe if any exception occurs. ATACOBOL can hereby serve to verify the satisfaction of Y2K date comparison related paths.

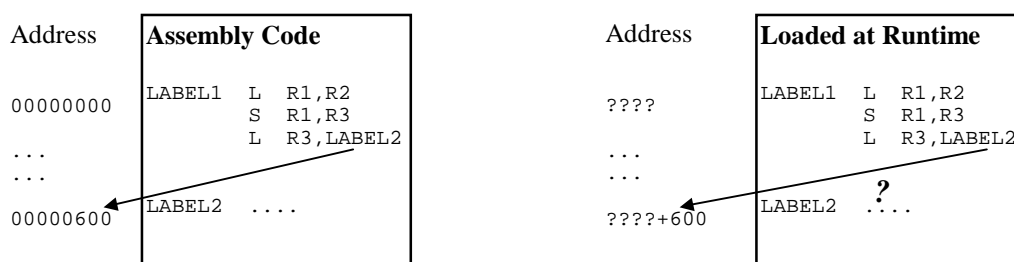


## 7. DISCUSSION

### 7.1 Another Type of Def/Use in Assembly Language — Addressing Use

As discussed in Chapter 2, Rapps and Weyuker have identified two type of def/use pairs: c-use and p-use in their idealised language. When considering assembly language, for example IBM 370 assembly language or the 80X86 assembly language, we can identify another type of def/use pair. This kind of def/use is for the purpose of addressing, so we call it addressing use or simply *a-use*.

In assembly language programming for modern systems, the address loaded for program execution is assigned at runtime (Figure 7.1). Hence, it is impossible to use exact addressing during programming. A base address pointer is used for relative addressing, pseudo codes is usually employed to instruct the compiler to make assumption of the base pointer to some known label for addressability calculations.



**Figure 7.1** Addressing Problem for Program Address Assigned at Runtime

Let us use IBM 370 Assembly Language for mainframes as an example. In 370 assembly language, it defines 16 registers: R0 - R15. Except R0, all registers can be applied as base address pointers. When a program is called from the OS, its address is passed by R15, the following is a sample piece of coding:

```

(OS loads Runtime address of LABEL1 to R15)
LABEL1      USING *,R15          ESTABLISH LOCATION FROM OS
            LA    R1,LABEL2      LOCATE LABEL2
            USING LABEL2,R1      ESTABLISH ADDRESSABILITY
            MVC   FRUIT,APPLE     MOVE CHARS 'APPLE' TO VAR FRUIT

LABEL2      EQU    *
APPLE       DC    C'APPLE'
FRUIT       DC    C'      '

```

The first USING pseudo code instructs the assembler to make assumption on using R15 as the based register for the whole program code segment. The runtime address of LABEL2 is loaded to R1. The second USING pseudo code instructs assembler to use R1 as the base address in assembling all labels after LABEL2. Note that USING has similar meaning as pseudo code ASSUME in 80X86, but of higher flexibility.

The instruction to load the address and USING pseudo code together *defines* the addressability of a base registers. For any codes using labels under the addressability (visibility) of that defined register, that may considered as the *use* of the addressability, should be assembled with the correct base register and executed with the correct addressing at runtime.

Incorrect addressing may cause wrong use of data and wrong branching. More severely, it may cause corruption of data or even the code itself. Subsequently, it may cause unrecoverable error, although MVS provides exception handling and recovery for software errors as described in [Iyer96]. The checking of a-use is of essential significance for any assembly language programmer.

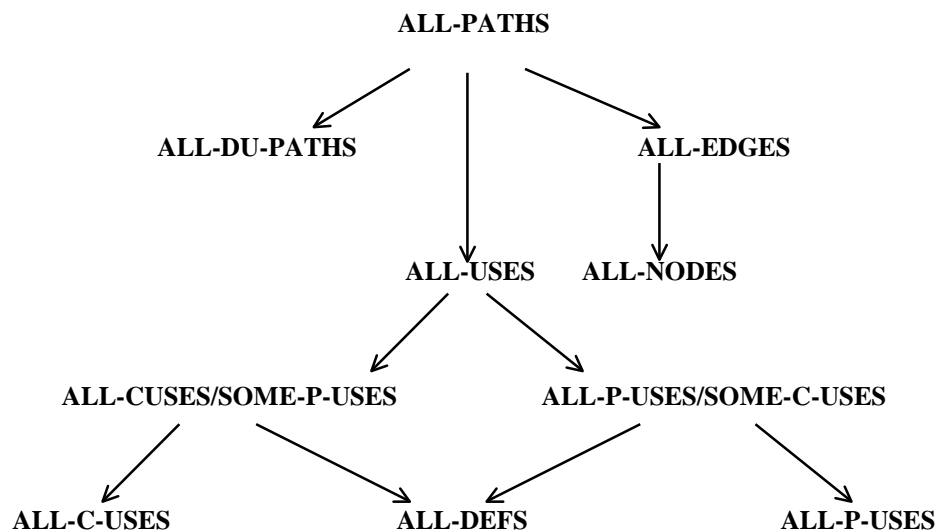
## 7.2 Effect of Unexecutable Path to Data Flow Coverage Criteria

Given a program  $P$  and a data flow coverage criterion  $C$ , it may be the case that no test set for  $P$  satisfies  $C$ . This occurs when none of the paths which cover a particular

association required by  $C$  is executable. In such a case,  $P$  cannot be adequately tested according to  $C$ . [Fra88] introduced a new family of criteria derived from the data flow criteria we had described in Chapter 2 which cater the existence of unexecutable paths.

We say that a complete path is executable or feasible if there exists some assignment of values of variables which causes the path to be executed. Whether or not a particular path is executable depends on the actual context of  $P$ , not just on its def-use graph. An association is executable if there is some executable complete path which covers it, otherwise, it is unexecutable.

Base on the existence of unexecutable paths, [Fra88] proves the subsumption relationship as figure 7.2.



**Figure 7.2**

In this case, the subsumption between all-uses to all edges is broken. Therefore, we cannot just measure all-uses and suppose the measurement can covers all edges coverage. However, if we employed the measurement strategy that starts with the measurement of a relatively weaker criteria, say edge coverage, upon the fulfilment of

such criteria, we then proceed with a stronger criteria, say all-uses. We can represent a general subsumption relationship by union of these criteria:

$$\text{all-uses coverage} \cup \text{all edges coverage} \cup \text{block coverage} \supseteq \text{block coverage}$$
$$\Rightarrow \text{all edges coverage} \cup \text{block coverage}$$
$$\Rightarrow \text{block coverage}$$

Therefore, it is not important for a criterion to completely subsume another criterion.

If a criterion can subsume the other in majority, the above measurement strategy can be applied smoothly.

For any of the path selection testing criteria (including block coverage, edge coverage and the data flow coverage) there can be no algorithm to decide, in general, whether or not such a path segment is executable. From the software testing experiment using ASSET in [Wey90] shows that human beings are frequently very good at determining unexecutability. (In section 7.9, we discuss how this ability can be enhanced by visual-aid.)

One helpful simple facility to assist the tester in dealing with the unexecutable path problem may be programmed. If a path or definition/use pair has been determined unexecutable, this facility checks other unexercised paths to see if they contain any of the known unexecutable subpaths. Any path containing an unexecutable subpath is clearly unexecutable, and is automatically removed from consideration.

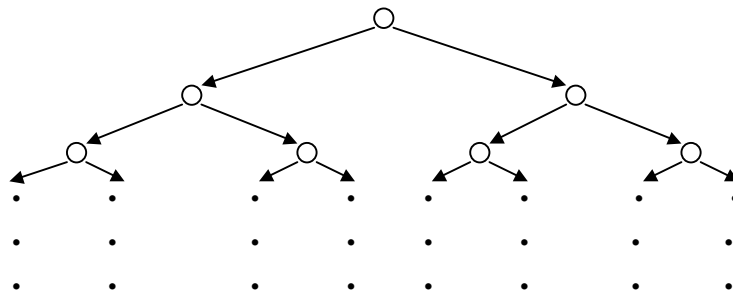
### **7.3 Complexity of Data Flow Coverage**

#### 7.3.1 Complexity of Data Flow Coverage Criteria

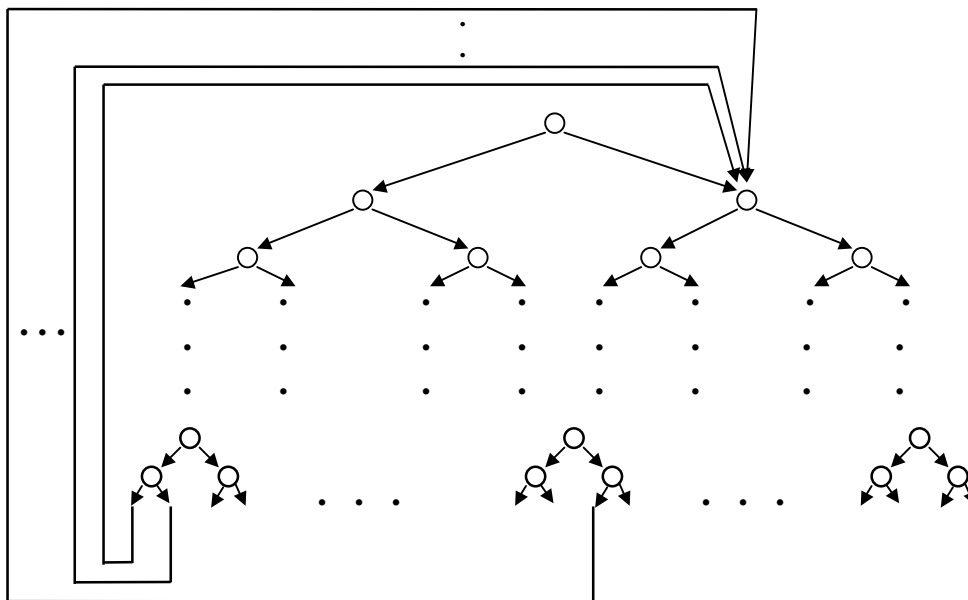
The complexity of data flow coverage criteria is considered as the number of test cases required to satisfy the criteria in [Wey84]. Such upper bound can be determined

theoretically. Let  $P$  be a program with  $n$  variables,  $m$  assignments,  $i$  input statements, and  $t$  conditional transfers. Let a test case consist of a single vector of input variables. Letting  $d$  be the number of defs in  $P$ , it follows that  $d \leq m + (i \times n)$  since each input statement defines between one and  $n$  variables.

For all-nodes and all-edges criteria, it requires at most  $t + 1$  test cases. Figure 7.3 outlines the control-flow graph that shows this extreme case.



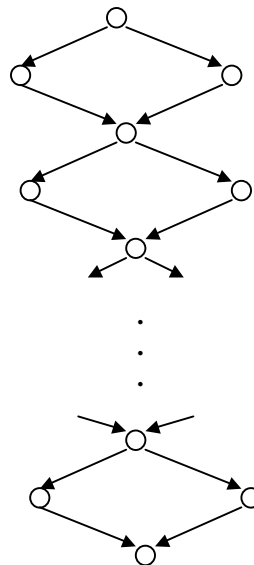
**Figure 7.3** Control-flow graph meets the upper bound of edge and node coverage



**Figure 7.4** Control-flow graph meets the upper bound of data flow coverage

Figure 7.4 shows the upper bound case for all-uses. This type of flow graph maximizes the number of def-use pairs and thus the number of paths to be traversed. There are  $\frac{1}{2}(t + 1)$  nodes in the last decision row of nodes of the graph, representing at most  $t + 1$  distinct uses of the definition of the root node. Of these uses,  $\frac{1}{2}(t + 1)$  are exit nodes, while  $\frac{1}{2}(t + 1)$  can contain defs, each with a use in every node in the other half of the flow graph. Thus there are a total of  $\frac{1}{2}(t + 1) \times \frac{1}{2}(t + 1) + \frac{1}{2}(t + 1) = \frac{1}{4}(t^2 + 4t + 3)$  distinct paths induced by def-use pairs.

For all-du-paths, the worst case for this criterion is a control flow graph which maximises the number of loop-free paths. This occurs when a flow graph has the form outlined in figure 7.5. In that case, there are  $2^t$  distinct loop-free paths.



**Figure 7.5** Control-flow graph that maximises the all du-paths

To summarise, the theoretical upper bound for all-nodes and all-edges criteria is  $t + 1$  test cases. All-uses criteria require at most  $\frac{1}{4}(t^2 + 4t + 3)$  test cases. All du-paths requires at most  $2^t$  test cases.

The result of an empirical study carried out in [Wey90] reveals that the complexity is far lower than the analytical worst condition for practical programs. The experiment is

carried out by a team of professional programmer on a set of well-developed PASCAL programs. The ratio of the number of test cases sufficient to satisfy the all-uses to the number of decision statements is found to be only 3.67. It is also surprised to observe that even though the all-du-paths criterion has an exponential upper bound whereas the all-uses criterion has a quadratic upper bound, in practice test sets sufficient to almost satisfy all-uses were frequently also sufficient to almost satisfy all-du-paths. Since ATACOBOL does not implement the all-du-paths measurement, we cannot take similar measures in our COBOL programs.

### 7.3.2 Complexity of Data Flow Coverage Measurement

Another complexity factor of data flow coverage is the measurement process. The complexity of test cases that fulfils coverage criteria in the previous discussion is directly related to the cost of the whole testing. On the other hand, the measurement complexity is just imposed on occasions when the coverage result is required.

From the ATACOBOL implementation, for a program P, with  $n$  statements, totally  $m$  variables and  $l$  trace records.

Building of Control Flow graph:  $n$

Building Variable Table:  $n \times m$

Building Data Flow graph:  $n \times m$

Instrument the source code:  $n$

Analysis of Block Coverage:  $n \times l$

Analysis of Decision Coverage:  $n \times l$

Search for def-use pairs:  $n^2$

Analysis of Data Flow Coverage:  $n \times m^2 \times l$

In general, the coverage measurement for block and decision coverage is linear while data flow coverage is quadratic.

#### **7.4 Comparison of the Fault-Detecting Ability of Coverage Criteria**

Several testing technique had been proposed in the literature, however, little concrete information about the fault-detecting ability of these criteria has been gathered. One difficult factor that makes it difficult to compare the fault-detecting ability of testing techniques is that typically a large number of different test suites satisfy the criterion for a given program. Often, some of these test suites expose a fault, while others do not. Consequently, a reasonable question to ask is whether test suites developed for a given program using one technique are more likely to detect a fault than test suites developed using another technique. In other words, we require to compare the effectiveness of different testing criteria.

##### 7.4.1 Problem of Empirical Comparison of Effectiveness

We may attempt to answer this question by using the empirical approach. However, there are a number of problems associated with this approach. A major problem is that we have no agreed-upon notation of representative program. Therefore, if we show that a given criterion is good at exposing certain faults contained in a set of programs, it is not generally meaningful to apply this result to other programs with different characteristics. [Wey93] also points out the problem of *misleading test*, it is because many experiments cannot covers all possible test cases to give an averaged result for comparison.



### 7.4.2 Problem of Analytical Comparison of Effectiveness

Previous analytical comparisons of testing criteria have been based on the subsumes relation as presented in Chapter 2. If  $C_1$  subsumes  $C_2$ ,  $C_1$  can always detect a fault that can be detected by  $C_2$ . This is denoted by  $C_1$  *BETTER*  $C_2$ . However, even if  $C_1$  subsumes  $C_2$ , it is possible for  $C_2$  to be more likely detects a fault than  $C_1$ . That means, we have higher probability to detect a fault out of test cases that fulfil  $C_2$ . In order to address this weakness, Weyuker et al. [Wey93] introduced a new relation to serve as the basis for analytical comparing criteria: the *PROBBETTER* relation.

### 7.4.2 PROBBETTER Relation of Coverage Criteria

$C_1$  is said to be *PROBBETTER* than  $C_2$  for a given program if a set case selected randomly from among all those satisfying  $C_1$  for that program is more likely to detect a failure than a test set selected randomly from those satisfying  $C_2$  for that program.

[Fra93] uses the *properly covers* relation to prove that All-uses *PROBBETTER* than All-edges. Informally speaking, the properly covers relation requires that each sub-domain of the domain division induced by  $C_2$  be the union of some of the sub-domains induced by  $C_1$ , with the added requirement that if there are  $n$  copies of a given sub-domain in the  $C_2$  division of the domain, then there must be at least  $n$  copies of that sub-domain in the  $C_1$  division of the domain.

## **7.5 Code Coverage and Reliability**

### 7.5.1 Inadequacy of Operational Profile in Reliability Estimation

Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment [Ieee91]. Another related metric is the Mean Time to Failure (MTTF). Several models have been proposed in the past

to estimate MTTF from failure data generated during system test [Mus87]. These models assume that a system under system test is being tested using an *operational profile* in accordance with which the system may be used. Such black-box models is subjected to criticism by [Hor96] to the fact that parts of the code that remain untested are sites for potential faults.

An argument against this criticism is that if the system is tested using an operational profile, then any untested parts of the code are unlikely to be executed during system operation and hence faults in such parts will not affect the reliability of the software. However, this argument is valid only if an accurate operational profile is available. [Hor96] sites several situations that we usually cannot have an accurate operational profile but just use educated guesses:

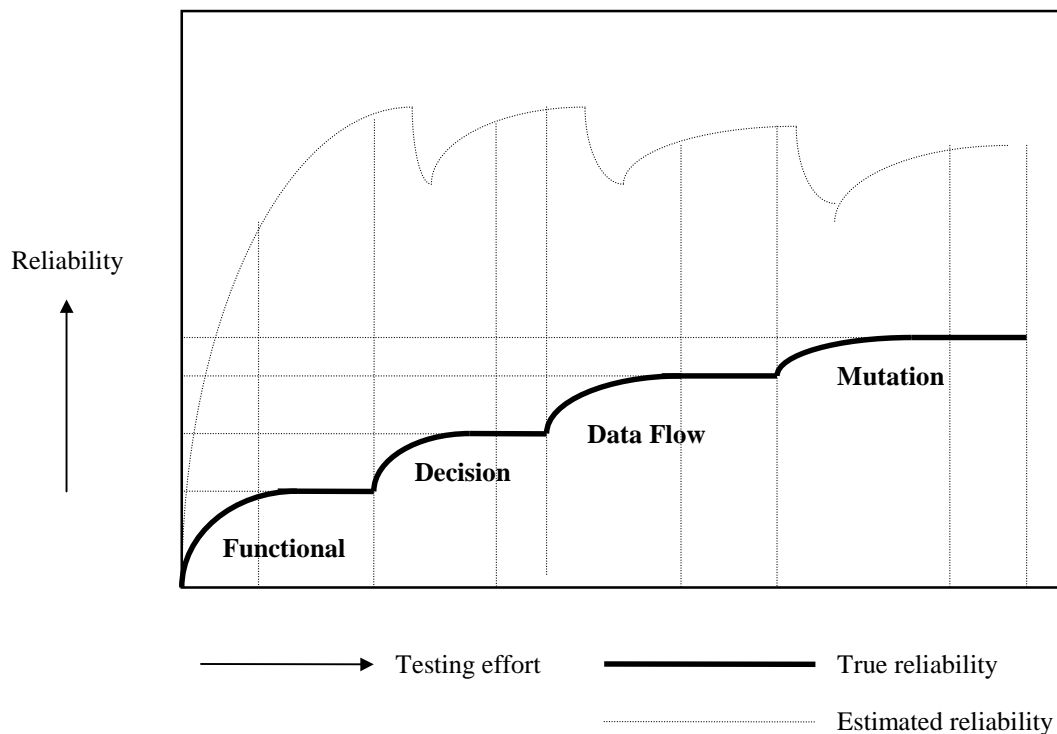
- New software: When a new system is designed, as opposed to modifying an existing system, one may not have any customer base for this system.
- New features: New versions of an existing system may be continually under development. Even though there exists a user base for the existing version of the system, there is no user base for the new version yet to be released.
- Feature definition: A feature is often not a well-defined entity.
- Multiple and unknown user groups:
- Multiple and unknown user groups: An operational profile usually can only reflect users of a relatively homogeneous class. For system of great variety of usage, the operational profile is unable to model the whole group of users accurately.

[Kar96] based on branch coverage to estimate commercial hardware-software system on IBM mainframes operating systems MVS/XA and VM/XA and the result suggest code coverage improves the accuracy of reliability estimation. [Chen96] based on

statement coverage technique to extract only effective data from a given operational profile for reliability estimation in the Iowa/Rockwell Autopilot Project, result indicates that overestimation of reliability is reduced. Stronger coverage measurements like all-uses and mutation should be applied to obtain more accurate result since some test cases would be dropped as ineffective while they do not satisfy the stronger criteria.

### 7.5.2 Overestimation of Reliability due to Saturation Effect

[Hor96] describes the overestimation of reliability due to saturation effect as shown in figure 7.6. We will comment the views of Horgan and Mathur on this issue.



**Figure 7.6** Overestimation of reliability due to saturation effect

A saturation effect refers to the tendency of an individual testing method to attain a limit in its ability to reveal faults in a given program. It is reasonable to assume that as functional testing proceeds, the reliability of the software being tested grows when faults found are removed. However, once its limit has been reached, no additional

faults are found. If existing models for reliability estimation are used, the reliability estimate can still be improved by increasing the number of test cases executed in the saturation region. Horgan and Mathur believe the counting of the test cases is unfair and will over-estimate the reliability.

For the whole argument, we need to address the assumption underneath. We always assume the existence of oracle, i.e., some methods to determine whether or not the output produced by a program is correct. This is the validation of the testing result. However, the correctness of result validation also involves uncertainty. This uncertainty can be minimised by repeatedly testing of the same cases. In practice, report producing program are frequently complained by users of incorrect format after release, it is not because the reports are not generated in testing but the result is overlooked by the testers.

Moreover, we expect that the shape of the graph should be convex. That means, the increase testing effort spent is inversely proportional to the reliability it gains. The reliability gain diminishes as criteria uncovered by the operational profile are much unlikely to happen. [Hor96] considers test effort as the CPU time for executing the test and the number of test cases only. In practice, to enforce the execution of more selective paths, more manpower is spent in determining the test cases. To enforce rare cases that users seldom encounters to happen, we may need extra system configuration and setup. For example, error paths invoked by insufficient memory, file corruption and database inconsistency. In a nutshell, the project manager or the tester should be very selective in handling the unbalanced trade-off between testing effort spent and reliability gained.

## 7.6 Developing and Using ATACOBOL in Cross-Platform Approach: Pros and Cons

Instead of implementing the software testing tools in the target platform like the IBM Code Assistant (Section 3.2.2), the cross platform setup of ATACOBOL enables the instrumentation and analysis tools to be written in C language using Microsoft Visual C++ version 6.0. C language is not commonly applied in mainframe industry nowadays. None of the mainframe production programs in HSBC are written in C language.

There is a trend of using cross platform approach in the mainframe industry as promoted by some mainframe software solution provider including Compuware (Section 3.2.2), Intersolv and Micro Focus. They believe that the workstation coming up much powerful can share the workload of the mainframe and we can be benefited by the graphical user interface of workstations.

The following is the pros and cons of using cross platform approach.

Pros:

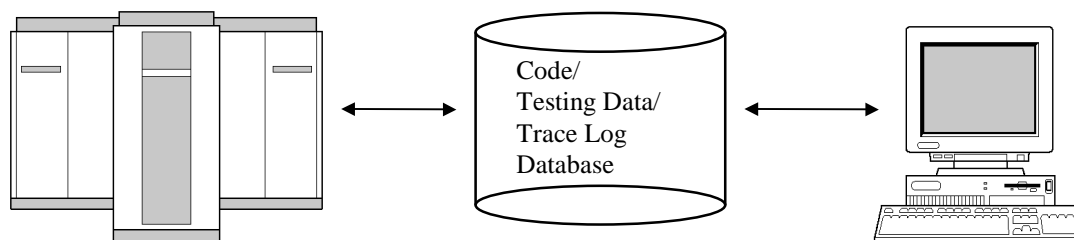
- C language is understood by a larger base.
- C language is supported by a wider base, so the programs developed can be potentially ported to other platforms. (With the introduction of IBM OS/390, it enables UNIX system to run on top of the mainframe OS core, mainly for the benefits of running Web servers. Mainframe Technical Support Team of HSBC is actively evaluating to put this technology to production. We may port the software tool developed in C back to the mainframe in the near future.)
- The coverage measurement tools themselves can be examined under ATAC or  $\chi$ Suds during the development.

- Visual C++ supports graphical user interface that facilitates the viewing of source in further development.
- Higher availability of PC platform .

Cons:

- Analysis time can be shortened by the powerful mainframe.
- The performance is constrained by the file transfer bottleneck.
- Unnecessary data passing overhead between platforms making the whole measurement inconvenience to handle.

For a efficient commercial mainframe product today, it is more reasonable to develop and execute ATACOBOL in IBM 370 assembly language solely on mainframe platform just like IBM Code Assistant. On the other hand, as a research project and taking the fringe benefits in account, the cross platform approach is employed in this project. The system setup of the project is restricted by the HSBC mainframe development environment. We suggest a desirable cross platform program development environment setup as illustrated in figure 7.7. The database may actually maintained by DB2 running on the mainframe. It provides access channels for mainframe as well as the workstations.

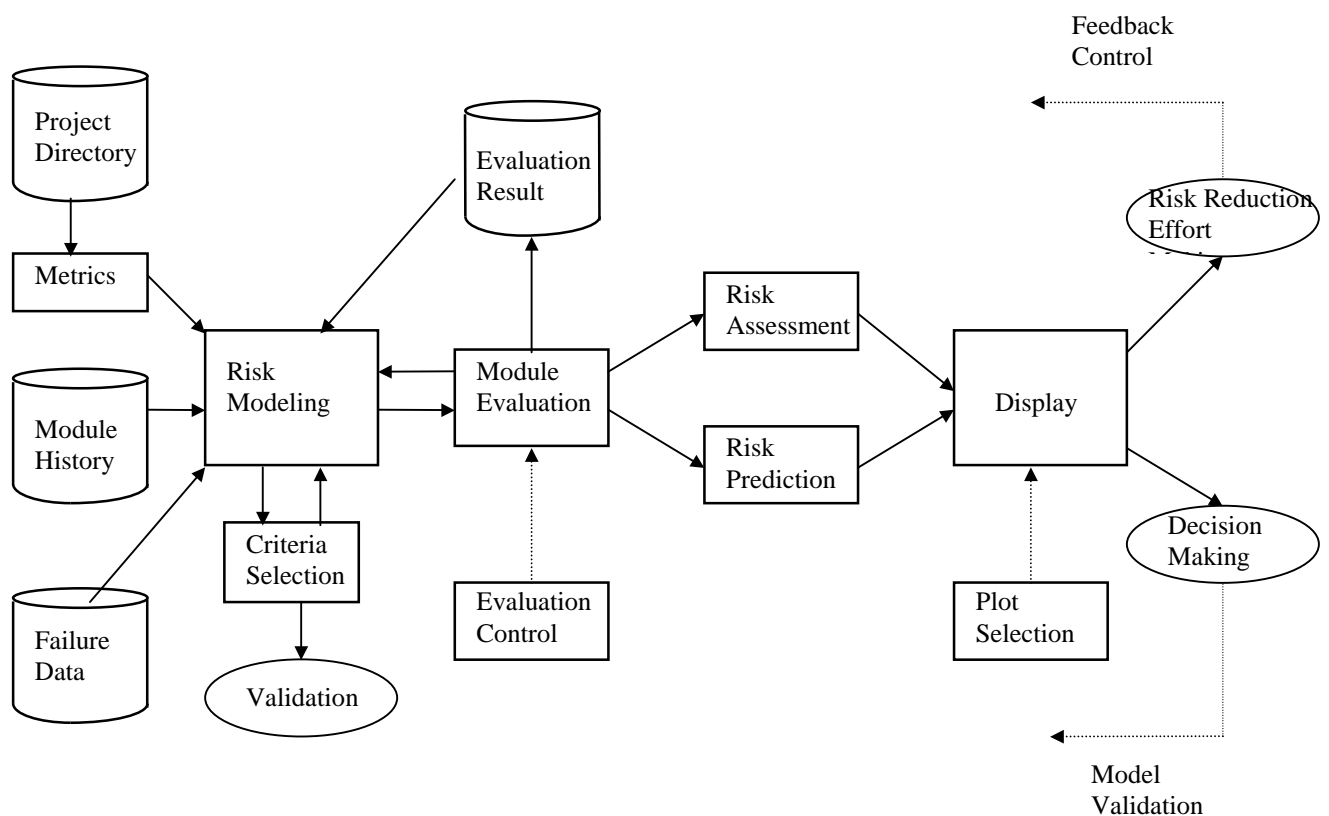


**Figure 7.7**

## 7.7 Incorporation of Coverage Metrics to ARMOR

[Lyu95] presented a software analysis tool ARMOR (Analyzer for Reducing Module Operational Risk) that takes data from project database, failure database, and program development database, establishes risk models and validate the risk models from field data. Figure 7.8 shows the high-level architecture for ARMOR.

To model software risk the quality indicator in the form of metrics of software modules are acquired. ARMOR can already calculate some metrics. [Hor96] described an application of coverage measurement as a metric of risk. Therefore, it is readily to incorporate this new metric into ARMOR.



**Figure 7.8** High-level architecture for ARMOR

The selection and weighting function of ARMOR allows users to select and weight candidate metrics as the basis to form risk models. After the risk model is evaluated,

we could make use of the validation function to study the correlation between the risk model constructed and failure data. As a result, ARMOR could help to:

- Explore whether coverage metric correlates to the failure data;
- Identify whether the coverage metric is the dominating source of risk.

### **7.8 Potential By-Products of ATACOBOL**

In the implementation of ATACOBOL, information of the subjected program is extracted. It includes the variable table, node-link structured control-flow graph, variable table, and data-flow graph of define or use of variables. In addition to coverage measurement, the information extracted can be readily or potentially employed in other software analysis applications.

The information can be applied as the inputs for many software quality metrics, or it may be further manipulated to produce summarised statistics for evaluation.

The node-link structured control-flow graph can be fed to some kinds of simulator that simulates the behaviour of a program.

### **7.9 Visual-Aid for Coverage Analysis**

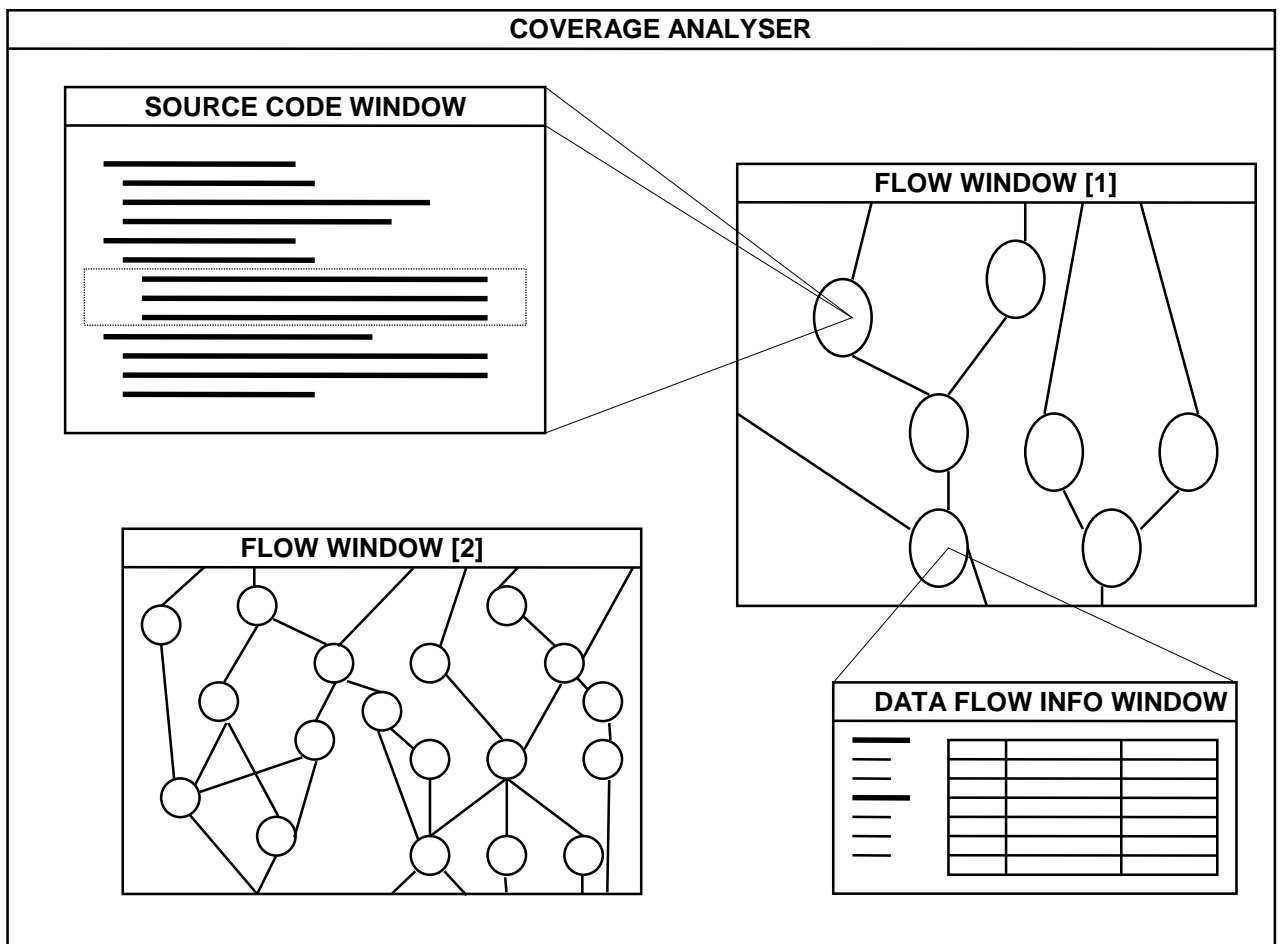
A good quality visual-aid can greatly facilitate the analysis of coverage. Especially for data flow coverage, good representation of the control and data flow information can help the tester to construct test cases that makes a particular path to be executed, or even to determine certain path to be unexecutable.

ATAC is able to highlights the uncovered code. In  $\chi$ Suds, the covered code is colored according to the frequency of execution. However, they do not visualise the control flow or data flow graphs. ASSET visualise the control flow graph. An ideal visual-aid proposed for coverage analysis is illustrated in figure 7.9.



A good visual-aid for coverage analysis should be user-friendly and informative. The proposed visual-aid consists of a set of windows. The *flow window* visualise the program under analysis by control-flow graphs. Zoom-in and zoom-out to the node-link structure is provided. More than one flow window can be opened for viewing the program structure in different scales. On clicking a node of the control-flow graph, the corresponding source statements will be shown on the *source code window*. Moreover, the nodes or links can be clicked to display the *data flow information window*. The data flow information window lists the variables defined/used in that node, and also the def/use graph associated with this node.

This kind of visual organisation is quite similar to computer-aided circuit design or analysis applications in electronic engineering. Designers can trace the circuit paths throughout the graphs and look into individual component specifications. This approach is widely accepted in large scaled but detailed VLSI design.



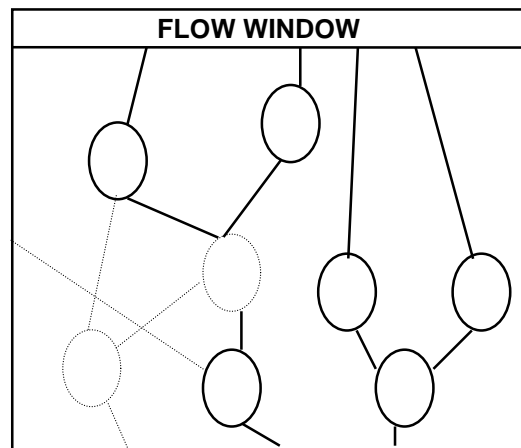
**Figure 7.9** Proposed visual-aid**7.10 Coverage Analysis for Program Version Changes**

Measuring the coverage of a program costs extra effort. If a program is revised and release again, it would be costly if we had to re-measure coverage of the whole program. Moreover, testers may be more concerned with whether the code that has been changed, added or deleted from one release to the next has been properly tested rather than the overall coverage with respect to a module, a subsystem or the entire software. In unit test phase, program may subject to frequently changes. If we need to keep track of coverage strictly all the time, it would be very costly. If a facility can smartly select amended coding smartly, take measure on the amended coding and merge the measurement result, great deal of efforts could be saved.

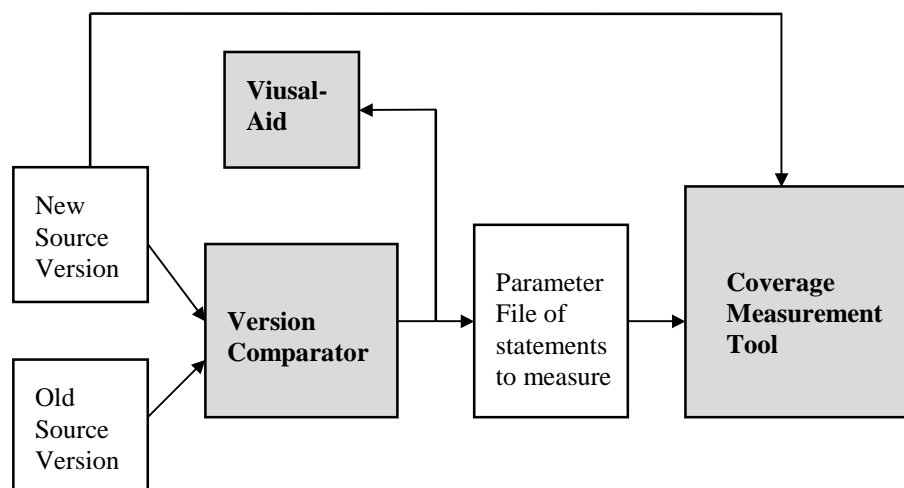
It would be possible for control-flow coverage to incorporate such merged changes coverage measurement. However, it would be far complicated for the highly associated data flow coverage. For the simplest case, if only part of a paragraph is amended, only that paragraph will be accounted for coverage measurement. For detail changes, variables being changed in the amendment may be selected. Then all paths associates with such variables then requires to be re-measured. Visual aid described in section 7.9 may be enhanced to support the viewing of differed paths to help the tester to determine which part of coding required to re-measured.

The IBM Coverage Assistant described in Chapter 3 supports an additional feature to measure coverage only for statements selected by the user. An integration of coverage tools with version comparison tools is proposed in figure 7.11. For  $\chi$ Suds, it supports an automatic *atacdiff* facility in similar approach. Atacdiff compares the versions and generates a parameter file of the changed statement information to  $\chi$ ATAC. In

addition, it identifies regression test cases automatically by selection those test cases have paths likely to pass through the changed statements.



**Figure 7.10** Visual-aid supports version changes viewing



**Figure 7.11** Integrated coverage measurement tool that supports version changes coverage

### 7.11 Data Flow Coverage: To Use or Not to Use?

It has been pointed out that time and money are the most commonly used criteria for determining whether or not it is the right point to stop testing. We test until we have exhausted the funds allocated for testing or until the delivery date has arrived. This is a usual practice experienced by programmer in real-life projects.

Obviously, the time and money criteria have little to do with either the quality of the software, or the quality of the testing performed. If a more sophisticated adequacy criterion is used in practice, it is likely to be code coverage criterion. Although code coverage is not an all-promising to expose all kind of bugs. Nevertheless, these criteria are used in practice and works. [Wey89] sites several reasons of choosing control and data flow coverage as the adequacy criteria:

- They are unambiguous. Different observers do not have different perceptions of whether or not a given code fragment has been exercised by a set of test cases.
- They force a “distribution” of test cases by requiring that every part of the program be exercised. In that way they represent necessary conditions for a comprehensive job of testing. If some part of the code has never been exercised, then we have no idea whether it contains bugs.
- They are easily automated. Tools can be readily built to determine whether or not these criteria have been satisfied.
- They can be quantified; i.e. they can be used to assess the degree of testing performed to date.

In [Dal93], an experiment carried out to compare the statement coverage of unit tests for 28 modules of a single system to the number of system test faults found for each module. The result shows a clear relation that high statement coverage in unit testing and low number of faults detected in system test.

It still needs further experiments to show that high data flow coverage relates to low number of faults. This kind of result is difficult to obtain due to two reasons. The first one is that we cannot easily distinguish the contribution of data flow coverage to the

reliability of a software from control flow coverage in large scale project in statistic measures. It is because we cannot analysis each fault.

Another reason is that the lack of data coverage tools. By the development and usage of ATACOBOL, it enables obtain field data of data flow coverage in COBOL language. The measurement result would reveals the usefulness of data flow coverage in business sectors.

Although time and money criteria cannot offer an objective and directive criteria for software testing, in applying a particular technique, we should not neglect to consider the cost and effectiveness of that technique. In previous section 7.3 and 7.4 discussions, experiments have shown that the cost or the complexity of data flow coverage measurement is reasonable. And, the fault-detecting effectiveness of data-flow coverage is analytically higher than control flow coverage.

## 8. SCHEDULE

### 8.1 Project Implementation Schedule

| Date      | Tasks  |
|-----------|--|
| Jan 99    | Literature Review.   |
| Feb 99    | Coverage Tool Product Survey.  |
| Mar 99    | Code and test ATACOBOL<br>Block Coverage.  |
| Apr 99    | Prepare Term Report<br><br>Code and test ATACOBOL:<br>Decision Coverage.   |
| May 99    | Code and test ATACOBOL:<br>Build Define/Use Graph.   |
| June 99   | Code and test ATACOBOL:<br>Create dcu and dpu sets.  |
| July 99   | Code and test ATACOBOL:<br>Implement data flow coverage analyser;<br><br>Carry out live test cases measurement on<br>mainframe.<br><br>Result analysis |
| August 99 | Prepare Final Report and presentation.   |

### 8.2 Resources

#### 8.2.1 Hardware

- A PC with Cyrix MII 200MHz CPU running MS-Windows 95.
- SUN SPARC Workstation running Solaris 2.5.1.

### 8.2.2 Software

MS-Visual C++ 6.0 Standard Edition.

### 8.2.3 Human Resources

Base on the expense of 8 hours per week constantly,

Total effort =  $8 \times 32 = 256$  head-hours.

## 9. CONCLUSION

We have surveyed literature about coverage technique and evaluated practical software tools applied for coverage techniques. We have identified the limitations of ATAC and have investigated the coverage measure implemented by ATAC based on C language features.

ATACOBOL, a coverage measurement tool for COBOL in mainframe, is designed and implemented. ATACOBOL is written in C language. It carries out instrumentation and measurement across the mainframe and PC platform. ATACOBOL is able to perform block and decision coverage measures.

We have enhanced the rules of data flow coverage to adapt high level data structure. Moreover, the importance of data flow coverage criteria in identifying real-world Y2K related problematic paths is also demonstrated.

ATACOBOL is applied to measure live programs from the banking sector with live test cases. With the extensive application of ATACOBOL, we hope to explore more about the usefulness of data flow coverage, and the relationship between coverage and reliability.



## 10. REFERENCE

- [Aho86] Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques, and Tools*, Addison Wesley, 1986.
- [Bee98] Beecham, B. J., *Monetary and Financial System in Hong Kong*, Hong Kong Institute of Bankers, 2<sup>nd</sup> Edition, 1998.
- [Bell94] Bellcore, *ATAC Tutorial*, enclosed in the ATAC Software Package, release 3.3.13, Bellcore, September 1994.
- [Bell98] Bellcore, *χSuds Software Understanding System User's Manual*, Bellcore, July 1998.
- [CA98] Computer Associates, *CA-TestCoverage/2000 for the MVS Environment*, Computer Associates International Inc., 1998.
- [Chen96] Chen, M., Lyu, M. R., and Wong, E., "An Empirical Study of the Correlation between Code Coverage and Reliability Estimation," *Proceedings of METRICS'96*, Germany, March 1996, pp. 133-141.
- [Choi89] Choi, B. J., DeMillo, R.A., Krauser, E. W., Martin, R. J., Mathur, A. P., Offutt, A. J. and Spafford, E. H., "The Mothra Tool Set," *Proceedings of Hawaii International Conference on System Sciences*, HI, January 3-6, 1989.
- [Clar89] Clarke L. A., Podgurski A., Richardson, D. J., and Zeil, S. J., "A Formal Evaluation of Data Flow Path Selection Criteria," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, 1989, pp. 1318-1332.
- [Dal93] Dalal, S. R., J. R. Horgan, and J. R. Kettenring "Reliable Software and Communication: Software Quality, Reliability, and Safety," *Proceedings of the 15<sup>th</sup> International Conference on Software Engineering*, Baltimore, MD, May, 1993.
- [Fra89] Frankl, Phyllis G., and Weyuker, Elaine J., "An Applicable Family of Data Flow Testing Criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, 1988, pp. 1483-1498.
- [Fra93] Frankl, P.G., and Weyuker, E.J., "A Formal Analysis on the Fault-Detecting Ability of Testing Methods," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, 1993, pp. 202-213.
- [Fra93] Frankl, P.G., and Weyuker, E.J., "An Analytical Comparison of the Fault-Detecting Ability of Data Flow Testing Techniques,"

- Proceedings 15<sup>th</sup> International Conference on Software Engineering*, 1993, pp. 415-424.
- [Ghe97] Ghezzi, C., Jazayeri, M., and Mandrioli, D., *Fundamentals of Software Engineering*, , Prentice Hall, Asia Edition, 1997.
- [How80] Howden, W. E., "Functional Testing," *IEEE Transaction on Software Engineering*, vol. 6, no. 2, 1980, pp. 162-169.
- [HSBC92] Hong Kong and Shanghai Banking Corporation Ltd. CAS Team, *APS User Guide*, 1992. HSBC Internal document.
- [HSBC99] Hong Kong and Shanghai Banking Corporation Ltd., *Technical Service Division (TSV) Handbook*, 1999. HSBC Internal document.
- [Hor90] Horgan, J. R., and London, S., "A Data Flow Coverage Testing Tool for C," *Proceeding of the 2<sup>nd</sup> Symposium on Assessment of Quality Software Development Tools*, 1992, pp. 2-10.
- [Hor94] Horgan, J. R., London, S., and Lyu, M. R., "Achieving Software Quality with Testing Coverage Measure," *IEEE Computer*, vol. 27, no. 9, 1994, pp. 60-69.
- [Hor96] Horgan, J. R., and Mathur, A. P., "Software Testing and Reliability," in *Software Reliability Engineering Handbook*, Lyu M. R. (ed.), McGraw Hill, 1996.
- [IBM97] IBM Corp. , *OS/390 MVS JCL User's Guide*, IBM Corp., 1997.
- [IBM98] IBM Corp., *IBM Application Testing Collection for MVS/ESA Version 1 Release 4 Modification 0*, 5<sup>th</sup> Edition, IBM Corp., December 1998.
- [ICL99] Hong Kong Interbank Clearing Ltd., *Global Payments Systems Test Information Package*, May 1999, Internal document.
- [Ieee91] Institute of Electrical and Electronics Engineers, *ANSI/IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std. 729-1991, 1991.
- [Int96a] Intersolv Inc., *APS Batch Generator Reference Manual*, Intersolv Inc., 1996.
- [Int96b] Intersolv Inc., *APS Program Painter User Manual*, Intersolv Inc., 1996.
- [Int96c] Intersolv Inc., *APS Customization Facility User Manual*, Intersolv Inc., 1996.
- [Int96d] Intersolv Inc., *APS Messages Reference Manual*, Intersolv Inc., 1996.

- [Int96e] Intersolv Inc., *APS/MVS General Index*, Intersolv Inc., 1996.
- [Int96f] Intersolv Inc., *APS/MVS Importers User Manual*, Intersolv Inc., 1996.
- [Iyer96] Iyer, R. K., and Lee, I., "Measurement-Based Analysis of Software Reliability," in *Software Reliability Engineering Handbook*, Lyu M. R. (ed.), McGraw Hill, 1996.
- [Kar96] Karcich R. M., Skibbe R., Mathur, A. P., and Garg, P., "On Software Reliability and Code Coverage," *Proceedings of Aerospace Applications Conference*, 1996.
- [Las83] Laski, J. W., and Korel, B., "A Data Flow Oriented Program Testing Strategy," *IEEE Transactions on Software Engineering*, vol. 9, no. 3, 1983, pp. 347-354.
- [Lyu95] Lyu, M. R., Yu, J. S., Keramidas, E., and Dalal, S. R., "ARMOR: Analyzer for Reducing Module Operational Risk," *Proceeding of FTCS'25*, Pasadena, California, June 1995, pp. 137-142.
- [Lyu96] Lyu, M. R., Horgan, J. R., and London, S., "A Coverage Analysis Tool for the Effectiveness of Software Testing," *IEEE Transaction on Reliability*, vol. 43, no. 4, 1994, pp. 527-535.
- [Lyu98] Lyu, M. R., "Design, Testing, and Evaluation Techniques for Software Reliability Engineering," *Proceedings 24th Euromicro Conference*, Vasteras, Sweden, August 25-27 1998, pp. xxxix-xlvi.
- [Mus87] Musa, J. D., Iannino, A., and Okumoto, K., *Software Reliability — Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.
- [Nta84] Ntafos, S. C., "On Required Element Testing," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, 1984, pp. 795-803.
- [Rap85] Rapps, S., and Weyuker, E. J., "Selecting Software Test Data Using Data Flow Information," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, 1985, pp. 367-375.
- [Som96] Sommerville, I., *Software Engineering*, Addison Wesley, Fifth Edition, 1996.
- [Viasoft98] Viasoft, *VIA/SmartTest Data Sheet*, Viasoft Inc., 1998.
- [Wey84] Weyuker, E. J., "The Complexity of Data Flow Criteria for Test Data Selection," *Information Processing Letter*, vol. 19, no. 2, pp. 103-109.

- [Wey88] Weyuker, E.J., "An Empirical Study of the Complexity of Data Flow Testing," *Proceedings of the 2<sup>nd</sup> Workshop on Software Testing, Verification, and Analysis*, 1998, pp. 188-195.
- [Wey89] Weyuker, E.J., "In Defense of Coverage Criteria," *11<sup>th</sup> International Conference on Software Engineering*, 1989, pp. 361.
- [Wey90] Weyuker, E.J., "The Cost of Data Flow Testing: An Empirical Study," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, 1990, pp. 121-128.
- [Wey93] Weyuker, E.J., "Can We Measure Software Testing Effectiveness?," *Software Metrics Symposium*, 1993, pp. 100-107.

## Appendix A ATACOBOL Program Specifications

ATACOBOL consists of 4 components: Code Parser, Instrumenter, Coverage Analyser and Runtime Routine as described in Chapter 4. The runtime routine is implemented on the mainframe. The 'DISPLAY' system call is substituted as this runtime routine. The rest of these components are implemented on PC Windows 95 Platform. A component may actually implemented by more than one module for the easy of debugging. The sources are written in Visual C++ 6.0 Win32 Console Mode.

### A.1 Module Specifications

#### A.1.1 Normaliser

| Module Name             | Normaliser                                    |
|-------------------------|---|
| Component Category      | Code Parser                                   |
| Module File Name        | NORMER.EXE                                    |
| Module Source           | NORMER.CPP                                    |
| No. of Lines of Coding: | 446   |
| Function                | Normalize program source to record of blocks. |
| Input File              | COBOL.SRC (Program source)                    |
| Output File             | COBOL.STR (Primitive program structure)       |
| Limitation              | N/A   |

#### A.1.2 Router

| Module Name             | Router   |
|-------------------------|--|
| Component Category      | Code Parser                                    |
| Module File Name        | ROUTER.EXE                                     |
| Module Source           | ROUTER.CPP                                     |
| No. of Lines of Coding: | 461  |
| Function                | Route the blocks by edges.                     |
| Input File              | COBOL.STR (Primitive program structure)        |
| Output File             | COBOL.RST (Control Flow Information File)      |
| Limitation              | Each paragraph can have maximum of 100 blocks. |

A.1.3 Variable Table Builder

| Module Name             | <b>Variable Table Builder</b>                                      |
|-------------------------|--|
| Component Category      | Code Parser  |
| Module File Name        | BVAR.EXE   |
| Module Source           | BVAR.CPP   |
| No. of Lines of Coding: | 365  |
| Function                | Base on program source and listing to build variable table.        |
| Input File              | COBOL.SRC (Program source)<br>COBOL.LST (Compiled Program Listing) |
| Output File             | COBOL.VAR (Variable Table File)                                    |
| Limitation              | Support 3 levels of hierarchical data structure.                   |

A.1.4 Data Flow Graph Builder

| Module Name             | <b>Data Flow Graph Builder</b>   |
|-------------------------|--|
| Component Category      | Code Parser  |
| Module File Name        | BDEFUSE.EXE  |
| Module Source           | BDEFUSE.CPP  |
| No. of Lines of Coding: | 562  |
| Function                | Build the data flow graph (def, c-use and p-use)   |
| Input File              | COBOL.SRC (Program source)<br>COBOL.RST (Control Flow Information File)<br>COBOL.VAR (Variable Table File)<br>RESERVE.WRD (S-COBOL Reserved Word File) |
| Output File             | COBOL.DEF (Data Flow Information File (Def))<br>COBOL.CUS (Data Flow Information File (C-Use))<br>COBOL.PUS (Data Flow Information File (P-Use))       |
| Limitation              | The variable table can have maximum of 3000 variables  |

A.1.5 Def-Use Path Searcher

| Module Name             | <b>Def-Use Path Searcher</b>  |
|-------------------------|---|
| Component Category      | Code Parser   |
| Module File Name        | PSEARCH.EXE   |
| Module Source           | PSEARCH.CPP   |
| No. of Lines of Coding: | 630   |
| Function                | Search the function: dcu and dpu based on the data flow graph.                            |
| Input File              | COBOL.RST (Control Flow Information File)<br>COBOL.DEF (Data Flow Information File (Def)) |

|             |  |
|-------------|--|
|             | COBOL.CUS (Data Flow Information File (C-Use))<br>COBOL.PUS (Data Flow Information File (P-Use)) |
| Output File | COBOL.DCU (Data Flow Information File (DCU))<br>COBOL.DPU (Data Flow Information File (DPU))     |
| Limitation  | Each paragraph can have maximum of 100 blocks.   |

### A.1.6 Coverage Analyser

|                         |  |
|-------------------------|--|
| <b>Module Name</b>      | <b>Coverage Analyser</b>   |
| Component Category      | Coverage Analyser  |
| Module File Name        | ANALYSER.EXE   |
| Module Source           | ANALYSER.CPP   |
| No. of Lines of Coding: | 898  |
| Function                | Analysis the coverage of program testing through records of the trace log.   |
| Input File              | COBOL.RST (Control Flow Information File)<br>COBOL.DEF (Data Flow Information File (Def))<br>COBOL.DCU (Data Flow Information File (DCU))<br>COBOL.DPU (Data Flow Information File (DPU))<br>COBOL.LOG (Trace Log) |
| Output File             | BLOCK.UCV (Uncovered Block Report File)<br>EDGE.UCV (Uncovered Decision Edge Report File)<br>DCUSE.UCV (Uncovered DCU Report File)<br>DPUSE.UCV (Uncovered DPU Report File)  |
| Display                 | Coverage Summary   |
| Limitation              | Each paragraph can have maximum of 100 blocks.<br>Each paragraph can have maximum of 1,500 def.<br>Each paragraph can have maximum of 1,500 dcu.<br>Each paragraph can have maximum of 1,500 dpu.                  |

## **A.2 File Layouts**

### A.2.1 Program Primitive Structure

|                   |               |
|-------------------|---------------|
| <b>Field Name</b> | <b>Format</b> |
| Paragraph No.     | Integer       |
| Block No.         | Integer       |
| Block Level No.   | Integer       |
| Block Type        | 1 Character   |
| Source Line No.   | Integer       |

A.2.2 Control Flow Information File

| Field Name                        | Format      |
|-----------------------------------|-------------|
| Paragraph No.                     | Integer     |
| Block No.                         | Integer     |
| Block Level No.                   | Integer     |
| Block Type                        | 1 Character |
| Source Line No.                   | Integer     |
| Repeat the following for 10 times |             |
| From Block No.                    | Integer     |
| To Block No.                      | Integer     |

A.2.3 Variable Table File

| Field Name                  | Format        |
|-----------------------------|---------------|
| Variable Level 1 Identifier | Integer       |
| Variable Level 2 Identifier | Integer       |
| Variable Level 3 Identifier | Integer       |
| Variable Label              | 30 Characters |

A.2.4 Data Flow Information File (Def)

| Field Name                      | Format  |
|---------------------------------|---------|
| Paragraph No.                   | Integer |
| Block No.                       | Integer |
| Def Variable Level 1 Identifier | Integer |
| Def Variable Level 2 Identifier | Integer |
| Def Variable Level 3 Identifier | Integer |

A.2.5 Data Flow Information File (C-Use)

| Field Name                        | Format  |
|-----------------------------------|---------|
| Paragraph No.                     | Integer |
| Block No.                         | Integer |
| C-Use Variable Level 1 Identifier | Integer |
| C-Use Variable Level 2 Identifier | Integer |
| C-Use Variable Level 3 Identifier | Integer |



A.2.6 Data Flow Information File (P-Use)

| Field Name                        | Format  |
|-----------------------------------|---------|
| Paragraph No.                     | Integer |
| Block No.                         | Integer |
| P-Use Variable Level 1 Identifier | Integer |
| P-Use Variable Level 2 Identifier | Integer |
| P-Use Variable Level 3 Identifier | Integer |
| To Block No.                      | Integer |

A.2.7 Data Flow Information File (DCU)

| Field Name                          | Format  |
|-------------------------------------|---------|
| Paragraph No.                       | Integer |
| Def-Use Variable Level 1 Identifier | Integer |
| Def-Use Variable Level 2 Identifier | Integer |
| Def-Use Variable Level 3 Identifier | Integer |
| Def Block No.                       | Integer |
| C-Use Block No.                     | Integer |

A.2.8 Data Flow Information File (DPU)

| Field Name                          | Format  |
|-------------------------------------|---------|
| Paragraph No.                       | Integer |
| Def-Use Variable Level 1 Identifier | Integer |
| Def-Use Variable Level 2 Identifier | Integer |
| Def-Use Variable Level 3 Identifier | Integer |
| Def Block No.                       | Integer |
| P-Use From Block No.                | Integer |
| P-Use To Block No.                  | Integer |

## Appendix B ATACOBOL Tutorial

This tutorial make uses of a simple S-COBOL composite interest calculation program to demonstrate the usage of ATACOBOL. The program have two input datasets: SPARMI and CARDI, and one output dataset: RESULTO. SPARMI stores system parameter today's date. CARDI is the input card that inputs the deposit amount and date of deposit. RESULTO shows the calculation to be fail or success with the calculated total amount. The program illustrates the def-use pair of the Y2K example described in Chapter 2. Also, in composite interest calculation, recursive summation is applied and hence recursive def-use pair is observed. The program source and macros are listed as follows:

### COBOL Macro: SPARMIF

```
BLOCK CONTAINS 0 CHARACTERS
LABEL RECORDS ARE OMITTED
DATA RECORD IS SPARM-IN.
*
* 01 SPARM-IN. SYSTEM PARAMETERS
* SPARM-IN-REC
* 05 FILLER OCCURS 80 TIMES
PIC X.
```

### COBOL Macro: SPARMIQ

```
ASSIGN TO UT-S-SPARMI.
```

### COBOL Macro: SPARMIW

```
*
* 01 SPARM-WORK. SYSTEM PARAMETER
*
* 05 SYS-DATE.
*
* 10 SYS-YEAR PIC 9(2).
* 10 SYS-MONTH PIC 9(2).
* 10 SYS-DAY PIC 9(2).
```

### COBOL Macro: CARDIF

```
BLOCK CONTAINS 0 CHARACTERS
LABEL RECORDS ARE OMITTED
DATA RECORD IS CARD-IN.
01 CARD-IN.
05 FILLER OCCURS 80 TIMES
PIC X.
```

**COBOL Macro: CARDIQ**

```
ASSIGN TO UT-S-CARDI.
```

**COBOL Macro: CARDIW**

```
01 CARD-WORK.
  *
      INPUT CARD
05 CARD-DEPOSIT-YEAR PIC 9(2).
05 CARD-DEPOSIT-AMOUNT PIC 9(10).
05 FILLER PIC X(68).
```

**COBOL Macro: RESULTOF**

```
LABEL RECORDS ARE OMITTED
RECORD CONTAINS 80 CHARACTERS
BLOCK CONTAINS 0 RECORDS
DATA RECORD IS RESULT-OUT.
SKIP1
01 RESULT-OUT PICTURE X(80).
```

**COBOL Macro: RESULTOQ**

```
ASSIGN TO UT-S-RESULTO.
```

**COBOL Macro: RESULTOW**

```
*
01 RESULT-WORK.
  *
      CALUATION RESULT
05 RESULT-REASON PIC X(10).
05 RESULT-AMOUNT PIC X(10).
```

**S-COBOL Program Source: SAMPLE**

```
REM
SKIP3
PROGRAM-ID. SAMPLE
SKIP1
AUTHOR. SZE KWAN SHAN, SAM
SKIP1
DATE-WRITTEN. 28 JUN 1999
SKIP1
DATE-COMPILED.
SKIP1
REMARKS.
SKIP1
ABSTRACT
THIS PROGRAM CALUATES COMPOSITE INTEREST
EJECT
AMENDMENT HISTORY
EJECT
OPERATING INSTRUCTIONS
SKIP3
1. CONTROL CARDS
SKIP1
//SAMPLE EXEC PGM=SAMPLE
//SPARMI DD DSN=SYSTEM.PARMETER.DATASET,
// DISP=OLD,UNIT=DISK,VOL=SER=SSSSSS
//CARDI DD DSN=INPUT.CARD.DATASET,
// DISP=OLD,UNIT=DISK,VOL=SER=SSSSSS
//RESULTO DD DSN=OUTPUT.RESULT.DATASET,
// DISP=OLD,UNIT=DISK,VOL=SER=SSSSSS
//SYSUDUMP DD SYSOUT=*
//SYSOUT DD SYSOUT=*
//SYSDBOU DD SYSOUT=*
//SYSDTERM DD SYSOUT=*
SKIP3
2. INPUT
SKIP1
SPARMI - SYSTEM PARAMTER FILE
CARDI - INPUT CARD FILE
SKIP3
3. OUTPUT
SKIP2
RESULTO - CALUATED INTEREST RESULT FILE
SKIP3
4. INTERMEDIATE EXTERNAL STORAGE
```

```

SKIP1
NONE
SKIP3
5. MESSAGES
SKIP1
A. IBM STANDARD MESSAGES
6. COMPLETION CODES
SKIP1
0000 - NORMAL COMPLETION
0016 - ABNORMAL END WITH ERROR MESSAGES
SKIP3
7. NOTES
SKIP1
NONE
SKIP3
8. OPERATING SYSTEM
SKIP1
OS/390,MVS/SP
EJECT
PROGRAM OPERATION
SKIP2
SKIP2
MAINLINE
CALUATE-PERIOD-RTN
CALUATE-INTEREST-RTN
SKIP1
EJECT
DATA FORMATS
SKIP3
PLEASE REFER TO DATA DIVISION
SKIP3
SUBPROGRAMS
SKIP3
NONE
SKIP3
STORAGE LAYOUT
SKIP3
ABOUT 1K
SKIP3
PROGRAM LIMITATION
SKIP3
NONE
EJECT
IO
SKIP3
SELECT SPARMI COPY SPARMIQ.
SKIP2
SELECT CARDI COPY CARDIQ.
SKIP2
SELECT RESULTO COPY RESULTOQ.
EJECT
FD SPARMI COPY SPARMIF.
SKIP3
FD CARDI COPY CARDIF.
SKIP3
FD RESULTO COPY RESULTOF.
EJECT
WS
SKIP3
FRFM 77 FILLER PIC X(35) VALUE
'SAMPLE WORKING STORAGE STARTS HERE.'.
SKIP3
01 SPARM-WORK COPY SPARMW.
EJECT
01 CARD-WORK COPY CARDW.
EJECT
01 RESULT-WORK COPY RESULTW.
EJECT
REC PGM-WORK-AREA
SKIP3
CALUATE-WORK
TEMP-AMOUNT N10 V ZERO
DEPOSIT-YEAR N4 V ZERO
DEPOSIT-PERIOD PIC S9(4) V +0
REASON-DES
RESULT-SUCCESS X10 V 'SUCCESS'
RESULT-FAIL X10 V 'FAIL'
RESULT-STARS X10 V '*****'
SKIP3
CONSTANTS
ANNUAL-RATE N2 V 4
SKIP3
COUNTER N(2) V ZERO
EJECT
PROC
SKIP3
OPEN INPUT SPARMI
... CARDI
... OUTPUT RESULTO
SKIP3
READ CARDI INTO CARD-WORK
READ SPARMI INTO SPARM-WORK
SKIP3
PERFORM CALUATE-PERIOD-RTN
SKIP3
IF DEPOSIT-PERIOD > ZERO
PERFORM CALUATE-INTEREST-RTN
RESULT-REASON = RESULT-SUCCESS
TEMP-AMOUNT = CARD-DEPOSIT-AMOUNT
RESULT-AMOUNT = TEMP-AMOUNT
ELSE
RESULT-REASON = RESULT-FAIL
RESULT-AMOUNT = RESULT-STARS
SKIP3
WRITE RESULT-OUT FROM RESULT-WORK
CLOSE SPARMI
... CARDI
... RESULTO
SKIP3

```

```

GOBACK
EJECT
/*      SUBROUTINE - CALUATE-PERIOD-RTN
SKIP1
/*      ENTRY POINT - CALUATE-PERIOD-RTN
SKIP1
/*      DESCRIPTION - CALUATE THE DEPOSIT PERIOD
SKIP3
PARA  CALUATE-PERIOD-RTN
SKIP3
IF CARD-DEPOSIT-YEAR > 49
    DEPOSIT-YEAR = 1900 + CARD-DEPOSIT-YEAR
ELSE
    DEPOSIT-YEAR = 2000 + CARD-DEPOSIT-YEAR
SKIP3
IF SYS-YEAR > 49
    DEPOSIT-PERIOD = 1900 + SYS-YEAR - DEPOSIT-YEAR
ELSE
    DEPOSIT-PERIOD = 2000 + SYS-YEAR - DEPOSIT-YEAR
EJECT
/*      SUBROUTINE - CALUATE INTEREST ROUTINE
SKIP1
/*      ENTRY POINT - CALUATE-INTEREST-RTN
SKIP1
/*      DESCRIPTION - CALUATE INTEREST OVER THE PERIOD
SKIP3
PARA  CALUATE-INTEREST-RTN
SKIP3
COUNTER = 0
WHILE COUNTER < DEPOSIT-PERIOD
    CARD-DEPOSIT-AMOUNT = CARD-DEPOSIT-AMOUNT *
    ... ANNUAL-RATE / 100 + CARD-DEPOSIT-AMOUNT
    COUNTER = COUNTER + 1
EJECT

```

**STEP 1**

Give the S-COBOL program the program name **COBOL.SRC**.

Duplicate **COBOL.SRC** with name **COBOL.TMP** for indexing by ATACOBOL, in

Windows 95 DOS prompt, enter command:

**COPY COBOL.SRC COBOL.TMP**

**STEP 2**

In Windows 95 DOS prompt, execute

**NORMER.EXE**

**ROUTER.EXE**

**INSTRUER.EXE**

The output instrumented file **COBOL.INT** is created.

**STEP 3**

Transfer COBOL.INT to mainframe dataset. Compile the program source with macros using the APS COBOL compiler. The **compile listing** and **executable module** are obtained.

**STEP 4**

Perform testing with the following test datasets:

| Test Case | Description  | SPARMI   | CARDI                          | RESULTO<br>(expected output)          |
|-----------|--|----------|--------------------------------|---------------------------------------|
| 1         | Both dates in 19XX   | Year: 98 | Year: 99<br>Amount: \$1,000.00 | Status: SUCCESS<br>Amount: \$1,040.00 |
| 2         | Both dates in 20XX   | Year: 01 | Year: 02<br>Amount: \$1,000.00 | Status: SUCCESS<br>Amount: \$1,040.00 |
| 3         | Deposit date in 1999<br>Today's date is 2000.                                | Year: 00 | Year: 99<br>Amount: \$1,000.00 | Status: SUCCESS<br>Amount: \$1,040.00 |
| 4         | Deposit date in 20XX<br>Today's date in 19XX.                                | Year: 99 | Year: 00<br>Amount: \$1,000.00 | Status: FAIL<br>Amount:<br>*****      |
| 5         | Deposit period more than<br>1 year (to demonstrate<br>recursive calculation) | Year: 01 | Year: 97<br>Amount: \$1,000.00 | Status: SUCCESS<br>Amount: \$1,169.86 |

The **trace log** is copied from the JES2 held queue.

### **STEP 5**

Transfer the compile listing and trace log back to PC.

### **STEP 6**

Name the compile listing as **COBOL.LST**

In Windows 95 DOS prompt, execute

**BVAR.EXE**

**BDEFUSE.EXE**

**PSEARCH.EXE**

### **STEP 7**

Name the trace log of test case 1 as **COBOL.LOG** and execute **ANALYSER.EXE**.

The analysed output is as follows:

```

Total Para Number=3
Para #   Block Cover      Decision Cover  C-Use Cover    P-Use Cover
-----
1        4/5      (80%)    1/2      (50%)    5/9      (55%)    1/2      (50%)

```

|       |       |        |     |        |       |       |      |       |
|-------|-------|--------|-----|--------|-------|-------|------|-------|
| 2     | 5/7   | (71%)  | 2/4 | (50%)  | 3/8   | (37%) | 2/4  | (50%) |
| 3     | 3/3   | (100%) | 2/2 | (100%) | 3/5   | (60%) | 5/6  | (83%) |
| ----- |       |        |     |        |       |       |      |       |
| Total | 12/15 | (80%)  | 5/8 | (62%)  | 11/22 | (50%) | 8/12 | (66%) |

We can always refer to output text files: BLOCK.UCV, EDGE.UCV, DCUSE.UCV and DPUSE.UCV for uncovered blocks, decision edges, def-c-uses and def-p-uses respectively. It requires to look into the program structure related files as shown in Appendix A to locate the exact source line number of corresponding block.

### **STEP 8**

Merge trace log of test case 1 and 2 to form COBOL.LOG. This two cases fulfil edge coverage in paragraph 2, however, the cross century cases is not included to fulfilled all c-uses. After the execution of ANALYSER.EXE, the result should look like:

| Total Para Number=3 |             |        |                |        |             |       |             |        |
|---------------------|-------------|--------|----------------|--------|-------------|-------|-------------|--------|
| Para #              | Block Cover |        | Decision Cover |        | C-Use Cover |       | P-Use Cover |        |
| -----               |             |        |                |        |             |       |             |        |
| 1                   | 4/5         | (80%)  | 1/2            | (50%)  | 5/9         | (55%) | 1/2         | (50%)  |
| 2                   | 7/7         | (100%) | 4/4            | (100%) | 6/8         | (75%) | 4/4         | (100%) |
| 3                   | 3/3         | (100%) | 2/2            | (100%) | 3/5         | (60%) | 5/6         | (83%)  |
| -----               |             |        |                |        |             |       |             |        |
| Total               | 14/15       | (93%)  | 7/8            | (87%)  | 14/22       | (63%) | 10/12       | (83%)  |

### **STEP 9**

Merge trace log of test cases 1 to 4 to fulfil all-uses in paragraph 2. However, the previous cases just tested the WHILE loop in paragraph 3 for 1 time. The WHILE loop should be looped more than 1 time to invoke the recursive def-use pair within the WHILE loop. After the execution of ANALYSER.EXE, the result should look like:

| Total Para Number=3 |             |        |                |        |             |        |             |        |
|---------------------|-------------|--------|----------------|--------|-------------|--------|-------------|--------|
| Para #              | Block Cover |        | Decision Cover |        | C-Use Cover |        | P-Use Cover |        |
| -----               |             |        |                |        |             |        |             |        |
| 1                   | 5/5         | (100%) | 2/2            | (100%) | 9/9         | (100%) | 2/2         | (100%) |
| 2                   | 7/7         | (100%) | 4/4            | (100%) | 8/8         | (100%) | 4/4         | (100%) |
| 3                   | 3/3         | (100%) | 2/2            | (100%) | 3/5         | (60%)  | 5/6         | (83%)  |
| -----               |             |        |                |        |             |        |             |        |
| Total               | 15/15       | (100%) | 8/8            | (100%) | 20/22       | (90%)  | 11/12       | (91%)  |

### **STEP 10**

Merge all 1 to 5 test cases to fulfil all-uses for the whole program:

| Total Para Number=3 |             |        |                |        |             |        |             |        |
|---------------------|-------------|--------|----------------|--------|-------------|--------|-------------|--------|
| Para #              | Block Cover |        | Decision Cover |        | C-Use Cover |        | P-Use Cover |        |
| -----               |             |        |                |        |             |        |             |        |
| 1                   | 5/5         | (100%) | 2/2            | (100%) | 9/9         | (100%) | 2/2         | (100%) |

---

|       |       |        |     |        |       |        |       |        |
|-------|-------|--------|-----|--------|-------|--------|-------|--------|
| 2     | 7/7   | (100%) | 4/4 | (100%) | 8/8   | (100%) | 4/4   | (100%) |
| 3     | 3/3   | (100%) | 2/2 | (100%) | 5/5   | (100%) | 6/6   | (100%) |
| ----- |       |        |     |        |       |        |       |        |
| Total | 15/15 | (100%) | 8/8 | (100%) | 22/22 | (100%) | 12/12 | (100%) |