NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-56465-2

AN ADAPTIVE LCSAJ TEST STRATEGY

by

Kerwyn Martin Roach

A thesis

submitted to the School of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

Master of Computer Science

University of Ottawa

Ottawa, Ontario, Canada

November, 1988

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ABSTRACT

A fundamental problem associated with white-box testing is the presence of infeasible paths. Adaptive versions of some white-box test case construction strategies have recently been developed in an attempt to avoid selecting these paths. We propose an adaptive white-box test case construction strategy that achieves Linear Code Sequence And Jump (LCSAJ) coverage. A proof of completeness of coverage is presented for this strategy. In addition, we give a technique for implementing the proposed test case construction strategy. Finally, we demonstrate the applicability of this strategy with a detailed example.

Our strategy provides practical guidance for selecting LCSAJs to be exercised. Moreover, our strategy is not as hampered by infeasible paths as most *a priori* LCSAJ coverage testing strategies.

Key words : Software testing, white-box testing, adaptive testing, infeasible paths, LCSAJ coverage.

ACKNOWLEDGEMENTS

I am indebted to my thesis supervisor Dr. Robert Probert for his support and advice. Dr. Probert suggested this topic, and his encouragement and guidance proved invaluable. I would also like to thank Dr. Hasan Ural for his informative discussions.

I am grateful to the government of Trinidad and Tobago for offering me the opportunity to further my education in Canada, and to the Canadian government for the financial support provided by a Canadian Commonwealth Scholarship under the Canadian Commonwealth and Fellowship Plan.

DEDICATION

à ma famille.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOTATION

Mathematical conventions

$\cup$	-	Union.
$\neq$	-	Not equal.
ϕ	-	The empty set.
$\wedge$	-	Logical AND.
$a * b$	-	a is multiplied by b.
a <-- b	-	a is assigned the value of b.
$a \in B$	-	a is an element of the set B.
{ }	-	Set of.

CHAPTER 1

INTRODUCTION TO THESIS

1.1 INTRODUCTION AND MOTIVATION

Computer programs (systems, software) are employed in activities that range from life threatening (such as air traffic control systems) to the seemingly insignificant (such as balancing bank accounts, especially mine!). Users of these systems require assurances (from the respective system designers) that these (computer) programs are correct. That is, they contain no errors[1]. Ideally, these assurances could be provided by proving that the system is correct. However, methods for proving the correctness of such systems are limited; they either cannot be used in a cost-effective way to verify large-scale systems, or are too weakly developed technically to be effective [Good75, Mill78, Howd87].

This implies that a more practical (but weaker) method must be employed to assure users that these programs are correct (since program proving is impossible [Howd87]). Software (program) testing accomplishes this by systematically exercising program code in a controlled environment. Testing for all errors in a program requires that all elements in the program input domain be accessed or exercised. Unfortunately, input domains usually have an infinite number of elements, implying that such a testing strategy is impractical and infeasible. Thus, we need effective strategies for selecting test data that have the highest likelihood of detecting the most errors in a program. This test case construction process can be more intellectually challenging than the software design process.

[1] An error is present if either i) a program does not do what it is supposed to do; or ii) a program does what it is not supposed to do [Myer79].

One such testing strategy (white-box testing) involves selecting paths[2] through the program that satisfy specific *criteria of code coverage* (such as statement coverage[3])and then deriving test data that will cause each selected path to be exercised. However, defining a criteria of code coverage and then selecting paths through the program that satisfy specific criteria of code coverage, is usually not as difficult as deriving test data that will cause each selected path to be exercised. Some of the selected paths may be **infeasible**, i.e., there does not exist any data which will cause the program to exercise these paths.

In an attempt to avoid selecting infeasible paths while choosing paths that satisfy specific criteria of code coverage, a number of researchers have proposed a testing strategy called **adaptive (white-box) testing** [Myer86, Ince87, Prat87]. In adaptive testing, the program under test is first executed on arbitrary input. The *degree of code coverage*[4] obtained is noted, as is the execution history of the program on that input. The execution history of the program is then utilized to obtain a new path (to be exercised) and corresponding test data. Moreover, if this path is exercised, then there is an increase of the degree of (code) coverage attained. This incremental process is repeated until the required degree of code coverage is attained, or, until the degree of (code) coverage cannot be further improved due to the presence of infeasible paths.

One adaptive testing strategy for branch coverage is the **path prefix testing strategy** [Myer86, Prat87]. This strategy is the first adaptive testing strategy that achieves branch coverage (with restrictions) and offers significant advantages over non-adaptive branch coverage testing strategies. Moreover, it is the only adaptive testing strategy that satisfies a common criteria of code coverage, namely branch coverage [Myer86, Ince87, Prat87].

This suggests that adaptive testing strategies should be developed for other criteria of code coverage. One such criteria of (code) coverage is **Linear Code Sequence And**

---

[2]A path in a program is defined to be a unique sequence of possibly executable statements, i.e., any possible route from one point to another in the program [Hedl85].

[3]Each statement in a program is exercised at least once during testing.

[4]$\frac{\text{Actual number of the chosen code-based syntactic units (e.g. statements) exercised.}}{\text{Total number of such units that can be exercised.}}$

Jump (LCSAJ) coverage[5] [Wood76, Henn84]. In this thesis, we develop an adaptive test strategy that achieves LCSAJ coverage (with restrictions).

1.2 MAJOR CONTRIBUTIONS

The main contribution of this thesis is the introduction of an adaptive testing strategy that achieves LCSAJ coverage (with restrictions) --- the first adaptive testing strategy for LCSAJs. Other contributions include :

i) a proof of completeness of coverage. That is, we prove that this testing strategy achieves LCSAJ coverage (with restrictions);

ii) a technique for implementing the proposed test strategy;

and iii) a demonstration of the applicability of the proposed test strategy.

In summary, a more pragmatic approach to achieving LCSAJ coverage (than currently available) is proposed and justified. In particular, our results provide practical guidance for selecting LCSAJs to be exercised.

1.3 ORGANIZATION OF THE THESIS

A brief survey of software testing strategies is presented in Chapter 2, together with a discussion of their applicability and limitations. In Chapter 3, we formally define and illustrate an adaptive test strategy for LCSAJs. A proof of completeness of coverage is also given. In Chapter 4, implementation considerations are discussed and an technique for implementing the proposed test strategy is suggested. The applicability of the proposed test strategy is illustrated in detail with a well-known example, the Naur Line Editor problem [Good75, Prob84]. Finally, Chapter 5 contains concluding remarks and suggestions for future research.

[5]Each LCSAJs is exercised at least once during testing.

CHAPTER 2

SOFTWARE TESTING : AN OVERVIEW

Testing strategies generally fall into one of three categories, namely black-box, white-box and grey-box testing [Mill78, Myer79, Prob82b]. In this chapter, we review each of these testing strategies; in particular, we show how white-box testing strategies are hampered by the presence of infeasible paths in programs. Adaptive white-box testing strategies are then presented as a means of overcoming this difficulty. Finally, we describe the basic code-based syntactic unit (*Linear Code Sequences And Jumps*) used in our adaptive testing strategy.

2.1 OVERVIEW OF TESTING STRATEGIES

Testing is defined by [Myer79] as "the process of executing a program with the intent of finding errors". Associated with this definition is the problem of designing test data such that all program errors are discovered. Given that even for trivial programs this is, in general, an impossible and often impractical task [Good75, Myer79], researchers are constantly occupied with developing feasible, yet effective, testing strategies by which an adequate degree of confidence in a program is realized. Often, this involves considerations of cost effectiveness tradeoffs.

Existing testing strategies are usually categorized as being either black-box, white-box or grey-box based. We now present and discuss each of these testing strategies.

2.1.1 BLACK-BOX TESTING

In black-box (specification-based) testing, the tester views the program under test as a black-box. That is, test data is derived solely from program specifications (user documentation, input-output requirements, etc.) and no information from the internal structure (source code) is utilized. Ideally, **exhaustive input testing** [Good75, Mill78, Myer79] in which test data is generated for every value in the input domain, detects all errors in a program. In practice, however, this testing strategy is usually infeasible because of the possibly infinite number of elements in the input domain. Thus, black-box testing strategies are limited to selecting subsets of the input domain that have a high probability of detecting undiscovered errors in the program.

Existing black-box testing strategies partition program input and output domains into finite, equivalence (valid and invalid) classes $D = \cup D_i$ (i= 1,2,...,n), such that one representative value of any equivalence class is equivalent to all other representative values in the same class. That is, if test data (i.e. a representative value) t from equivalence class D_i detects an error, then all other test data from D_i will detect the same error. Similarly, if t does not detect an error, then all other test data from D_i (except, possibly, those belonging to other equivalence classes) would not be expected to find an error.

Design specifications, user documentation, input-output requirements, etc. are used to define equivalence classes. For example, in **equivalence partitioning** [Myer79], program input conditions are partitioned into disjoint classes $D_1, ..., D_n$ say. A test set $T = \{t_1, t_2, ..., t_n\}$, such that D_i (t_i) (i = 1, 2, ... , n) is considered adequate. **Functional testing** [Howd86, Howd87] requires the identification of, and the testing of all functions (programs are viewed as interacting collections of functions) implemented in the program. That is, each function denotes an equivalence class. **Boundary Value Analysis** [Myer79] requires that conditions denoting regions *on, above and beneath* the boundaries of input and output equivalence classes be satisfied. **Cause-effect graphing** [Myer79] explores the relationship

between combinations of input classes and output classes and derives reportedly high-yield test data. **Error guessing** [Myer79] relies on the intuition and experience of a tester that certain types of errors may be present in a program, i.e., each error type denotes an equivalence class. Other black-box strategies are found in [Mill78, Dura80, Howd86, Howd87]. In practice, more than one black-box testing strategy may be used in order to improve testing thoroughness.

Problems usually associated with black-box testing strategies are :

i) Program specifications are seldom detailed to the extent that an equivalence partition is easily derived. Moreover, deriving test data to satisfy partition conditions can be difficult;

ii) A partition may be an inadequate representation of a program specification;

iii) Some items or features of a program specification may have been omitted;

iv) Program specifications may be ambiguous;

v) Information about the source code (usually containing errors) is ignored in generating test data.

A major goal of black-box testing is to detect errors in program specifications. Even if one exhaustively tests program specifications (if this is possible), program errors may still be present in the source code. In order that the actual (source) code be incorporated into the testing process, black-box testing is usually supplemented with a complementary test strategy --- **white-box testing.**

2.1.2 WHITE-BOX TESTING

In white-box (code-based, structural) testing, test data are derived solely from the source code and no use is made of program specifications (e.g. design requirements, etc.). Two distinct phases are usually associated with white-box testing, namely *test path selection* and *test data derivation* [Rapp85, Prat87].

Test path selection usually requires the use of either the program itself or the corresponding program (control or data) flowgraph, from which a set of paths satisfying specific criteria of code coverage are selected. Then in the **test data derivation** phase, test data is derived from the program input domain, that will cause each selected path to be exercised. In general, this latter activity is extremely difficult --- so difficult that, in fact, deriving test data to cause even a single statement (in a program) to be exercised is, in general, an unsolvable problem [Mins67].

The strongest structural testing strategy is **exhaustive path testing** [Myer79, Rapp85] in which all paths in a program are exercised. Programs containing loops usually have an infinite number of paths, implying that this testing strategy is impractical and infeasible. A program may still have a large number of paths, even if paths exercising the same loop a different number of times are considered equivalent [Howd80]. Thus, white-box testing strategies are usually restricted to choosing subsets of all paths (in a program) that have a high probability of detecting undiscovered errors associated with exercising these paths.

Most test path selection techniques are based on **control flow analysis**, which examines the branch and loop structure of a program. White-box testing strategies with test path selection phases based on control flow analysis usually detect errors related to the internal structure (source code) of the program under test. Common criteria of code coverage that are associated with control flow analysis are :

i) *Statement coverage* - each statement in a program is exercised at least once;

ii) *Branch coverage* - for each predicate[6] in the program, exercise each decision outcomes at least once;

iii) *Multiple Condition Coverage* - for each predicate in the program, exercise all possible combinations of simple boolean conditions at least once.

[6]A predicate is a conditional transfer statement. For example, the statement {IF (x> 1) THEN go to 5} is a predicate. Moreover, it is an IF predicate.

Other criteria of code coverage associated with control flow analysis are described in [Good75, Howd75, Huan75, Huan78, Mill78, Paig77, Myer79, Fost80, Henn84].

Other test path selection techniques are based on **data flow analysis**, which (by observing the flow of data in either the program itself or the corresponding flowgraph) focuses on the associations between assignments of values to variables (i.e., their definitions), and the effects of these variables on other variables (i.e., their uses) during execution of the program. White-box testing strategies with test path selection phases based on data flow analysis usually detect errors related to computational and data dependencies between these variables [Rapp85]. Common criteria of code coverage associated with data flow analysis are Rapps and Weyuker's **all du-paths** [Weyu84, Rapp85], Ntafos's **required k-tuples** [Ntaf84], Laski and Korel's (**ordered**) **context coverage** [Lask83] and Ural and Yang's **all simple OI-paths** [Ural88]. Other criteria of code coverage associated with data flow analysis are discussed in [Huan78, Clar85, Kore87, Yang88].

Not all white-box testing strategies have a test path selection phase. For example, in **mutation testing** [DeMi78], a set of mutant programs $\mathbf{P_1}$, $\mathbf{P_2}$, ... , $\mathbf{P_k}$ of **P** (the program under test) are created such that they differ from **P** only in the occurrence of simple coding errors[7]. The mutant programs are then executed with test data assumed adequate for **P**, and those producing the same output as **P** are analyzed and used in deriving additional test data that distinguishes **P** from these mutant programs.

SETAR [Kund79], the path prefix testing strategy [Myer86, Prat87], domain testing [Whit80] and our adaptive testing strategy (presented in Chapts. 3 and 4) are examples of white-box testing strategies that have no *a priori* test path selection phase.

Problems experienced with white-box testing strategies are :

i) Infeasible paths, i.e., paths through a program that can never be exercised, may be selected in an attempt to satisfy specific criteria of code coverage;

[7]For example, if **P** contains $B \leq C$ then one mutant program will have $B = C$.

ii) A path **p**, can compute correct values for some, but not all input for that path [Howd80] and no distinction is made (by most white-box testing strategies) between different input that cause **p** to be exercised. Thus, test data selected can result in **p** generating correct output (and not incorrect output as we would prefer); and

iii) No use is made of possibly incorrect (i.e., ambiguous, etc.) program specifications in deriving test data.

As implied above, even if a program has been thoroughly tested by white-box testing strategies, the program can still contain errors because of ambiguities in the specification. Ambiguities are often detected with black-box testing strategies. Thus, white-box and black-box testing strategies are complementary. Errors relating to program design requirements, however, may be present in programs tested by both strategies, and in an attempt to detect these errors, another testing strategy can be employed --- **grey-box testing.**

2.1.3 GREY-BOX TESTING

In this testing strategy, program design requirements and possibly, functional specifications and/or source code, are used to derive test data [Prob82b]. For example, in **semantic instrumentation** [Prob82b], the source code of the program under test is instrumented with probes [Huan78, Prob81, Prob82a], to measure the coverage of equivalence classes of program design behaviour, the latter being determined solely from (program) design requirements. That is, this testing strategy tests design requirements by causing the corresponding source code to be exercised. Information derived from this (testing) process is then used to derive additional test data and also modify design requirements.

Another grey-box testing strategy uses information derived **solely** from program design requirements [Prob82b]. Design requirements can be represented by formal models such as **Finite State Automata** [Mins67, Hopc79] and **Augmented Transition Networks** (ATNs) [Prob82b]. Methods for validating, analyzing, etc. these models have been used in

verifying corresponding design requirements. For example, ATNs representing design requirements have been analyzed for consistency and completeness. In addition, test data can be derived from the ATNs.

Protocol testing strategies [Chow78, Sari84] are usually grey-box based. Other examples (of grey-box testing strategies) are found in [Prob82b, Ince87]. Moreover, grey-box testing strategies are usually complementary testing strategies that combine and/or modify existing testing strategies, and should be used in conjunction with white-box and black-box testing strategies in order to improve testing thoroughness.

In the next section, we address one problem (initially discussed in Sec. 2.1.2) associated with white-box testing strategies --- selecting infeasible paths in programs.

2.2 A FUNDAMENTAL PROBLEM WITH WHITE-BOX TESTING STRATEGIES

White-box testing strategies usually use either the program itself or the corresponding flowgraph to select a set of paths $\mathbf{P} = \{\mathbf{p_i}\}$ satisfying specific criteria of code coverage (See Sec. 2.1.2) . Associated with each path $\mathbf{p_i}$ is a **conjunctive path predicate** [Kund79, Myer86, Prat87]

$$\mathbf{P_i} = C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

having conjuncts C_i (i = 1, 2, ..., n) corresponding to the decision outcomes encountered on path $\mathbf{p_i}$. Each conjunct C_i is an interpreted predicate[8] obtained via symbolic execution techniques [Huan75, Clar76, King76]. Test data $\mathbf{x_i}$ is then derived for $\mathbf{p_i}$ by utilizing all of the equations and/or inequalities derived from conjuncts C_i.

[8]An interpreted predicate is an expression (in terms of input variables) that must be satisfied in order to determine the decision outcome at the corresponding predicate.

```
Line

1       Input x
2       If  x > 0 then
3         x <-- x -1
4         If x > 0  then
5           print " x -1 > 0 "
6         Else
7           print " x - 1 ≤ 0"
8         Endif
9       Else
10        print " x > 0"
11      Endif
12      stop
```

Fig. 2.1 Illustrating conjunctive path predicates.

For example, consider the program in Fig. 2.1. If the statements corresponding to lines 1, 2, 3, 4, 5 and 12 are exercised, then the path corresponding to the execution of these statements has conjunctive path predicate **P**, where

$$\mathbf{P} = (x > 0) \wedge (x\text{-}1 > 0).$$

Conjuncts C_1 and C_2 are (x > 0) and (x-1 > 0) respectively. Any x > 1 satisfies the conjunctive path predicate **P**.

Huang [Huan75] suggests an approach to solving the equations and/or inequalities derived from conjuncts C_i's Clark [Clar76] and others [King76, Howd77], have developed symbolic execution systems that automate the process of finding solutions to these equations and/or inequalities. All of the above are hampered by one fundamental problem --- finding input to cause a path (in this case, paths associated with corresponding conjunctive path predicates) to be exercised is, in general, an unsolvable problem [Mins67].

Herein lies the most serious problem with code-based testing strategies --- all paths to be exercised are selected without regards to infeasibility. Whenever infeasible paths are detected, new paths need to be selected (all over again) because the chosen (code-based) syntactic units (e.g. statements) associated with a specific criteria of code coverage may be exercised by another path not previously selected. The process is repeated until either all occurrences of the

chosen syntactic unit are exercised (i.e., the required degree of code coverage is attained) or until it can be shown that specific (syntactic) units can never be exercised. This (i.e., selecting new paths) implies significant costs in computer resources. In addition, the computational effort spent in analyzing an infeasible path is seldom used in the remainder of the testing process (an exception is "allegations" [Wood80]). In an attempt to overcome these difficulties, a new testing strategy has evolved --- **adaptive testing.**

2.3 ADAPTIVE WHITE-BOX TESTING STRATEGIES

Adaptive white-box testing strategies are based on two principles [Kund79, Myer86, Ince87, Prat87], namely :

i) Use of previously exercised paths (and corresponding inputs), to guide the selection of subsequent paths; and

ii) Incremental addition of one path to existing (exercised) paths to increase the degree of code coverage attained. (e.g. increase the number of statements exercised).

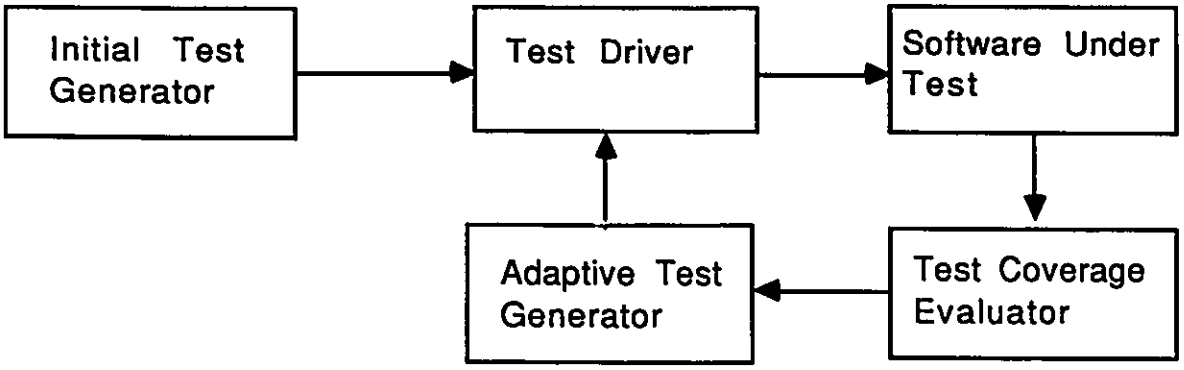


Fig. 2.2 The components of an adaptive white-box testing environment.

The components of an adaptive white-box testing environment [Ince87] are shown in Fig. 2.2. An *initial test generator* provides the initial test data. The *test driver* controls the application of test data to the *software under test.* The *test coverage evaluator* updates coverage statistics (i.e., indicates the degree of code coverage attained), and the *adaptive*

test generator uses previously exercised paths to select new paths to be exercised. Moreover, these paths (if exercised) will increase the degree of code coverage attained. The strategy used by the *adaptive test generator* serves to distinguish one adaptive white-box testing strategy from another.

We elaborate on two such strategies : **SETAR** [Kund79] and the **path prefix testing strategy** [Myer86, Prat87]. Other adaptive white-box testing strategies are discussed in [Ince87].

2.3.1 THE METHOD OF KUNDU - SETAR

Kundu [Kund79] describes a testing strategy, called **SETAR** (Simple and Effective Testing through Avoidance of Regions) that explores the relationship between paths $p_1, p_2, ..., p_{k-1}$ and their corresponding conjunctive path predicates $P_1, P_2, ..., P_{k-1}$ (See Sec. 2.2). Path p_k, the next path to be exercised, is obtained by determining test data x_k such that at least one of the conjuncts in each of the conjunctive path predicates P_i $(1 \leq i \leq k-1)$ is violated. That is, if $P_i = C_1 \wedge C_2 \wedge ... \wedge C_n$ (See Sec. 2.2), then test data x_k is chosen such that at least one of the conjuncts C_j $(1 \leq j \leq n)$ in each of the conjunctive path predicates P_i, is not satisfied. This forces path p_k to be distinct from all other paths $p_1, p_2, ..., p_{k-1}$.

In contrast with most white-box testing strategies, SETAR has no *a priori* test path selection phase (See Sec. 2.1.2). In particular, path p_n is determined from test data t_n, and not vice-versa (as in most white-box testing strategies). Moreover, SETAR has not been designed with any specific criteria of code coverage in mind even though Kundu states that any such criteria can be utilized.

Kundu claims that SETAR has two main advantages over white-box testing strategies that have *a priori* test path selection phases, namely :

i) There is no *a priori* test path selection phase. As a result, infeasible paths are not selected. (It may be difficult to find input that violates at least one conjunct in each conjunctive path predicate); and

ii) Finding test data that violates at least one of the conjuncts in each conjunctive path predicate P_i is usually simpler than finding a solution to any single conjunctive path predicate P_i[9]. As such, test data is usually easily generated.

The major disadvantage of the SETAR testing strategy is that it has no termination criterion. Test data x_k (the next input selected), is chosen such that it forces path p_k to be distinct from all previously exercised paths, implying that it is up to the tester to determine when a sufficient degree of code coverage has been achieved.

2.3.2 THE PATH-PREFIX TESTING STRATEGY

In this branch coverage testing strategy, maximum use is made of all previously exercised paths [Myer86, Prat87]. Associated with every exercised path **p** is its **reversible prefix q** --- the minimal initial portion of **p** to a predicate (inclusive) whose decision outcomes have not yet been fully exercised (See Fig. 2.3).

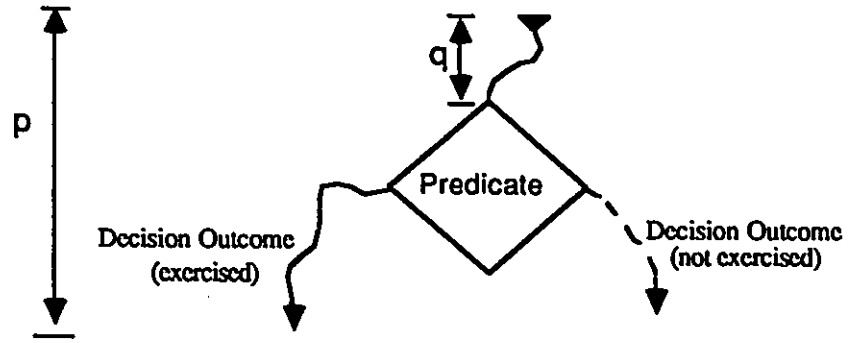


Fig. 2.3 Reversible prefix **q** of path **p**.

Given that paths $p_1, p_2, ..., p_{k-1}$ have already been exercised, new test data is derived that will cause reversal of the **shortest** reversible prefix **q** among all the p_i 's, to be exercised. We illustrate with an example.

---

[9]White-box testing strategies with *a priori* test path selection phases require that solutions to each conjunctive path predicate be obtained.

EXAMPLE 2.1

Consider the program in Fig. 2.4 in which predicates are labelled numerically by parenthesized, super-scripts. A branch coverage matrix, listing all predicates and corresponding decision outcomes (i.e.,*True* (T) or *False* (F) values), is shown in Table 2.1 (for now, ignore the numbers under T and F in Table 2.1).

```
Line

1     input x, y, z
2     while ( x > 0 ) and ( y > 0 ) (1) do
3      begin
4           if x > 50 (2) then
5            begin
6                 while y > 50 (3) do
7                   begin
8                     if even(z) (4) then
9                         print  "x > 50, y > 50, z - even"
10                    else
11                        print  "x > 50, y > 50 , z - odd";
12                    y <-- y-25;
13                  end;
14                  x <-- x - 25;
15           end
16          else
17           begin
18                if x > 25 (5) then
19                  begin
20                    if y > 25 (6) then
21                        print  "x > 25, y > 25"
22                    else
23                        print "x > 25, y ≤ 25";
24                    y <-- y - 15;
25                  end
26                else
27                  print  "x ≤ 25";
28                x <-- x - 15;
29           end;
30     end;
31    if ( x = 0 ) and ( y = 0) (7) then
32          print  "x=y=0"
33    else
34          print  "x <> 0  or y <> 0";
35    stop;
```

Fig. 2.4 Example 2.1.

Predicate	Decision Outcome T	F
1	1	1
2	2	1
3	3	2
4	4	3
5	2	1
6	3	2
7	5	1

Table 2.1 Branch coverage matrix for the program in Fig. 2.4.

The path prefix testing strategy achieves branch coverage as follows. First, the program is executed on arbitrary input, say x = 10, y = 20 and z = 5, that causes path P_1 corresponding to 1T 2F 5F 1F 7F[10] to be exercised[11]. (A "1" is recorded in either the second or third column of Table 2.1 for each previously unexercised decision outcome associated with a predicate). Next, since 1T 2F is the only reversible prefix, we try to derive data that will cause 1T 2T to be generated (i.e., reverse 2F in reversible prefix 1T 2F). Having x = 51, y = 20, z = 5 causes path P_2 = 1T 2T 3F 1T 2F 5T 6F 1T 2F 5F 1F 7F to be exercised ("2" in Table 2.1).

Now, there are two reversible prefixes, 1T 2T 3F and 1T 2F 5F 1F 7F in P_2 and P_1 respectively. We choose the former since it is shorter. Moreover, having x = 51, y = 51 and z = 5, causes path P_3 = 1T 2T 3T 4F 3F 1T 2F 5T 6T 1T 2F 5F 1F 7F to be exercised ("3" in Table 2.1).

At this stage, there are three reversible prefixes, namely :

i) 1T 2F 5F 1F 7F in **P1**;

ii) 1T 2T 3F 1T 2F 5T 6F 1T 2F 5F 1F 7F in **P2**; and

iii) 1T 2T 3T 4F in **P3**.

[10] 1T 2F ... means that predicate 1 evaluated to *True* (T), predicate 2 evaluated to *False* (F), etc.. Moreover, the predicates were evaluated in the above order.

[11] Hereafter "corresponding to" and "=" are used interchangeably. Thus path P_1 = 1T 2F 5F 1F 7F.

Thus, we should generate reversible prefix iii) above. Having x = 51, y = 51 and z = 6 reverses 4F in **P3** and causes path $\mathbf{P_4}$ = 1T 2T 3T 4T 3F 1T 2F 5T 6T 1T 2F 5F 1F 7F to be exercised ("4" in Table 2.1).

Now, there are four reversible prefixes , namely :

iv) 1T 2F 5F 1F 7F in **P1**;

v) 1T 2T 3F 1T 2F 5T 6F 1T 2F 5F 1F 7F in **P2**;

vi) 1T 2T 3T 4T 3F 1T 2F 5T 6T 1T 2F 5F 1F 7F in **P3**; and

vii) 1T 2T 3T 4T 3F 1T 2F 5T 6T 1T 2F 5F 1F 7F in **P4**.

Thus, we should generate reversible prefix iv) above. Having x = 15, y = 0 and z = 5 causes path $\mathbf{P_5}$ = 1T 2F 5F 1F 7T to be exercised ("5" in Table 2.1).

Each decision outcome associated with a predicate, has been exercised. That is, branch coverage has been achieved.

Myers [Myer86] proved that this testing strategy always achieves branch coverage, and can be extended to achieve multiple condition coverage. Moreover, Myers claims that this testing strategy has several advantages over non-adaptive branch coverage testing strategies, namely :

i) The probability is high that the subpath associated with a reversible prefix will be feasible, since a corresponding path has already been exercised. A strength of this result is that the path prefix testing strategy avoids (to some degree) selecting infeasible paths;

ii) Fewer decision outcomes are (usually) evaluated in subpaths associated with reversible prefixes than with paths selected by an *a priori* test path selection phase (See Sec. 2.1.2). Thus, there are fewer computational requirements; and

iii) The shortest reversible prefix is chosen (for reversal), further simplifying computation requirements.

The path prefix testing strategy achieves branch coverage modulo feasibility[12]. It is possible, given the reversible prefixes currently available for reversal, certain decision outcomes will not be exercised, even though they could have been exercised, because the subpath associated with the corresponding reversible prefix is infeasible. This is known as **relative infeasibility** [Myer86, Prat87] and is a perceived weakness of the path prefix testing strategy. Moreover, it implies that another testing strategy must be used to derive test data that will cause these decision outcomes to be exercised.

We extend this adaptive white-box testing strategy (in Chapts. 3 and 4) to achieve **Linear Code Sequence And Jump** (LCSAJ) coverage modulo feasibility[13]. In the next section, LCSAJ coverage testing strategies are discussed.

2.4 LCSAJ COVERAGE TESTING STRATEGIES

In this section, we discuss several LCSAJ coverage testing strategies. Before we begin, however, we must define an **LCSAJ** (Linear Code Sequence And Jump) [Wood76, Wood80, Henn84, Hedl85] Moreover, before we can define an **LCSAJ**, we need to introduce the following terms :

A **target line** is defined to be the entry point to the operating system[14] or any line to which execution control flow can be given (other than from the previous line).

A **start line** is defined to be the first line of the program text or a *target line* (other than the entry point to the operating system).

An **end line** is defined to be any line from which execution control flow can be given to a *target line*. (Clearly, the last statement in a program that can be exercised is an end line).

---

[12]Branch coverage modulo feasibility means that the strategy does not claim to cause unrealizable decision outcomes to be exercised, since no input exists that will cause such decision outcomes to be exercised. For example, the IF predicate {IF (A $\wedge$ A) then go to 5 } can never have a *False* outcome. Thus, this decision outcome can never be exercised.

[13]LCSAJ coverage modulo feasibility means that the strategy does not claim to cause unrealizable LCSAJs to be exercised, since no input exists that will cause such LCSAJs to be exercised.

[14]We assume that execution control flow is always transferred from the last executed statement in a program to the operating system (See Sec. 3.1).

Then, we have the following definition :

An **LCSAJ** (Linear Code Sequence And Jump) is any *linear sequence of (possibly executable) code* that commences at a start line ***s*** and terminates at an end line ***e*** such that ***e*** can be reached from ***s*** by an *unbroken linear sequence of (possibly executable) code,* and transfers execution control flow from ***e*** to the target line ***t***.

Clearly, an LCSAJ is characterized by a start line ***s***, an end line ***e*** and a target line ***t*** (associated with end line ***e***). (In fact, we use ***s***, ***e*** and ***t*** to describe LCSAJs) In addition, an LCSAJ may contain predicates which must be satisfied in order to exercise the corresponding linear code sequence (and terminating jump).

We now illustrate the above concepts with an example. The program in Fig 2.5 (taken from [Rapp85]), contains 11 LCSAJs, listed in Table 2.2. If the program in Fig. 2.5 is executed on data x = 1 and y = 1, then LCSAJs 2, 6, 7, 9 and 11 (from Table 2.2) are exercised.

Line	
1	Start
2	Read x, y
3	If y < 0 then goto 6
4	pow <-- y
5	goto 7
6	pow <-- -y
7	z <-- 1
8	If pow = 0 then goto 12
9	z <-- z * x
10	pow <-- pow - 1
11	goto 8
12	If y ≥ 0 then goto 14
13	z <-- 1/z
14	answer <-- z + 1
15	print answer
16	stop

Fig. 2.5 Example program with 11 LCSAJs.

LCSAJ	Start Line	End Line	Target Line
1	1	3	6
2	1	5	7
3	6	8	12
4	6	11	8
5	7	8	12
6	7	11	8
7	8	8	12
8	8	11	8
9	12	12	14
10	12	16	exit[15]
11	14	16	exit

Table 2.2 LCSAJs for the program in Fig. 2.5.

Note that several LCSAJs may have either the same start line, end line or target line. For example, from Table 2.2, LCSAJs 1 and 2 each have line 1 as a start line; LCSAJs 6 and 8 each have line 11 as an end line and line 8 as a target line; and LCSAJs 3 and 5 each have line 12 as a target line and line 8 as an end line. In addition, note that a sequence of LCSAJs form a path if the target line of one LCSAJ is the same as the start line of its successor. For example, in Table 2.2, a path can be formed from LCSAJs 2, 6, 7, 9 and 11.

Clearly, programs with **IF**, **WHILE** and **REPEAT** structured control constructs only (See Sec. 3.2) have the following properties :

PROPERTY I : The end line of any LCSAJ is either at :

i) The last statement (in the program) that can be exercised;

ii) An IF predicate;

iii) A WHILE predicate;

iv) An UNTIL predicate;

v) An ENDWHILE statement; or

vi) The statement *immediately preceding* an ELSE statement.

[15]"Exit" denotes the entry point to the operating system (see Sec. 3.1).

PROPERTY II : The start line of any LCSAJ is either at :

i) The first statement of the program;

ii) An ELSE statement;

iii) A REPEAT statement;

iv) A WHILE predicate;

v) The statement *immediately following* an ENDWHILE statement; or

vi) The statement *immediately following* an ENDIF statement.

Now, branch coverage is widely regarded as the minimal standard of achievement in white-box testing. Moreover, it is generally accepted that branch coverage testing strategies cannot detect as many errors (in a program) as other white-box testing strategies that cause combinations of decision outcomes to be exercised, rather than single decision outcomes [Wood80, Rapp85]. An LCSAJ coverage testing strategy is one such strategy.

In one LCSAJ coverage testing strategy, Woodward [Wood84] attempts to determine the amount of work needed to achieve LCSAJ coverage, given that branch coverage has already been achieved, by selecting a **minimal** set of paths that will achieve LCSAJ coverage. This experiment determined that, on average, less than six additional paths were needed to accomplish this task. However, this result is purely theoretical --- no attempt was made to derive test data that will cause each selected path to be exercised.

Another LCSAJ coverage testing strategy [Henn84] required that test data, derived from functional testing [Howd86], be applied to the program under test. Unexercised LCSAJs are reported and, in addition, if statement and branch coverage have not yet been achieved, test data are derived in order that they now be achieved. Finally, an attempt is made to exercise the remaining (unexercised) LCSAJs. However, no method is suggested by [Henn84] on how one should accomplish this task.

Woodward [Wood76], in another LCSAJ coverage testing strategy, suggested that after branch coverage has been achieved, that an attempt then be made to exercise the

shortest path containing unexercised LCSAJs. His experimental study determined that a high proportion of these paths were, in fact, infeasible.

In Chapters 3 and 4, we present an alternative LCSAJ coverage testing strategy --- one that improves upon the above approaches to attaining LCSAJ coverage.

2.5 SUMMARY

In this chapter, we discussed several test strategies; in particular we examined the fundamental problem (selecting infeasible paths) associated with white-box testing strategies and showed how, to some extent, adaptive testing strategies can overcome this difficulty. Recognizing these advantages, we develop, in the next chapter, an adaptive testing strategy that achieves **Linear Code Sequence and Jump** (LCSAJ) coverage modulo feasibility.

CHAPTER 3

AN ADAPTIVE LCSAJ TEST STRATEGY AND PROOF OF COMPLETENESS

3.1 INTRODUCTION

The readability of a computer program is enhanced if the execution control flow sequence follows the sequence in which the source code is written. This (readability) in turn eases the understanding, debugging, testing and modification of the computer program [Fair85]. One method of attaining such a standard of readability is via *structured control constructs*, namely **IF**, **WHILE** and **REPEAT** constructs [Prob82a, Fair85]. In this chapter, we first show how LCSAJs manifest themselves in programs with these structured control constructs. Then we present an adaptive test strategy for LCSAJs and prove that this strategy achieves LCSAJ coverage, modulo feasibility.

Before proceeding, however, we state standard assumptions (utilized in this and subsequent chapters) based on the structure and behaviour of the program under test. These assumptions are satisfied by most programs; thus, no significant loss of generality is involved.

i) All observed program executions (traces) are finite. (This is self-evident);

ii) All programs (modules, etc.) have one entry and one exit point. This assumption can be relaxed (See Chapt. 5);

iii) All programs are GOTO less --- this assumption can be relaxed (See Chap. 5);

iv) Only **IF**, **WHILE** and **REPEAT** (structured) control constructs are allowed. This assumption can also be relaxed (See Chap. 5);

v) All programs have at least one structured control construct;

vi) Execution control flow is transferred from the last executed statement in a program to the operating system. This is denoted by "exit" [Henn84];

vii) Programs are "well-delimited" [Prob82a]. That is, program delimiters such as **ENDWHILE**, **ENDIF**, etc. are explicitly present in the source code. Moreover, we assume that these statements are executable;

viii) **1.** When an **IF** predicate (in an **IF** construct with an **ELSE** part) evaluates to *False*, control (i.e., execution control flow) is transferred to the **ELSE** statement. When the predicate evaluates to *True*, the statement *immediately preceding* the **ELSE** statement is the last to be exercised (in the **IF** construct) before control is transferred to the statement *immediately following* the **ENDIF** statement;

2. When the **IF** predicate (in an **IF** construct with no **ELSE** part) evaluates to *False*, control is transferred to the statement *immediately following* the **ENDIF** statement;

3. When an **UNTIL** predicate evaluates to *False*, control is transferred to the **REPEAT** statement itself;

4. When a **WHILE** predicate evaluates to *False*, control is transferred to the statement *immediately following* the **ENDWHILE** statement;

5. The **ENDWHILE** statement is the last to be exercised in a **WHILE** construct, before control is transferred to the corresponding WHILE predicate.

Note that assumption viii) is of critical importance, and is necessary for a complete understanding of the *Adaptive LCSAJ Test Strategy*.

We also use the following conventions :

i) Sn, where **n** is an integer, denotes a single executable statement;

ii) (n), where **n** is an integer, denotes a boolean condition. Moreover, **n** can be used to represent predicates in a program (see Sec. 3.2).

3.2 PROPERTIES OF LCSAJs IN STRUCTURED CONTROL CONSTRUCTS

Program control constructs [Prob82a, Fair85], are generally composed of combinations of three structured control constructs, namely **IF**, **WHILE** and **REPEAT** constructs[16]. In this section, we show how LCSAJs manifest themselves in programs containing these constructs. Moreover, properties related to LCSAJ coverage (in these programs) are proved.

First, however, we introduce terms used in this and subsequent sections :

A **decision element** or **element** is a pair (**nD**), where **n** is an integer that represents a predicate in a program, and **D** $\in$ {**T**, **F**} represents the corresponding decision outcome at evaluation (i.e., a *True* (**T**) or *False* (**F**) value). Every predicate in the program is associated with a distinct integer.

A **T-element** is a decision element with a *True* (**T**) value and an **F-element** is a decision element with a *False* (**F**) value.

Whenever a predicate is evaluated, the corresponding decision element is recorded in a *trace output file.*

A **program execution trace** or **trace** is the *actual sequence* of decision elements obtained from the trace output file (by executing the program on data). It represents the execution history of the program on that data. For example, 1T 2T 3F could be a program execution trace

[16] In fact, most programs can be written using only **IF**, **WHILE** and **REPEAT** structured control constructs [Fair85].

in which predicates associated with integers "1", "2" and "3" evaluated to *True*, *True* and *False* respectively. Moreover, the predicates associated with integers "1", "2" and "3" were exercised (and evaluated) in the above specified order.

3.2.1 LCSAJs in IF CONSTRUCTS

When an IF predicate evaluates to *False*, the predicate is the last to be exercised (in the IF construct) before control is transferred to either :

i) the corresponding ELSE statement (in an IF construct with an ELSE part); or

ii) the statement *immediately following* the corresponding ENDIF statement (in an IF construct with no ELSE part).

In either case, the predicate is the end line of an LCSAJ. Moreover, each of the statements in i) and ii) above, is the start line of another LCSAJ.

When an IF predicate evaluates to *True* (in an IF construct with an ELSE part), the statement *immediately preceding* the ELSE statement is the last to be exercised (in the IF construct) before control is transferred to the statement *immediately following* the corresponding ENDIF statement. In this case, the *former* is the end line of an LCSAJ and the *latter* is the start line of another LCSAJ

However, when the IF predicate (in an IF construct with no ELSE part) evaluates to *True*, the corresponding ENDIF statement is the last to be exercised (in the IF construct) before control is transferred to the statement *immediately following* the ENDIF statement. Note that in this case, neither the ENDIF statement nor the statement *immediately following* the ENDIF statement , are start lines or end lines of LCSAJs (See Sec. 2.4).

Line					
1	S1	**LCSAJ**	**Start line**	**End line**	**Target line**
2	IF (1) THEN				
3	S2				
4	ELSE	1	1	2	4
5	S3	2	1	3	7
6	ENDIF	3	4	7	exit
7	S4	4	7	7	exit
	(a)		(b)		

Fig. 3.1 LCSAJs in an example IF construct with an ELSE part.

We illustrate the above concepts with two examples. First, consider the program in Fig. 3.1(a). Its LCSAJs are listed in Fig. 3.1(b). If 1F is the program execution trace then LCSAJs 1 and 3 are exercised. Similarly, if 1T is another trace then LCSAJs 2 and 4 are exercised. Thus, two distinct program execution traces suffice to cause all LCSAJs in a single IF construct (with an ELSE part) to be exercised.

Next, consider the program in Fig. 3.2(a). Its LCSAJs are listed in Fig. 3.2(b). If 1F is the program execution trace, then LCSAJs 1 and 3 are exercised, and if 1T is another trace then LCSAJ 2 is exercised.

Line					
1	S1	**LCSAJ**	**Start line**	**End line**	**Target line**
2	IF (1) THEN				
3	S2				
4	ENDIF	1	1	2	5
5	S3	2	1	5	exit
		3	5	5	exit
	(a)			(b)	

Fig. 3.2 LCSAJs in an example IF construct with no ELSE part.

It must be stressed that if LCSAJ 2 (in Fig. 3.2(b)) is exercised, then it does not imply that LCSAJ 3 will be exercised, even though they each have the same end line and target line. This is because, from the definition of an LCSAJ (Sec. 2.4), LCSAJ 3 is exercised **if and**

only if control was transferred from the IF predicate (in Fig. 3.2(a)) to line 5, whereas LCSAJ 2 is exercised **if and only if** the IF predicate evaluated to *True*. Clearly if 1T is the program execution trace (for the program in Fig. 3.2(a)) then LCSAJ 2 (and not LCSAJ 3) is exercised.

Based on the above discussion, we have the following results :

Lemma 3.1

Let S be a program with IF constructs only. Then whenever S is executed on a single set of inputs, any LCSAJ in S is exercised at most once.

Proof

No loops are in S, implying that any linear code sequence (and, by extension, any LCSAJ) is exercised at most once. The result then follows. ∎

Lemma 3.2.1

Let S be a program with IF structures only. Then each F-element in a program execution trace from S implies that two LCSAJs are exercised (in S).

Proof

By consideration of cases considered in the above discussion. ∎

Theorem 3.2

Let S be a program with n (n > 0) IF structures and let **T1** be a program execution trace (from S) with m F-elements, where ($1 \leq m \leq n$). Then at least m + 1 LCSAJs are exercised (in S).

Proof

Without loss of generality, assume that the program execution trace starts and terminates with an F-element. Suppose that **T1** is one such trace, where

T1 = 1F X_1 2F X_2 3F ... (m-1)F X_{m-1} mF

and each X_i (i = 1,2,..., m-1) contains zero or more T-elements.

From Lemma 3.2.1, each F-element in **T1** implies that two LCSAJs are exercised. Thus, one LCSAJ (associated with F-element 1F) is that with start line at the first line of the program text, and end line at the predicate associated with F-element 1F. The other LCSAJ (associated with decision element 1F) has its start line at either :

i) the corresponding ELSE statement (if an ELSE part exists); or

ii) the statement *immediately following* the corresponding ENDIF statement (if no ELSE part exists).

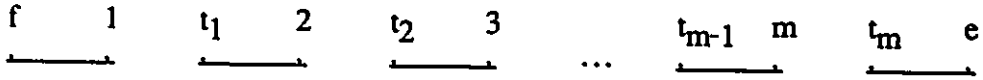
For now, let us define this start line (i.e., i) or ii) above) to be the target line "indicated by" (defined later in this Chapter) F-element 1F. Assume that the LCSAJ with this start line is as long as possible, i.e, it has its end line at the IF predicate associated with F-element 2F.

Now, suppose that similar assumptions are made for IF constructs associated with F-elements 2F and 3F, F-elements 3F and 4F, etc.. That is, we assume that :

i) each (exercised) LCSAJ with start line "indicated by" F-element kF ($1 \leq k < m$), has its end line at the IF predicate associated with F-element (k + 1) F; and

ii) each (exercised) LCSAJ with start line "indicated by" F-element mF, has its end line at the last line of the program text.

Then, in this case, m + 1 LCSAJs are exercised. This is the worst case, i.e., the case with fewest LCSAJs being exercised (See Fig 3.3).



f, e	-	First and last lines respectively of the program text.
k ($1 \leq k \leq m$)	-	IF predicate associated with F-element kF.
t_k	-	Target line "indicated by" F-element kF.
a b	-	LCSAJ with start line **a**, end line **b**, target line t_b. (t_e denotes the entry point to the operating system).

Fig. 3.3 Worst case : fewest LCSAJs exercised.

Clearly, more than m + 1 LCSAJs are exercised if any of the IF constructs associated with a T-element in X_i has an ELSE part. The result then follows. ∎

The following corollary is immediate.

Corollary 3.2.1

Let S be a program with n (n > 0) IF constructs only and let T = 1F 2F ... (n-1)F nF be a program execution trace from S. Then n + 1 LCSAJs are exercised (in S).

Proof

Let X_i (i = 1,2,...,n-1) = ϕ and let m = n in the proof of Theorem 3.2. Then the result follows by Theorem 3.2. ∎

Theorem 3.3

Let S be a program with n (n > 0) IF constructs only, all of which have ELSE parts. Then any program execution trace from S with n decision elements, implies that n + 1 LCSAJs are exercised in S (regardless of the number of F-elements in the trace).

Proof

The result is proved by induction on the number of IF constructs, all of which have an else part, in S. Suppose that only one IF construct with an ELSE part is in S. Clearly, any program execution trace from S contains one decision element. Moreover, only two LCSAJs are exercised (in S). This was earlier illustrated in Fig. 3.1 (in this Section) --- if 1T is a program execution trace (for the program in Fig 3.1(a)) then LCSAJs 2 and 4 are exercised, and if 1F is another trace then LCSAJs 1 and 3 are exercised.

Suppose that the result holds for S with k ($k \geq 1$) IF constructs, all of which have ELSE parts. Then by assumption, any trace from S implies that k+1 LCSAJs are exercised.

Without loss of generality, let **p** be any statement in the k+1 st LCSAJ. Also, let **s** be the start line and **e** the end line of this LCSAJ (See Fig. 3.4(a)).

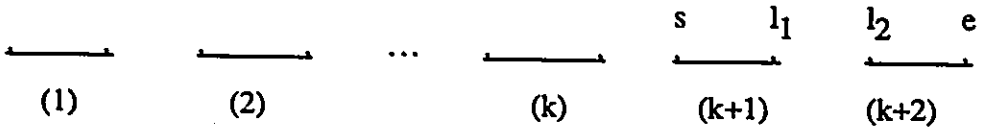
s, e - Start line and end line respectively of LCSAJ k+1.
p - Executable statement.
(n) - LCSAJ n. $(1 \leq n \leq k+1)$

Fig. 3.4(a) k+1 LCSAJs exercised.

Suppose that an IF construct with an ELSE part is inserted in the program code at **p**. Then the LCSAJ with start line **s** and end line **e** no longer exists. Instead, two new LCSAJs are exercised[17] (See Fig. 3.4(b)), namely :

i) one LCSAJ with start line **s**, and end line (labelled l_1 in Fig. 3.4(b)) at either the IF predicate or the statement *immediately preceding* the ELSE statement, of the IF construct at **p**; and

ii) another LCSAJ with end line **e**, and start line (labelled l_2 in Fig. 3.4(b)) at either the ELSE statement or the statement *immediately following* the ENDIF statement, of the IF construct at **p**.



s, l_1 - Start and end line respectively of LCSAJ k+1.
l_2, e - Start and end line respectively of LCSAJ k+2.
(n) - LCSAJ n. $(1 \leq n \leq k+1)$

Fig. 3.4(b) k+2 LCSAJs exercised.

[17]We (implicitly) assume that LCSAJ 1 to LCSAJ k (inclusive) will always be exercised whenever the program is executed with the same data used before the IF construct with an ELSE part is inserted at statement **p**.

This means that for k+1 decision elements in the program execution trace, (k+1) - 1 (removed LCSAJ) + 2 (new LCSAJs) LCSAJs are exercised. The result then follows. ∎

Corollary 3.3.1

Let S be a program with n (n > 0) IF constructs, all of which have ELSE parts, in strict sequence (i.e., no nesting). Then any program execution trace from S implies that n + 1 LCSAJs are exercised (in S).

Proof

No nesting in S implies that any program execution trace from S must have n decision elements. The result then follows from Theorem 3.3. ∎

Note that the above two results (Theorem 3.3 and Corollary 3.3.1) do not hold if an IF construct with a null ELSE part exists. For example, consider the program in Fig. 3.5(a), whose LCSAJ are listed in Fig. 3.5(b). It contains two IF constructs, one of which has a null ELSE part. If 1T 2T is its program execution trace, then LCSAJs 3 and 6 are exercised. Clearly, Theorem 3.3 does not hold in this case.

Line					
1	S1	**LCSAJ**	**Start line**	**End line**	**Target line**
2	IF (1) Then				
3	S2				
4	IF (2) Then	1	1	2	8
5	S3	2	1	4	7
6	ENDIF	**3**	**1**	**7**	**11**
7	S4	4	7	7	11
8	ELSE	5	8	11	exit
9	S5	**6**	**11**	**11**	**exit**
10	ENDIF				
11	S6				

(a) (b)

Fig. 3.5 Illustrating necessity of restrictions in Theorem 3.3.

Next, consider a program with a single IF construct that has no ELSE part (See Fig 3.2 in this Section). Clearly, any program execution trace for this program contains a single decision element. Moreover, if the IF predicate evaluates to *True*, then only one LCSAJ is exercised, namely LCSAJ 2 (in Fig. 3.2(b)). Corollary 3.3.1 does not hold in this case.

3.2.2 LCSAJs in WHILE CONSTRUCTS

When a WHILE predicate evaluates to *False*, the predicate is the last statement to be exercised (in the WHILE construct) before control is transferred to the statement *immediately following* the corresponding ENDWHILE statement. This implies that the predicate is the end line of an LCSAJ and that the statement *immediately following* the ENDWHILE statement is the start line of another LCSAJ.

Recall that by assumption (Sec. 3.1), when the WHILE predicate evaluates to *True*, the ENDWHILE statement is the last to be exercised (in the WHILE construct), before control is transferred to the corresponding WHILE predicate. In this case :

i) the ENDWHILE statement is the end line of an LCSAJ; and

ii) the WHILE predicate is the start line of another (possibly the same) LCSAJ.

Line

```
1    S1
2    WHILE (1) DO
3     Begin
4      S2
5     Endwhile
6    S3
```

(a)

LCSAJ	Start line	End line	Target line
1	1	2	6
2	1	5	2
3	2	2	6
4	2	5	2
5	6	6	exit

(b)

Fig. 3.6 LCSAJs in an example WHILE construct.

The above concepts are illustrated with an example. Consider the program in Fig. 3.6(a). Its LCSAJs are listed in Fig. 3.6(b). If 1T 1F is the program execution trace, then

LCSAJs 2, 3 and 5 are exercised. Note that LCSAJ 4 is **not** counted as exercised, even though it has the same end line and target line as LCSAJ 2.

Observe that LCSAJs associated with WHILE constructs have :

a) start lines either :
   i) preceding the WHILE predicate; or
   ii) at a statement in the construct (inclusive of the ENDWHILE statement and the WHILE predicate); and
b) end lines at a statement in the construct (inclusive of the ENDWHILE statement and the WHILE predicate).

Based on the above discussion, we have the following results :

Lemma 3.4

LCSAJs associated with WHILE constructs have end lines at a statement in the construct (inclusive of the ENDWHILE statement and the WHILE predicate).

Proof

Follows from the discussion above. ∎

Lemma 3.5

If a WHILE predicate evaluates to *False*, then two LCSAJs are exercised.

Proof

Follows from the above discussion. ∎

3.2.3 LCSAJs in REPEAT CONSTRUCTS

When an UNTIL predicate evaluates to *False*, the predicate is the last statement to be exercised (in the REPEAT construct) before control is transferred to the corresponding

REPEAT statement. This implies that the predicate is the end line of an LCSAJ, and that the REPEAT statement is the start line of another (possibly the same) LCSAJ.

When the UNTIL predicate evaluates to *True*, control is transferred to the *statement immediately* following the predicate. Note that in this case, neither the UNTIL predicate nor the statement immediately following the UNTIL predicate, are end lines or start lines of LCSAJs (See Sec. 2.4)

We illustrate the above with an example. The program in Fig. 3.7(a). Its LCSAJs listed are in Fig. 3.7(b). If 1F 1T is the program execution trace, then LCSAJs 2 and 3 are exercised. As before, note that LCSAJ 4 is **not** counted as exercised, even though it has the same end line and target line as LCSAJ 2.

Line

1	S1
2	REPEAT
3	S2
4	UNTIL (1)
5	S5

(a)

LCSAJ	Start line	End line	Target line
1	1	5	exit
2	1	4	2
3	2	5	exit
4	2	4	2

(b)

Fig. 3.7 LCSAJs in an example REPEAT construct.

Observe that LCSAJs associated with REPEAT constructs have :

(a) start lines either :

i) preceding the REPEAT statement; or

ii) at a statement in the construct (inclusive of the REPEAT statement and the UNTIL predicate); and

(b) end lines either at :

i) a statement in the construct (other than the REPEAT statement) inclusive of the UNTIL predicate; or

ii) a statement following the UNTIL predicate.

Based on the above discussion, we have the following result :

Lemma 3.6

If an UNTIL predicate evaluates to *False* then at most two LCSAJs are exercised.

Proof

Follows from the above discussion. ∎

3.3 ADAPTIVE LCSAJ TESTING

In this section, we present an adaptive test strategy for LCSAJ coverage. We then prove that this strategy achieves LCSAJ coverage, modulo feasibility.

3.3.1 FUNDAMENTAL DEFINITIONS OF ADAPTIVE LCSAJ TESTING

We now introduce fundamental definitions that facilitate the presentation and proof of correctness of coverage of the *Adaptive LCSAJ Test Strategy*. First, however, an additional assumption used in the remainder of this Chapter and subsequent Chapters, is stated. (See Sec. 3.1 for the additional assumptions).

ix) Whenever an ENDWHILE statement is exercised, an "*" is recorded in the *trace output file*.

This assumption can be satisfied quite easily by adapting common software instrumentation techniques [Huan78, Prob81, Prob82a].

We now introduce our fundamental definitions.

An L-trace is the actual execution sequence of "*"'s and/or decision elements (see Sec. 3.2) obtained from the *trace output file* (by executing the program on data). It represents the execution history of the program on that data.

For example 1T 2T *1F could be an L-trace, in which predicates associated with integers "1", "2" and "1" evaluated to *True*, *True* and *False* respectively. Moreover, "*" indicates that an ENDWHILE statement was exercised. In addition, predicates associated with integers "1" and "2" were exercised (and evaluated) in the above order. (Clearly, the predicate corresponding to the integer "1" above is a WHILE predicate, because "1" is preceded by an "*").

The essential component of our adaptive testing strategy is called an L-sequence.

An **L-sequence** is either :

i) an **L-trace** with a **single decision element** and **no "*" 's**; or

ii) **a sub-sequence** of an **L-trace** that **starts** with an "*" or a decision element, **continues** with zero or more T-elements and **ends** with a decision element **such that the following two conditions are satisfied :**

1. An **L-sequence** can end with a T-element **if and only if :**
   i) the T-element is also the last decision element of the L-trace; or
   ii) the T-element is the decision element immediately preceding an "*".

2. An **L-sequence** can start with a T-element **if and only if** the T-element is at the beginning of the L-trace.

   (Note that an F-element can be part (stop/start) of two L-sequences).

We now need to distinguish between types of LCSAJs.

A **Primary LCSAJ** is an LCSAJ (Sec. 2.4) with at least one predicate in its linear code sequence, and a **Secondary LCSAJ** is an LCSAJ with no predicate in its linear code sequence. For example, the program in Fig 3.8(a) has primary LCSAJs 1-8 inclusive (See Fig. 3.8(b)) and secondary LCSAJs 9-12 inclusive.

Line		LCSAJ	Start line	End line	Target line
1	S1				
2	IF (1) THEN				
3	S2				
4	ELSE	**1**	**1**	**2**	**4**
5	S3	**2**	**1**	**3**	**7**
6	ENDIF	**3**	**4**	**8**	**16**
7	S4	**4**	**4**	**10**	**12**
8	IF (2) THEN	**5**	**4**	**11**	**15**
9	S5	**6**	**7**	**8**	**16**
10	IF (3) THEN	**7**	**7**	**10**	**12**
11	S6	**8**	**7**	**11**	**15**
12	ELSE	9	12	15	19
13	S7	10	15	15	19
14	ENDIF	11	16	19	exit
15	S8	12	19	19	exit
16	ELSE				
17	S9				
18	ENDIF				
19	S10				
	(a)		(b)		

Fig. 3.8 Illustrating Primary and Secondary LCSAJs.

We have the following definitions :

Let s be the start line and e the end line of an LCSAJ **L**. Then the **length** of **L** is given by $|s - e| + 1$.

To distinguish between L-sequences **L1** and **L2**, we define **L1** and **L2** to be **different** if and only if **L1 (L2)** has a decision element or "*", not in **L2 (L1)**.

A T-element (F-element) is **reversed**, if its *True* (*False*) value is changed to a *False* (*True*) value.

An **LCSAJ span** [Henn84] is a minimal partition of a program such that the *linear code sequence* of any LCSAJ is *wholly* contained within the partition. For example, the program in Fig. 3.9 has three LCSAJ spans (highlighted by the full traverse lines).

```
Line

1       S1
2       WHILE (1) DO
3        Begin
4           S2
5           IF (2) THEN
6              S3
  ---------------------------
7           ELSE
8              S4
9           ENDIF
10       Endwhile
  ---------------------------
11      S9
```

Fig. 3.9 Illustrating LCSAJ spans.

Finally, from the definition of an LCSAJ span we have the following properties[18]

PROPERTY III : The first line of an LCSAJ span is either at :

i) The first line of the program text;

ii) An ELSE statement; or

iii) The statement *immediately following* an ENDWHILE statement.

PROPERTY IV : The last line of an LCSAJ span is either at :

i) The last statement (in the program) that can be exercised;

ii) An ENDWHILE statement; or

iii) The statement *immediately preceding* an ELSE statement.

3.3.2 AN ADAPTIVE LCSAJ TEST STRATEGY

In this test strategy, the relationship between selecting subpaths through a program (represented by L-sequences) and exercising LCSAJs is explored. Let ***L1*** be an L-sequence in an L-trace ***L***. Then the *Adaptive LCSAJ Test Strategy* follows the following sequence of steps :

[18]Recall that by assumption (Sec. 3.1) only IF, WHILE and REPEAT control constructs are allowed.

For each (not previously selected) L-sequence ***L1*** in an L-trace ***L*** **do**

i) reverse the decision element to the right of the leftmost reversed element in ***L1***, i.e. do not reverse an element which is the reverse of an element in an existing L-sequence;

Note : if there is no leftmost reversed element, then reverse the first element in the L-sequence.

ii) Derive data that will cause the initial portion of ***L*** to the element reversed in ***L1*** to be generated; and

iii) Repeat i) and ii) for the remaining elements (starting in a left to right direction) of ***L1***. If no more elements to be reversed in ***L1***, proceed to analyze another L-sequence in ***L***.

Endfor

This procedure is repeated for each L-trace ***L*** until each (not previously selected) L-sequence ***L1*** in ***L*** has been selected, or until LCSAJ coverage modulo feasibility[19] has been achieved. Note that if an F element is part (last element / first element) of two L-sequences, then the F-element in the latter L-sequence is not reversed. This is, of course, because it will have been reversed once already by applying the procedure to the first L-sequence.

For now, however, assume that this strategy achieves LCSAJ coverage. Then it (the strategy) is an adaptive white-box testing strategy (See Sec. 2.3) because :

i) Previously exercised paths (represented by L-traces) are used to guide the selection of subsequent paths; and

ii) Incremental addition of one path (determined by L-sequences) to existing (exercised) paths in order to increase the degree of code coverage attained (i.e. to increase the number of LCSAJs exercised).

---

[19]Recall (Sec. 2.3.2) that LCSAJ coverage modulo feasibility means that the strategy does not claim to cause unrealizable LCSAJs to be exercised, since no input exists that will cause such LCSAJs to be exercised

We illustrate the Adaptive LCSAJ Test Strategy with an example.

EXAMPLE 3.1

Consider the program shown in Fig. 3.10. Its LCSAJs are listed in Table 3.1. (For now, ignore the last column in Table 3.1).

```
Line

1       S1
2       WHILE (1) DO
3        Begin
4          S2
5          IF (2) THEN
6             S3
7          ENDIF
8          S4
9        Endwhile
10      S9
```

Fig. 3.10 Example 3.1.

LCSAJ	Start line	End line	Target line	Exercised
1	1	2	10	2
2	1	5	8	3
3	1	9	2	1
4	2	2	10	1
5	2	5	8	4
6	2	9	2	1
7	8	9	2	3
8	10	10	exit	1

Table 3.1 LCSAJs for the program in Fig. 3.10.

The strategy achieves LCSAJ coverage as follows. First, the program is executed on arbitrary input, resulting in, say L-trace L = 1T 2T *1T 2T *1T 2T *1F (A "1" is recorded in the last column of Table 3.1 for each previously unexercised LCSAJ). LCSAJs 3, 6, 4 and 8 are exercised; in particular; L-sequence 1T 2T implied that LCSAJ 3 was exercised, L-sequence *1T 2T implied that LCSAJ 6 was exercised, and L-sequence *1F implied that LCSAJs 4 and 8

were exercised. (Hereafter "implied that" and "exercised" are used interchangeably. Thus L-sequence *1F exercised LCSAJs 4 and 8).

Next, L-sequence 1T 2T is selected. Decision element 1T is reversed, and we derive data that will cause 1F to be generated. Suppose that L-trace **L1** = 1F is generated. Then, LCSAJ 1 is exercised ("2" in Table 3.1). Decision element 2T (in L-sequence 1T 2T) is next to be reversed, giving say L-trace **L2** = 1T 2F *1F. LCSAJs 2, 7, 4 and 8 are exercised ("3" in Table 3.1); in particular, L-sequence 1T 2F exercised LCSAJs 2 and 7, and L-sequence *1F exercised LCSAJs 4 and 8. Note that no more decision elements can be reversed in L-sequence 1T 2T.

At this stage, L-sequence *1T 2T is selected. Decision element 1T is reversed, giving say, L-trace **L3** = 1T 2T *1F. LCSAJs 3, 4 and 8 are exercised; in particular L-sequence 1T 2T exercised LCSAJ 3, and L-sequence *1F exercised LCSAJs 4 and 8. Note that no unexercised LCSAJ was exercised when L-trace **L3** was generated.

Next, decision element 2T in L-sequence *1T 2T is reversed, giving say, L-trace **L4** = 1T 2T *1T 2F * 1F. LCSAJs 3, 5, 7, 4 and 8 are exercised ("4" in Table 3.1); in particular, L-sequence 1T 2T exercised LCSAJ 3, L-sequence *1T 2F exercised LCSAJs 5 and 7, and L-sequence *1F exercised LCSAJs 4 and 8.

All LCSAJs have been exercised, i.e., LCSAJ coverage has been achieved.

Next, we prove several results on the capabilities of the *Adaptive LCSAJ Test Strategy*. We reiterate that all results are presented *modulo feasibility*. (See the remarks in Sec. 2.3.2 on branch coverage modulo feasibility.and LCSAJ coverage modulo feasibility)

3.3.2.1 Preliminary Results

We first state several results that follows from the definition of an L-trace and an L-sequence (Sec. 3.3.1), namely :

Lemma 3.7.1

An "*" is in an L-trace if and only if a WHILE predicate evaluates to *True*.

Proof

Follows from assumption ix) (Sec. 3.3.1), and from the definition of an L-trace.

∎

Lemma 3.7.2

A decision element is always at the beginning of an L-trace.

Proof

Follows from Lemma 3.7.1 and the definition of an L-trace. ∎

Lemma 3.7.3

F-elements and "*" are always at the beginning of L-sequences.

Proof

Follows from the definition of an L-sequence. ∎

Note that F-elements and "*" 's in L-traces usually indicate that two distinct LCSAJs are exercised (Lemmas 3.2.1 and 3.5 in Sec.3.2.1 and 3.2.2 respectively). However, from Lemma 3.6 (Sec. 3.2.3), when the UNTIL predicate evaluates to *False*, the corresponding F-element indicates that at most two LCSAJs are exercised.

Next, we state a result that gives an upper bound on the number of L-sequences that can be generated from another L-sequence.

Theorem 3.8

Let $S = A\ 1X_1\ 2X_2 \ldots nX_n$ be an L-sequence with n ($n > 0$) decision elements, where X_i ($i = 2, \ldots n-1$) $\in \{T\}$, $A \in \{\phi,*\}$ and $X_1, X_n \in \{T,F\}$. Then at least n L-sequences can be generated from S, with the restriction that "1" is a component of the first decision element of these L-sequences.

Proof

There are n decision elements in S and only one decision element can be reversed at a time. In addition, "1" is a component of the first decision element in S. This implies that for each decision element reversed, at least one L-sequence (with "1" as a component of the first decision element) is generated. ∎

The following results combine to prove that the Adaptive LCSAJ Test Strategy achieves branch coverage.

Lemma 3.9.1

The *Adaptive LCSAJ Test Strategy* causes every predicate in the program to be exercised at least once during testing.

Proof

Clearly D_1, the first predicate encountered, is exercised. Consider a predicate D_k that has not been exercised, and without loss of generality, assume that all predicates at a "shorter distance" in the program have already been exercised. Then this assumption implies that an immediate predecessor, predicate D_{k-1}, has already been exercised. Now, L-sequences are generated by the strategy and eventually one of them contain a decision element associated with predicate D_{k-1}. Reversal of this decision element will result in D_k being exercised (if not already exercised). The result then follows. ∎

The following result is immediate.

Theorem 3.9

The Adaptive LCSAJ Test Strategy achieves branch coverage.

Proof

From Lemma 3.9.1, the strategy causes all predicates to be exercised. This implies that every predicate in the program is associated with a decision element (of an L-sequence). Since the strategy reverses every decision element (of an L-sequence), all decision outcomes are exercised. That is, the strategy achieves branch coverage. ∎

Corollary 3.9.1

The *Adaptive LCSAJ Test Strategy* achieves statement coverage.

Proof

Branch coverage implies statement coverage [Myer79]. The result then follows from Theorem 3.9. ∎

We now prove our first result on LCSAJ coverage --- the strategy cause all secondary LCSAJs to be exercised.

Lemma 3.10.1

Let **L** be a secondary LCSAJ with start line **k** and let **s** be the first line of the program. Then $s \neq k$.

Proof

From Assumption v) (Sec. 3.1), every program has at least one structured construct. This implies that any LCSAJ with start line **s** has at least one predicate in its corresponding linear code sequence. Since secondary LCSAJs have no predicates, the result then follows. ∎

Lemma 3.10.2

Let L be a secondary LCSAJ with start line k and let s be a REPEAT statement. Then $s \neq k$.

Proof

There are two cases to consider. Either the REPEAT construct has at least one structured control construct between the REPEAT statement and the corresponding UNTIL predicate, or no such control construct exists. (Without loss of generality, assume that these constructs are not nested).

Case 1

Clearly, if there exists at least one at least one structured control construct between the REPEAT statement and the corresponding UNTIL predicate, then any LCSAJ with start line s has at least one predicate in its corresponding linear code sequence. In this case, s is not the start line of a secondary LCSAJ.

Case 2

If no structured control construct is between the REPEAT statement and the corresponding UNTIL predicate, then any LCSAJ with start line s will have the UNTIL predicate in its corresponding linear code sequence. In this case, s is not the start line of a secondary LCSAJ.

All cases have been considered. The result follows. ∎

Lemma 3.10.3

Let s be the start line of a secondary LCSAJ L in a program P. Then L is the only LCSAJ in P with start line s.

Proof

Follows from the definition of an LCSAJ (Sec. 2.4) and a secondary LCSAJ (Sec. 3.3.1). ∎

Theorem 3.10

If branch coverage is achieved then all secondary LCSAJs are exercised.

Proof

A secondary LCSAJ (Sec. 3.3.1) has no predicate in its corresponding linear code sequence. This implies (from PROPERTY II (Sec 2.4) and Lemmas 3.10.1 and 3.10.2) that possible start lines of secondary LCSAJs are either :

i) an ELSE statement;

ii) the statement *immediately following* an ENDWHILE statement (if not a predicate); or

iii) the statement *immediately following* an ENDIF statement (if not a predicate).

These start lines are exercised respectively **if and only if**

i) an IF predicate (in an IF construct with an ELSE part) evaluates to *False*;

ii) a WHILE predicate evaluates to *False*; or

iii) either an IF predicate (in an IF construct with an ELSE part) evaluate to *True*, or an IF predicate (in an IF construct with no ELSE part) evaluates to *False*.

Since branch coverage must be attained (by assumption), then each of the start lines discussed above will be exercised. The result then follows from Lemma 3.10.3. ∎

Theorem 3.11

The *Adaptive LCSAJ Testing Strategy* causes all secondary LCSAJs to be exercised.

Proof

The result follows from Theorems 3.9 and 3.10. ∎

3.3.2.2 Proof of Completeness

In this section, we prove that the *Adaptive LCSAJ Test Strategy* achieves LCSAJ coverage modulo feasibility. First, however, we introduce a term used in the remainder of this Section and subsequent Sections :

A start line s of an LCSAJ is *indicated by* an F(T)-element, if the F(T)-element implies that s is a start line.

A start line s of an LCSAJ is *indicated by* an "*" if it (the "*") implies that s is a start line. (Clearly, only WHILE predicates are *indicated by* "*" 's).

A end line e of an LCSAJ is *indicated by* an F(T)-element, if the F(T)-element implies that e is an end line.

We illustrate with an example --- from PROPERTY II (Sec. 2.4), a REPEAT statement is a possible start line of an LCSAJ. Moreover, if it is a start line, it is *indicated by* an F-element (associated with the corresponding UNTIL predicate). Note that neither "*" 's nor decision elements can represent **(only indicate)** start lines or end lines of LCSAJs.

We now prove a major result that involves start lines in LCSAJ spans (Sec. 3.3.1).

Theorem 3.12

All start lines in an LCSAJ span (other than the first line of the program text) are *indicated by* either :

(i) an F-element; (ii) an "*"; or (iii) a T-element;

of an L-sequence.

Proof (by exhaustive analysis of cases) :

From PROPERTY II (Sec. 2.4), possible start lines of LCSAJs are :

i) an ELSE statement;

ii) the statement *immediately following* an ENDWHILE statement; or;

iii) a REPEAT statement.

Thus, these start lines are exercised respectively **if and only if** a corresponding IF predicate (in an IF construct with an ELSE part), WHILE predicate and UNTIL predicate evaluate to *False*, *False* and *False* respectively. That is, these start lines are each *indicated by* corresponding F-elements (of an L-sequence).

Again, from PROPERTY II, the WHILE predicate is a possible start line of an LCSAJ. This start line is exercised **if and only if** control is transferred from either :

i) an ENDWHILE statement;

ii) another WHILE predicate;

iii) an IF predicate (in an IF construct with no ELSE part); or

iv) the statement *immediately preceding* an ELSE statement (of an IF construct);

to the (WHILE) predicate. Thus, this start line is *indicated by* a corresponding "*", F-element, F-element and T-element respectively.

Finally, from PROPERTY II, the statement *immediately following* an ENDIF statement is a possible start line of an LCSAJ. This start line is exercised **if and only if** either :

i) an IF predicate (in an IF construct with an ELSE part) evaluates to *True*; or

ii) an IF predicate (in an IF construct with no ELSE part) evaluates to *False*.

Thus, this start line is *indicated by* a corresponding T-element and F-element respectively.

All start lines (other than the first line of the program text) listed in PROPERTY II have been considered. The result then follows. ∎

Our next set of results proves that the Adaptive LCSAJ Test Strategy causes all LCSAJs whose start lines are indicated by F-elements and "" 's (of an L-sequence), to be exercised.*

Theorem 3.13

Let s be the start line of an LCSAJ and let L be the longest LCSAJ with start line s. Then the end line of L is the last line of an LCSAJ span.

Proof

Follows from the definition of an LCSAJ span (Sec 3.3.1). ∎

Theorem 3.14

Let s be the start line (of an LCSAJ) *indicated by* either

i) an F-element; or ii) an "*";

at the beginning of an L-sequence.

Then, the strategy causes the longest LCSAJ with start line s to be exercised (if not already exercised).

Proof

Let L be such an L-sequence, and let e be the end line of the LCSAJ with start line s that is exercised (by the program). In addition, let $\underline{e}$ be the end line of the longest LCSAJ with start line s. Then either :

i) $|s - \underline{e}| = |s - e|$; or ii) $|s - \underline{e}| > |s - e|$.

Case 1

If $|s - \underline{e}| = |s - e|$ then clearly the results hold. (Note that from Theorem 3.13, $\underline{e}$ is the last line of an LCSAJ span).

Case 2

If $|s - \underline{e}| > |s - e|$ then from Theorem 3.13 e is not the last line of an LCSAJ span. From PROPERTY I (Sec 2.4) and PROPERTY IV (Sec. 3.3.1), e is either :

i) an IF predicate; ii) a WHILE predicate; or iii) an UNTIL predicate.

Thus, **e** is *indicated by* a corresponding F-element. Moreover, this F-element is the last decision element of **L**.

Now, the strategy reverses the last decision element of **L** (i.e, the F-element). This causes a new L-sequence, say **L1**, to be generated. Moreover, an LCSAJ (with start line s) of increased length is exercised. Suppose that **a** is the end line of this (exercised) LCSAJ. Then either :

iii) $|s - \underline{e}| = |s - a|$; or iv) $|s - \underline{e}| > |s - a|$.

If (iii) occurs, then the result follows. Otherwise, **a** is indicated by an F-element that is the last decision element of **L1**. Reversal of this decision element causes a new L-sequence to be generated. Moreover, an LCSAJ (with start line s) of increased length is exercised. Continuing in this manner (i.e., reversing the last decision element of **L1**, etc.), the last line of the LCSAJ span containing s is reached. From Theorem 3.13, this line is the end line of the longest LCSAJ with start line s. The result then follows.

All cases have been considered. The result follows. ∎

Corollary 3.14.1

Let s be the start line of an LCSAJ *indicated by* either :

i) an F-element; or ii) an "*";

at the beginning of an L-sequence. Then the longest LCSAJ with start line s is "represented by" either :

i) an F-element ; or ii) an "*" ;

followed by a finite sequence of T-elements.

Proof

Follows from the proof of Theorem 3.14. ∎

Theorem 3.15

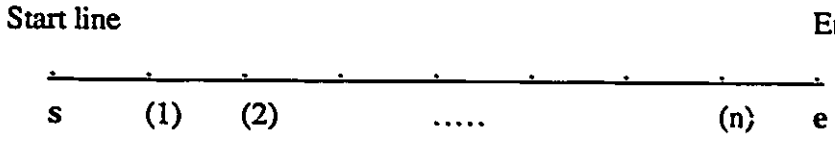
Let s be the start line of an LCSAJ *indicated by* either :

i) an F-element ; or ii) an "*";

at the beginning of an L-sequence. Then the strategy causes all LCSAJs with start line s.to be exercised.

Proof

From Theorem 3.14, the strategy causes the longest LCSAJ with start line s to be exercised. From Corollary 3.14.1, this LCSAJ is "represented by" either an F-element or an "*", followed by a sequence of, say n (n > 0) T-elements (see Fig 3.11). (Trivially, the result holds for n = 0).



(k), (1≤ k ≤n) - predicates associated with a T-element.
s, e - start line and end line respectively of longest LCSAJ with start line s.

Fig. 3.11 Illustrating the longest LCSAJ.

Each of the predicates in Fig. 3.11, is a possible end line of an LCSAJ with start line s. The strategy reverses each decision element; in particular, the T-elements associated with predicates (k), (1 ≤ k ≤ n) in Fig. 3.11. This implies that corresponding predicates each become the end line of an LCSAJ with start line s. The result then follows. ∎

Corollary 3.15.1

If the longest LCSAJ with start line s can be "represented by" either an F-element or an "*", followed by n (n > 0) T-elements, the strategy causes all Primary LCSAJs with start line s to be exercised.

Proof

The result follows from the proof of Theorem 3.15, and from the definition of a Primary LCSAJ. ∎

Next, we prove that the strategy causes all LCSAJs whose start lines are indicated by T-elements to be exercised

Theorem 3.16

Let s be the start line of an LCSAJ *indicated by* a T-element of an L-sequence L. Then the strategy causes the longest LCSAJ with start line s to be exercised.

Proof

Let X denote this T-element, and let $\underline{e}$ be the end line of the longest LCSAJ with start line s. Also, let e be the end line of the LCSAJ with start line s, that is actually exercised. Then either :

(i) $|s - \underline{e}| = |s - e|$; or (iii) $|s - \underline{e}| > |s - e|$.

The remainder of the proof is similar to Theorem 3.14, if we realize that all decision elements preceding X (inclusive) remain unchanged by the strategy whenever decision elements following X are reversed. ∎

Corollary 3.16.1

Let s be the start line of an LCSAJ *indicated by* a T-element of an L-sequence. Then the longest LCSAJ with start line s is "represented by" this T-element followed by a finite sequence of T-elements.

Proof

Follows from the proof of Theorem 3.16. ∎

Theorem 3.17

Let s be the start line of an LCSAJ *indicated by* a T-element of an L-sequence L. Then the strategy causes all LCSAJs with start line s to be exercised.

Proof

The proof is similar to Theorem 3.15 (Corollary 3.16.1 is substituted for Corollary 3.14.1). ∎

Corollary 3.17.1

If the longest LCSAJ with start line s can be "represented by" an T-element followed by n (n > 0) T-elements, the strategy causes all Primary LCSAJs with start line s to be exercised.

Proof

Similar to Corollary 3.15.1. (Theorem 3.17 is substituted for Theorem 3.15). ∎

The following results prove that the strategy achieves LCSAJ coverage, modulo feasibility.

Theorem 3.18

Let s be the start line of an LCSAJ *indicated by* either :

i) an F-element at the beginning of an L-sequence;

ii) a T-element of an L-sequence; or

iii) an "*" at the beginning of an L-sequence.

Then the strategy causes all LCSAJs with start line s to be exercised.

Proof

The result follows from Theorems 3.15 and 3.17. ∎

Corollary 3.18.1

Let f be the first line of a program. Then the strategy causes all LCSAJs with start line f to be exercised.

Proof

The first line of a program is always exercised. Thus, f is always *indicated by* the decision element at the beginning of an L-trace. (From Lemma 3.7.2, a decision element is always at the beginning of an L-trace). The result then follows from Theorem 3.18. ∎

Theorem 3.19

The *Adaptive LCSAJ Test Strategy* causes all LCSAJ in an LCSAJ span to be exercised.

Proof

From Theorem 3.12, all start lines (other than the first line of a program) in an LCSAJ span are *indicated by* either an F-element, a T-element or an "*" of an L-sequence. From Theorem 3.18, the strategy causes all LCSAJs (in an LCSAJ span), with start lines *indicated by* either :

i) an F-element at the beginning of an L-sequence;

ii) a T-element of an L-sequence; or

iii) an "*" at the beginning of an L-sequence;

to be exercised. From Lemma 3.7.3, F-elements and "*" are always at the beginning of an L-sequence. Thus, the strategy causes all LCSAJs with start lines *indicated by* F-elements, T-elements and "*" (of an L-sequence) to be exercised.

We need to show that all such start lines are actually exercised. From Theorem 3.9, the *Adaptive LCSAJ Test Strategy* achieves branch coverage. This implies, together with Lemma 3.7.1, that all such start lines are actually exercised. Thus, the strategy

causes all LCSAJs (in an LCSAJ span) with start lines *indicated by* F-elements, T-elements and "*" 's to be exercised.

The only start line (in an LCSAJ span) not considered is the first line of the program text. From Corollary 3.18.1, the strategy causes all LCSAJs with such start lines to be exercised.

The result then follows. ∎

We now prove the main result.

Theorem 3.20

The *Adaptive LCSAJ Test Strategy* achieves LCSAJ coverage.

Proof

Follows from Theorem 3.19 and from the definition of an LCSAJ span (Sec 3.3.1). ∎

3.4 SUMMARY

Thus, we have proved that the *Adaptive LCSAJ Test Strategy* achieves LCSAJ coverage modulo feasibility. The strategy works by generating (in the worst case) all possible L-sequences. Note that this strategy may not be optimal in terms of causing unexercised LCSAJs to be exercised. For example, if newly generated L-sequences (and, by extension, corresponding L-traces) do not cause at least one unexercised LCSAJ to be exercised, testing resources are wasted. This was earlier demonstrated in Example 3.1 (Sec. 3.3.2), in which a newly generated L-trace, namely L-trace **L4**, caused no unexercised LCSAJ to be exercised. This issue (among others) is addressed in the next Chapter.

CHAPTER 4

IMPLEMENTATION CONSIDERATIONS FOR THE ADAPTIVE LCSAJ TEST STRATEGY

The *Adaptive LCSAJ Test Strategy* achieves LCSAJ coverage modulo feasibility by generating (in the worst case) all possible L-sequences. In an efficient implementation of this strategy, at least one unexercised LCSAJ should be exercised whenever a decision element is reversed and the corresponding L-trace determined. In this Chapter, we develop an implementation of the *Adaptive LCSAJ Test Strategy* that reflects the above (practical) testing concern. In addition, we prove that this implementation (of the strategy) achieves LCSAJ coverage modulo feasibility. A detailed example is then used to illustrate the applicability of our proposed test implementation strategy. Moreover, we refer to the strategy discussed in Chapter 3 as the **left to right Adaptive LCSAJ Test Strategy**, and, the strategy developed in this Chapter as the **modified Adaptive LCSAJ Test Strategy**.

4.1 IMPLEMENTATION CONSIDERATIONS

In this section, we first show that the *left to right Adaptive LCSAJ Test Strategy* need not exhaustively select every L-sequence in an L-trace nor reverse every decision element in an L-sequence in order to achieve LCSAJ coverage (modulo feasibility). Then, we give guidelines for efficiently using this approach to attain LCSAJ coverage (modulo feasibility).

4.1.1 JUSTIFICATION AND METHOD FOR EFFECTIVE SELECTION OF L-SEQUENCES

In this section, we develop a method for effective selection of L-sequences based on several results that prove that the *left to right Adaptive LCSAJ Test Strategy* need not exhaustively select every (not previously selected) L-sequence in an existing L-trace nor reverse every decision element of an L-sequence, in order to achieve LCSAJ coverage (modulo feasibility).

We begin by stating a result that follow from the definition of an L-sequence (Sec. 3.3.1) and an LCSAJ (Sec 2.4).

Lemma 4.1.1

Every LCSAJ is associated with at least one L-sequence.

Proof

From Theorem 3.12 (Sec. 3.3.2.2) all start lines in an LCSAJ span (other than the first line of the program text) are *indicated by* either an F-element, a T-element or an "*" of an L-sequence. From the proof of Corollary 3.18.1 (Sec. 3.3.2.2), the first line of the program text is indicated by the first decision element of an L-trace (and corresponding L-sequence). The result then follows from Theorem 3.20 (Sec. 3.3.2.2) and from the definition of an L-sequence. ∎

Theorem 4.1

If no L-sequence different from existing L-sequences is generated when a decision element is reversed, then no (previously) unexercised LCSAJ is exercised.

Proof

Follows from Lemma 4.1.1 ∎

Theorem 4.2

If a (previously) unexercised LCSAJ is exercised then at least one L-sequence different from existing L-sequences was generated (when the decision element was reversed).

Proof

Follows from Lemma 4.1.1. ∎

Thus Theorems 4.1 and 4.2 give the motivation for generating at least one L-sequence different from existing L-sequences, whenever a decision element is reversed.

Next, we introduce the following term :

Let **L1** and **L2** be two L-sequences that start with the same "*" or decision element. Then **L1** is **longer** than **L2** if the number of decision elements in **L1** is greater than those in **L2**. Moreover, if **L1** and **L2** have the same number of elements, but **L1(L2)** ends with a T-element and **L2(L1)** ends with an F-element, then **L1(L2)** is **longer** than **L2(L1)**.

Now, suppose that the *left to right Adaptive LCSAJ Test Strategy* is modified so that only longest (not previously selected) L-sequences are selected. Then the following results hold.

Lemma 4.3.1

Let **L** be the longest existing L-sequence that starts with a particular decision element or "*". Then the longest L-sequence **P** that starts with the same decision element or "*" is eventually generated (if not already generated).

Proof

The strategy causes decision elements to be reversed. This implies that the longest L-sequence that starts with a particular decision element or "* will be generated (if not already generated). ∎

Lemma 4.3.2

Let L be the longest existing L-sequence that starts with a particular decision element or "*". Then all L-sequences that start with the same decision element or "*" are eventually generated (if not already generated).

Proof

Similar to Lemma 4.3.1, i.e., since all decision elements in the longest L-sequence are successively reversed, all L-sequences with the same start point will be generated. ∎

Theorem 4.3

If the *left to right Adaptive LCSAJ Test Strategy* selects only longest (not previously selected) L-sequences then LCSAJ coverage (modulo feasibility) is still achieved.

Proof

By exhaustive enumeration of L-sequences, all L-sequences can be generated. ∎

In effect, Theorem 4.3 states that the *left to right Adaptive Test Strategy* need only select (in a left to right direction) longest (not previously selected) existing L-sequences that start with a particular decision element or "*", in order to achieve LCSAJ coverage, modulo feasibility.

The *left to right Adaptive Test Strategy* strategy works by generating (in the worse case) all L-sequences. Theorems 4.1 and 4.2 give the motivation for generating at least one L-sequence different from existing L-sequences, whenever a decision element is reversed. Thus, there is a need to avoid generating L-sequences which have already (or will eventually) be generated. That is, we should avoid generating duplicate L-sequences.

To introduce a method to accomplish this, we need the following definition :

Let S be an L-sequence that starts with either a decision element, say X, or an "*". Also let Y be an element reversed in S. Then its **initial L-sequence** is a subsequence of S that starts at the same point as S (i.e., either the "*" or corresponding X), and ends with Y. For example, if decision element 2T is reversed in L-sequence 1T 2T 3F, then 1T 2F is the corresponding initial L-sequence.

We now have the following results.

Lemma 4.4.1

Let s be an initial L-sequence and $\mathbf{Q} = \{l_1, l_2, ..., l_n\}$, where $l_1, l_2, ..., l_n$ are existing L-sequences. Then if $s \in \mathbf{Q}$ and s is generated, a duplicate L-sequence will be generated.

Proof

Since $s \in \mathbf{Q}$, the result follows. ∎

Therefore our approach will be to select initial L-sequences which are not existing L-sequences. The effectiveness of this approach is shown in the following results.

Lemma 4.4.2

Let s be an initial L-sequence and $\mathbf{Q} = \{l_1, l_2, ..., l_n\}$, where $l_1, l_2, ..., l_n$ are existing L-sequences. Then if $s \notin \mathbf{Q}$, s is itself an L-sequence and s is generated, a non-duplicate L-sequence will be generated.

Proof

Since s is itself an L-sequence and $s \notin \mathbf{Q}$, the result follows. ∎

Lemma 4.4.3

Let s be an initial L-sequence that starts with a particular decision element or "*", and let $Q = \{l_1, l_2, ..., l_n\}$, where $l_1, l_2, ..., l_n$ are existing L-sequences. Suppose that the *left to right Adaptive LCSAJ Test Strategy* always selects longest (not previously selected) L-sequences. Then if s is a subsequence of an existing L-sequence l_i (i =1, 2, .. n) that starts at the same point as s and $s \notin Q$, the corresponding L-sequence generated (i.e., the L-sequence that starts with the same decision element or "*") when s is generated will cause a duplicate L-sequence to be eventually generated.

Proof

Since longest L-sequences are selected, then from Lemma 4.3.2 all L-sequences that start with a particular decision element or "*" will be eventually generated. The result then follows. ∎

Lemma 4.4.3 suggests checking whether an initial L-sequence s is already a subsequence of an existing L-sequence that starts at the same point as s. If it is, then no further consideration need be given to s since a duplicate L-sequence will eventually be generated. Thus, we have the following result.

Theorem 4.4

Let s be an initial L-sequence, created by reversing a decision element in a longest existing L-sequence, that starts with a particular decision element or "*". Then if s is not a subsequence of an existing L-sequence that starts at the same point as s, a non-duplicate L-sequence (that starts at the same point as s) will be generated.

Proof

If s is an L-sequence then from Lemmas 4.4.2 the result follows. If s is itself not an L-sequence, then it is associated with an L-sequence different from existing L-sequences, to be generated. The result then follows. ∎

We now prove the main results of this section.

Theorem 4.5

The *left to right Adaptive LCSAJ Strategy* need only :

i) select (in a left to right direction) longest (not previously selected) L-sequences of an L-trace;

ii) generate initial L-sequences that are themselves not subsequences of existing L-sequences that start at the same point as the corresponding initial L-sequences;

in order to achieve LCSAJ coverage (modulo feasibility).

Proof

The results follows from Theorem 4.3 and Theorem 4.4. ∎

Theorem 4.6

If the *left to right Adaptive LCSAJ Strategy* :

i) selects (in a left to right direction) longest (not previously selected) L-sequences of an L-trace; and

ii) generates initial L-sequences that are themselves not subsequences of existing L-sequences that start at the same point as corresponding initial L-sequences,

then at least one non-duplicate L-sequence is generated when a decision element is reversed.

Proof

The result follows from Theorems 4.3 and 4.4. ∎

Thus, we have proved (in Theorem 4.5) that the *left to right Adaptive LCSAJ Test Strategy* need not exhaustively select every (not previously selected) L-sequence in an L-trace nor reverse every decision element in an L-sequence, in order to achieve LCSAJ coverage (modulo feasibility). Moreover, Theorem 4.6 states that, if the modifications suggested in Theorem 4.5

are adopted, at least one non-duplicate L-sequence is generated whenever a decision element is reversed. (We hope that whenever a non-duplicate L-sequence is generated at least one unexercised LCSAJ is exercised).

Therefore our method for effective selection of L-sequences will be to incorporate Theorem 4.5 into the *left to right Adaptive LCSAJ Test Strategy*. We now illustrate this method in the following example. In addition, Theorem 4.6 is illustrated.

EXAMPLE 4.1

Consider **EXAMPLE 3.1** (See Sec. 3.3.2). Now, L-trace L = 1T 2T *1T 2T *1T 2T *1F is the L-trace obtained after the program was first executed on arbitrarily input. The longest L-sequence (in this L-trace) that starts with "*" is *1T 2T, and the longest L-sequence that starts with decision element 1T is 1T 2T[20].

From L-sequence *1T 2T, initial L-sequences *1F and *1T 2F can be generated. However, initial L-sequence *1F already exists, i.e., it is an L-sequence in L-trace **L**, implying that only initial L-sequence *1T 2F should be generated (L-trace **L4** in **EXAMPLE-3.1**), since it is not a subsequence of an existing L-sequence that starts with the corresponding "*". Similarly, from L-sequence 1T 2T, initial L-sequences 1F and 1T 2F can be generated (L-traces **L1** and **L2** respectively in **EXAMPLE 3.1**).

These are all of the initial L-sequences that can be generated from L-trace **L**. (In fact, their generation caused all unexercised LCSAJs (in **EXAMPLE 3.1**) to be exercised.

Thus, by our method for effective selection of L-sequences, we achieved LCSAJ coverage by considering one fewer L-sequence (in L-trace **L**) than with the the *left to right Adaptive LCSAJ Test Strategy*. Moreover, we note that at least one non-duplicate L-sequence was generated whenever a decision element was reversed, and at least one unexercised LCSAJ was exercised whenever an L-trace was generated. This was not the case in **EXAMPLE 3.1** in

[20]All L-sequences in L-trace L start with either decision element 1T or "*".

which L-trace L3 cause no unexercised LCSAJ to be exercised, and no non-duplicate L-sequence to be generated.

4.1.2 EXERCISING AT LEAST ONE UNEXERCISED LCSAJ

In this section, we show how to modify the *left to right Adaptive LCSAJ Test Strategy* to ensure that for every decision element reversed at least one (previously) unexercised LCSAJ is exercised. (Note that from Theorem 4.2, this implies that at least one non-duplicate L-sequence is generated).

First, we assume that the modifications suggested in Theorem 4.5 are adopted. That is, the *left to right Adaptive Test Strategy* selects only longest (not previously selected) L-sequences and generates initial L-sequences that are themselves not subsequences of existing L-sequences that start at the same point as corresponding initial L-sequences. Then, we show that even if these modification are adopted, it is still possible for no unexercised LCSAJ to be exercised when at least one non-duplicate L-sequence is generated. (That is, we show that Theorem 4.6 does not suffice for causing at least one unexercised LCSAJ to be exercised). We then give guidelines for efficiently using the modifications suggested in Theorem 4.5 to attain LCSAJ coverage (modulo feasibility).

We begin by considering the following example.

EXAMPLE 4.2

The program in Fig. 4.1 has its LCSAJs listed in Table 4.1 (for now, ignore the last column in Table 4.1).

```
Line

1       S1
2       IF ( 1 ) THEN
3         S2
4         IF ( 2 ) THEN
5           S3
6         ELSE
7           S4
8         ENDIF
9         S5
10      ELSE
11        S6
12      ENDIF
13      S7
14      IF (3) THEN
15        S8
16      ELSE
17        S9
18      ENDIF
19      S10
```

Fig. 4.1 Example 4.2.

LCSAJ	Start line	End line	Target line	Exercised
1	1	2	10	2
2	1	4	6	3
3	1	5	9	1
4	6	9	13	3
5	9	9	13	1
6	10	14	16	2
7	10	15	19	4
8	13	14	16	3
9	13	15	19	1
10	16	19	exit	2
11	19	19	exit	1

Table 4.1 LCSAJs for the program in Fig 4.1.

The *left to right Adaptive LCSAJ Test Strategy* (incorporating the modifications suggested in Theorem 4.5) achieves LCSAJ coverage as follows. First, the program is executed on arbitrarily chosen input, resulting in say, L-trace **L1** = 1T 2T 3T. LCSAJs 3, 5, 9 and 11 are exercised. (A "1" is recorded in last column of Table 4.1 for each previously unexercised LCSAJ).

Since L-trace **L1** = 1T 2T 3T is (by definition) an L-sequence, we select L-sequence **L1**. Then, we derive data that will cause initial L-sequence 1F to be generated (since 1F is not a subsequence of an existing L-sequence) giving say, L-trace **L2** = 1F 3F ("2" in Table 4.1). Next, we derive data that will cause initial L-sequence 1T 2F to be generated (since 1T 2F is not a subsequence of existing L-sequences, namely 1T 2T 3T and 1F 3F), resulting in say, L-trace **L3** = 1T 2F 3F ("3" in Table 4.1). Finally, initial L-sequence 1T 2T 3F is generated (since 1T 2T 3F is not a subsequence of an existing L-sequence that starts with T-element 1T), resulting in say, L-trace **L4** = 1T 2T 3F. LCSAJs 3, 5, 8 and 10 are exercised. However, these LCSAJs were exercised when L-traces **L1**, **L2** and **L3** were generated, i.e., L-trace **L4** caused no unexercised LCSAJ to be exercised, implying that testing resources were wasted in generating L-trace **L4**.

At this stage, note that no initial L-sequences can be further generated from L-sequence **L1**. Moreover, since L-trace **L4** = 1T 2T 3F is (by definition) an L-sequence, and decision element 3F is the leftmost reversed element in this L-sequence, no initial L-sequences can be generated from L-sequence **L4**. Thus, we apply the strategy to L-traces **L2** and **L3**. Suppose that L-trace **L2** is selected. Since L-trace **L2** = 1F 3F is (by definition) an L-sequence, we derive data that will cause initial L-sequence 1F 3T to be generated (since this initial L-sequence is not a subsequence of an existing L-sequence that starts with F-element 1F), giving say, L-trace **L5** = 1F 3T ("4" in Table 4.1).

All LCSAJs have been exercised. That is, LCSAJ coverage has been achieved.

In this example, one of the generated L-traces (i.e., L-trace **L4**) did not cause any unexercised LCSAJs to be exercised, even though at least one non-duplicate L-sequence was generated whenever a decision element was reversed (demonstrating that generating non-duplicate L-sequences does not suffice for exercising at least one unexercised LCSAJ). Clearly, testing resources are wasted whenever this occurs.

The following result is immediate.

Lemma 4.7.1

Theorem 4.6 does not suffice for causing at least one unexercised LCSAJ to be exercised whenever a decision element is reversed. (Thus, LCSAJs are not uniquely characterized by L-sequences).

Proof

Follows from the above discussion. ∎

We now suggest how to efficiently use the results of Theorem 4.5 to cause at least one unexercised LCSAJ to be exercised, whenever a decision element is reversed and the corresponding L-trace determined. (Note that from Theorem 4.2 (Sec. 4.1.1) this implies that at least one non-duplicate L-sequence will be generated whenever a decision element is reversed).

First, we introduce the following term.

Let **L** be an LCSAJ, and **P** a subsequence of an L-sequence **L1** such that whenever **P** is in **L1** it implies that **L** has been exercised. Define **P** to be an **LCSAJ Descriptor of L**[21].

Then, the following result is an immediate consequence of this definition :

Lemma 4.7.2

All LCSAJs can be described in terms of corresponding LCSAJ Descriptors.

Proof

Follows from Lemma 4.1.1 (Sec 4.1.1) and from the definition of an LCSAJ Descriptor. ∎

[21]Intuitively, the start line of an LCSAJ is *indicated by* (See Sec. 3.3.2.2) the first decision element of an LCSAJ Descriptor, and the end line of the same LCSAJ is *indicated by* the last decision element of the same LCSAJ Descriptor.

We illustrate the above definition with the following examples. Consider LCSAJ 5 in Table 4.1 (See **Example 4.2** in this Section). This LCSAJ is exercised whenever decision element 2T (associated with predicate 2 in Fig. 4.1) is a subsequence of an L-sequence, i.e., 2T is an LCSAJ Descriptor for LCSAJ 5. Similarity, 1T 2F is an LCSAJ Descriptor for LCSAJ 2. Now consider LCSAJ 8. This LCSAJ is exercised **if and only if**

i) the predicate at line 4 (in Fig. 4.1) has either a *True* or *False* value; and

ii) the predicate at line 14 (in Fig. 4.1) is *False*.

Thus, 2F 3F and 2T 3F are each LCSAJ Descriptors for LCSAJ 8, demonstrating that LCSAJ Descriptors are not necessarily unique.

Next, we very briefly describe how LCSAJ Descriptors, together with the results of Theorem 4.5, can be used to ensure that for every decision element reversed, at least one unexercised LCSAJ is exercised.

First, LCSAJs and their corresponding LCSAJ Descriptors are recorded in a data structure called an **LCSAJ coverage matrix.** An example of an LCSAJ coverage matrix is shown in Table 4.2 (For now, ignore the last column in Table 4.2). The program in Fig. 4.1 corresponds to this LCSAJ-Coverage matrix.

LCSAJ	Start line	End line	Target line	LCSAJ Descriptor	Exercised
1	1	2	10	1F	2
2	1	4	6	1T 2F	3
3	1	5	9	1T 2T	1
4	6	9	13	2F	3
5	9	9	13	2T	1
6	10	14	16	1F 3F	2
7	10	15	19	1F 3T	4
8	13	14	16	2F 3F/2T 3F	3
9	13	15	19	2F 3T/2F 3T	1
10	16	19	exit	3F	2
11	19	19	exit	3T	1

A/B - either *A* or *B*, where *A* and *B* are LCSAJ Descriptors.

Table 4.2 LCSAJ coverage matrix for the program in Fig. 4.1.

Next, the program (under test) is executed on arbitrary input, and the exercised LCSAJs are recorded in the matrix. A longest (not previously selected) L-sequence is then selected from the corresponding L-trace. Moreover, before an attempt is made to generate an initial L-sequence s that is itself not a subsequence of an existing L-sequence that starts at the same point as s, the LCSAJ coverage matrix is examined to determine if this initial L-sequence is associated with at least one LCSAJ Descriptor corresponding to an unexercised LCSAJ. If no such LCSAJ Descriptor exists then s is not generated (since there is no guarantee that an unexercised LCSAJ will be exercised). Otherwise, at least one unexercised LCSAJ will be exercised when s is generated.

This procedure is then applied (in a left to right direction) to remaining longest (not previously selected) L-sequences until LCSAJ coverage (modulo feasibility) has been achieved, or until all longest (not previously selected) L-sequences have been selected. Thus, the LCSAJ coverage matrix can be used to ensure that at least one unexercised LCSAJ is exercised, whenever a decision element is reversed and the corresponding L-trace determined.

In the next section, we prove that the *left to right Adaptive LCSAJ Test Strategy* can be modified to ensure that at least one unexercised LCSAJ is exercised, whenever a decision element is reversed and the corresponding L-trace determined, and at the same time achieve LCSAJ coverage (modulo feasibility).

4.2 IMPLEMENTATION OF THE LCSAJ COVERAGE DIRECTED APPROACH

We now combine Theorem 4.5 and the LCSAJ coverage matrix to yield an effective *modified Adaptive LCSAJ Test Strategy*. Moreover, we prove that the *modified Adaptive LCSAJ Test Strategy* achieves LCSAJ coverage (modulo feasibility). We begin by defining the following term :

Let **L1** be the **longest L-sequence** in an L-trace **L** that starts with a particular decision element or "*". Then the **reversible L-prefix q** of **L** is the initial portion of **L** to the decision element reversed (inclusive) in **L1**.

Then the *modified Adaptive LCSAJ Test Strategy* is summarized (in the following algorithm) as follows :

i) Create LCSAJ coverage matrix **M** for program **f**, initialize the set **Z** of all generated L-traces to null, and initialize the set **R** of all generated L-sequences to null.

ii) Arbitrarily choose an input **x**. (In practice, this input probably should be derived via a black-box testing strategy).

iii) Execute program **f** on **x** to determine the corresponding L-trace **D**, and record in **M** the LCSAJs exercised.

IF (LCSAJ coverage has not been achieved) **THEN**

iv) Add **D** to **Z**, and add all L-sequences in **D** to **R**.

REPEAT { for all L-traces **T** }

v) Select a (not previously selected) L-trace **T** in **Z**.

REPEAT {for all longest, not previously selected, L-sequences ***L1*** }

vi) Select (in a left to right direction) the longest (not previously selected) L-sequence ***L1*** in **T** that starts with a particular decision element or "*".

{ Determine which decision element to reverse}

IF (more than one occurrence of ***L1*** exists in **T**) **THEN**

vii) Select the occurrence that will cause **shortest** reversible L-prefixes to be generated.

ENDIF

viii) First <-- False

IF (there is no leftmost reversed element in ***L1***) **THEN**

IF (the first decision element is either a T-element) or (the first decision element is an F-element that is not part (last element/first element) of two L-sequences) **THEN**

ix) First <-- True

ELSE {the first decision element is an F-element that is part (last element/first element) of two L-sequences}

x) **X** <-- the first decision element.

ENDIF

ELSE { leftmost reversed decision element exists }

xi) X <-- leftmost decision element.

ENDIF

REPEAT {for all decision elements to be reversed }

xii) noreversal <-- False.

IF (First = True) THEN

xiii) Reverse the first decision element of ***L1***.

xiv) Determine the corresponding initial L-sequence **I**.

xv) **X** <-- first decision element of ***L1***.

xvi) First <-- False.

ELSEIF (there is a decision element to the immediate right of **X**) THEN

xvii) Reverse the decision element to the right of **X**.

xviii) Determine the corresponding initial L-sequence **I**.

xix) **X** <-- the decision element to the (immediate) right of **X**.

ELSE

xx) noreversal <-- True

ENDIF

IF (noreversal = False) THEN

IF (**I** is itself not a subsequence of an L-sequence **p** ∈ **R** that starts at the same point as **I**) THEN

IF (at least one LCSAJ Descriptor associated with an unexercised LCSAJ will be generated when **I** is generated) THEN

xxi) Derive data **y** that will cause the initial portion of ***T*** to the element reversed in ***L1*** (i.e., the corresponding reversible L-prefix **q**) to be generated.

xxii) Execute program **f** on data **y** to determine the corresponding L-trace **Q**, and record in **M** the LCSAJs exercised.

xxiii) Add **Q** to **Z**, and add all generated L-sequences in **Q** to **R**.

ENDIF

ENDIF

ENDIF

UNTIL (noreversal = True) or (LCSAJ coverage has been achieved).

UNTIL (all longest L-sequences (not previously selected) in L-trace **T** have been selected) or (LCSAJ coverage has been achieved).

UNTIL (all L-traces in **Z** have been selected) or (LCSAJ coverage has been achieved).

ENDIF

xxiv) Stop

Very briefly, the strategy achieves LCSAJ coverage (modulo feasibility) by selecting a reversible L-prefix **q** such that **q**, if generated, will cause at least one unexercised LCSAJ to be exercised, and then deriving data that will cause **q** to be generated. An example serves to illustrate :

EXAMPLE 4.3.

Let us apply the *modified Adaptive LCSAJ Test Strategy* to the program in Fig. 4.1 (Sec. 4.1.2). First, the program is executed with (arbitrarily chosen) input, resulting in say, L-trace **L1** = 1T 2T 3T. (A "1" is recorded in the last column of Table 4.2 (Sec. 4.1.2) for each previously unexercised LCSAJ). Now, L-trace **L1** is (by definition) an L-sequence, and therefore is the longest L-sequence in this L-trace.

First, decision element 1T is reversed giving 1F as the corresponding initial L-sequences. Initial L-sequence 1F is itself not a subsequence of an existing L-sequence that

starts with 1F such that 1F is at the beginning of an L-trace. In addition, the LCSAJ coverage matrix indicates that this initial L-sequence, if generated, will cause at least one unexercised LCSAJ to be exercised (because at least one LCSAJ Descriptor associated with an unexercised LCSAJ will be generated). Thus, we derive data that will cause this initial L-sequence to be generated, giving say, L-trace **L2** = 1F 3F ("2" in Table 4.2).

Next, decision element 2F in L-sequence **L1** is reversed giving initial L-sequence 1T 2F. Moreover, this initial L-sequence is itself not a subsequence of an existing L-sequence that starts with T-element 1T. In addition, the LCSAJ-Coverage matrix indicates that this initial L-sequence, if generated, will cause at least one unexercised LCSAJ to be exercised. Thus, we derive data that will cause initial L-sequence 1T 2F to be generated, resulting in say, L-trace **L3** = 1T 2F 3F ("3" in Table 4.2).

At this stage, decision element 3T is reversed giving initial L-sequence 1T 2T 3F. Even though this initial L-sequence is not a subsequence of an existing L-sequence that starts with T-element 1T, it is not generated because it is not associated with at least one LCSAJ Descriptor corresponding to an unexercised LCSAJ.

Next, we apply the strategy to L-traces **L2** and **L3**. Suppose that the latter L-trace is selected, i.e., L-trace **L3** = 1T 2F 3F. This L-trace has two L-sequences, namely 1T 2F and 2F 3F. Suppose we select L-sequence 1T 2F. Then no decision element can be reversed in this L-sequence because decision element 2F is the leftmost reversed decision element[22]. As such, L-sequence 2F 3F is selected. Now, decision element 3F should be reversed (decision element 2F is the leftmost reversed element in this L-sequence), giving initial L-sequence 2F 3T that is itself not a subsequence of an existing L-sequence with the same start point (i.e., F-element 2F). However, this initial L-sequence is not generated since it is not associated with at least one LCSAJ Descriptor corresponding to an unexercised LCSAJ.

[22]Recall that the strategy reverses decision elements to the right of the leftmost reversed decision element of an L-sequence.

This leaves L-trace **L2**. L-trace **L2** = 1F 3F is (by definition) an L-sequence. Moreover, decision element 1F is the leftmost reversed decision element in this L-sequence. Thus, if decision element 3F is reversed, 1F 3T is the corresponding initial L-sequence. In addition, this initial L-sequence 1F 3T is itself not a subsequence of an existing L-sequence that starts with F-element 1F. Since the LCSAJ coverage matrix indicates that initial L-sequence 1F 3T, if generated, will cause at least one unexercised LCSAJ to be exercised, we derive data that will results in say, L-trace **L4** = 1F 3T being generated ("4" in Table 4.2).

All LCSAJs have now been exercised, i.e., LCSAJ coverage has been achieved.

In this example, whenever a decision element was reversed at least one unexercised LCSAJ was exercised, and at least one non-duplicate L-sequence was generated when the corresponding L-trace was determined. This was not the case in **Example 4.2** (Sec. 4.1.2) in which the same example program is used (i.e., Fig. 4.2). In particular, L-trace **L4** (in **Example 4.2**) caused no unexercised LCSAJs to be exercised, even though at least one non-duplicate L-sequence was generated (in **L4**).

We now prove that the modified Adaptive LCSAJ Test Strategy achieves LCSAJ coverage (modulo feasibility).

Theorem 4.7

The *modified Adaptive LCSAJ Test Strategy* achieves LCSAJ coverage modulo feasibility.

Proof

The strategy works by reversing decision elements that will cause at least one unexercised LCSAJ to be exercised. From Theorem 4.2 (Sec. 4.1.1), at least one non-duplicate L-sequence is generated when an unexercised LCSAJ is exercised. By exhaustive enumeration of L-sequences, all L-sequences can be generated. The result then follows. ∎

Thus, we have proved that the *modified Adaptive LCSAJ Test Strategy* achieves LCSAJ coverage modulo feasibility[23]. It is possible that, given the available reversible L-prefixes, certain LCSAJs will not be exercised even though they could have been exercised. This is due to the fact that an initial L-sequence associated with a specific LCSAJ could not be generated because the subpath associated with the corresponding reversible L-prefix is infeasible. Moreover, it is possible that other LCSAJs, associated with initial L-sequences of non-duplicate L-sequences that could have been generated from this initial L-sequence, will not be exercised even though they could have been exercised. This is known as **relative infeasibility**, and is a perceived weakness of this testing strategy since it implies that another testing strategy must be used to derive test data that will cause these LCSAJs to be exercised.

On the other hand, from Theorem 4.2 (Sec. 4.1.1), if we focus our attention on an unexercised LCSAJ and some how succeed in exercising at least one unexercised LCSAJ, we will have generated at least one new (non-duplicate) L-sequence. Then, from this L-sequence it may be possible to reapply the strategy to increase the degree of code coverage (i.e., to increase the number of LCSAJs exercised).

The above concepts are illustrated in the next section. Moreover, we demonstrate the applicability of *modified Adaptive LCSAJ Test Strategy* with a well-known example, the Naur Line Editor [Good75, Prob84].

4.3 A DETAILED EXAMPLE

The program in Fig. 4.2 is a Pascal program that is equivalent to the Naur Line Editor [Good75, Prob84]. (The Pascal programming language is described in [Grog80]). The LCSAJ coverage matrix for this program is shown in Table 4.3 --- the first decision element of an LCSAJ Descriptor that is also at the beginning of an L-trace is *underlined*. (For now, ignore

[23]Recall (Sec 2.3.2) that LCSAJ coverage modulo feasibility means that the strategy does not claim to cause unrealizable LCSAJs to be exercised, since no input exists that will cause such LCSAJs to be exercised.

the last column in Table 4.3). In addition, predicates are labelled numerically by parenthesized super-scripts.

Line

```
1   program naur;
2       const
3         bl    = '+';          (* blank *)
4         nl    = '*';          (* new line character *)
5         et    = '!';          (* end of text character *)
6         maxpos =  3 ;

7       var
8         k, bufpos, fill,
9         count                     : integer;
10        alarm                     : boolean;
11        cw                        : char  ;

12      begin
13        alarm  := false;
14        bufpos := 0;
15        fill   := 0;
16        repeat
17          incharacter( cw );
18          if ( cw = bl ) or ( cw = nl ) or (cw = et )(1) then
19            begin
20              if ( bufpos <> 0 )(2) then
21              begin
22                if (( fill + bufpos ) < maxpos ) and ( fill <> 0 ) (3) then
23                  begin
24                    outcharacter(bl);
25                    fill := fill + 1;
26                  end
27                else
28                  begin
29                    outcharacter(nl);
30                    fill := 0;
31                  end;

32                k := 1;
33                while ( k <= bufpos )(4) do
34                 begin
35                  outcharacter( buffer[k] );
36                  k := k + 1;
37                end;
38              fill := fill + bufpos;
39              bufpos := 0;
40            end;
41      end
42    else
```

```
43      begin
44        if bufpos = maxpos (5) then
45          alarm := true
46        else
47          begin
48            bufpos := bufpos + 1;
49            buffer[ bufpos ] := cw;
50          end;
51      end;
52  until (alarm or ( cw = et ) ); (6)
53 end.
```

Fig. 4.2 A Pascal program equivalent to the Naur Line Editor.

LCSAJ	Start Line	End Line	Target Line	LCSAJ Descriptor	Exercised
1	1	18	42	1F	2
2	1	20	41	1T 2F	1
3	1	22	27	1T 2T 3F	-
4	1	26	32	1T 2T 3T	-
5	16	18	42	6F 1F	1
6	16	20	41	6F 1T 2F	3
7	16	22	27	6F 1T 2T 3F	1
8	16	26	32	6F 1T 2T 3T	1
9	27	33	38	3F 4F	-
10	27	37	33	3F 4T	1
11	33	33	38	*4F	1
12	33	37	33	*4T	2
13	32	33	38	3T 4F	-
14	32	37	33	3T 4T	1
15	38	41	52	4F	1
16	41	41	52	2F	1
17	42	44	46	1F 5F/1F 5F	1
18	42	45	51	1F 5T/1F 5T	4
19	46	52	16	5F 6F	1
20	46	53	exit	5F 6T	-
21	51	52	16	5T 6F	-
22	51	53	exit	5T 6T	4
23	52	52	16	4F 6F/2F 6F	1
24	52	53	exit	4F 6T/2F 6T	1

A/B - *A* or *B*, where *A* and *B* are LCSAJ Descriptors.

Table 4.3 LCSAJ coverage matrix for the program in Fig. 4.2.

Then the *modified Adaptive LCSAJ Test Strategy* achieves LCSAJ coverage as follows[24] :

(0) First the program is executed on arbitrary input, say +b*n+n*u!, resulting in L-trace

L1 = 1T 2F 6F 1F 5F 6F 1T 2T 3F 4T *4F 6F 1F 5F 6F 1T 2T 3T 4T *4F 6F 1F 5F 6F 1T 2T 3F 4T *4F 6F 1F 5F 6F 1T 2T 3T 4T *4F 6T .

(A "1" is recorded in the last column of Table 4.3 for each previously unexercised LCSAJ).

(1) Next the longest L-sequence that starts with decision element 1T, such that 1T is at the beginning of L-trace **L1**, is selected. L-sequence 1T 2F is the only such L-sequence. Decision element 1T is reversed to obtain the corresponding initial L-sequence 1F. This initial L-sequence is itself not a subsequence of an existing L-sequence with the same start point. That is, no L-sequence in **L1** starts with 1F such that 1F is at the beginning of L-trace **L1**. Moreover, the LCSAJ coverage matrix indicates that this initial L-sequence, if generated, will cause at least one unexercised LCSAJ to be exercised (because at least one LCSAJ Descriptor associated with an unexercised LCSAJ will be generated). Thus, we derive data that will cause initial L-sequence 1F to be generated. That is, generate reversible L-prefix 1F. Suppose that uu*k! is used as input, then L-trace

L2 = 1F 5F 6F 1F 5F 6F 1T 2T 3F 4T *4T *4F 6F 1F 5F 6F 1T 2T 3F 4T *4F 6T is generated ("2" in Table 4.3).

(1.1) Next decision element 2F in L-sequence 1T 2F is reversed giving the corresponding initial L-sequence 1T 2T. This initial L-sequence is itself not a subsequence of an existing L-sequence that starts with T-element 1T. Moreover, the LCSAJ coverage matrix indicates that this initial L-sequence, if generated, will cause at least one unexercised LCSAJ to be exercised. Thus, we should generate reversible L-prefix 1T 2T. However the subpath associated with

[24] In order to follow the steps of the strategy, the following labelling convention is used :
(0) - Generate first L-trace;
(1) - First application of strategy to an L-sequence;
(1.1) - Indicates reversal of a decision element in L-sequence from (1); etc..

reversible L-prefix 1T 2T is infeasible[25]. In addition, since 1T is at the beginning of L-trace **L1**, LCSAJs 3 and 4 associated with LCSAJ Descriptors 1T 2T 3F[26] and 1T 2T 3T can never be exercised (A "-" is recorded in the last column of Table 4.3 to indicate that these LCSAJs can never be exercised).

(2) The strategy is next applied to the longest L-sequence (in L-trace **L1**) that starts with decision element 2F. Since L-sequence 2F 6F is the only such L-sequence, then reversing element 6F[27] requires that initial L-sequence 2F 6T be generated. Even though this initial L-sequence is not a subsequence of an existing L-sequence that starts with F-element 2F, it not generated because it is not associated with at least one LCSAJ Descriptor corresponding to an unexercised LCSAJ.

(3) The longest L-sequence that starts with decision element 6F is now selected, i.e. L-sequence 6F 1T 2T 3T 4T. (Since there are two occurrences of this L-sequence in L-trace **L1**, the occurrence that generates shortest reversible L-prefixes is used). Only initial L-sequences 6F 1F and 6F 1T 2T 3F are themselves subsequences of existing L-sequences that start with F-element 6F, implying that initial L-sequences 6F 1T 2F and 6F 1T 2T 3T 4F should be generated.

(3.1) Initial L-sequence 6F 1T 2F requires that reversible L-prefix,

RL1 = 1T 2F 6F 1F 5F 6F 1T 2T 3F 4T *4F 6F 1F 5F 6F 1T 2F,

be generated. However, the subpath associated with this reversible L-prefix is infeasible[28]. On the other hand, the subpath associated with initial L-sequence 6F 1T 2F is feasible (in fact, this subpath is actually exercised in step (**10.1**)). Since reversible L-prefix **RL1** cannot be

---

[25]bufpos is set to zero (line 14); bufpos value does not change and is subsequently tested at line 20 to determine if bufpos ≠ 0. Bufpos must be zero, implying that the subpath corresponding to reversible L-prefix 1T 2T is infeasible.

[26]Recall that 1T means that decision element 1T is also at the beginning of an L-trace.

[27]Decision element 2F is not reversed because it is part (last element/first element) of two L-sequences. The strategy does not reverse the F-element in the latter L-sequence.

[28]bufpos is set to a value > 0 (line48); bufpos value does not change and is subsequently tested at line 20 to determine if bufpos ≠ 0. Bufpos must be > 0, implying that the subpath associated with this reversible L-prefix is infeasible.

generated, and at least one unexercised LCSAJ could have been exercised if this reversible L-prefix could have been generated, this is an example of **relative infeasibility** (defined in Sec. 4.2)).

(3.2) Initial L-sequence 6F 1T 2T 3T 4F requires that reversible L-prefix,

RL2 = 1T 2F 6F 1F 5F 6F 1T 2T 3F 4T *4F 6F 1F 5F 6F 1T 2T 3T 4F,

be generated. However, the subpath associated with initial L-sequence 6F 1T 2T 3T 4F is infeasible[29]. This implies that LCSAJ Descriptor 3T 4F can never be generated. Moreover, since this LCSAJ Descriptor is the **only** LCSAJ Descriptor associated with a particular unexercised LCSAJ, namely LCSAJ 13, no input exists that will cause LCSAJ 13 to be exercised ("-" in Table 4.3).

(4) Next the longest L-sequence that starts with decision element 1F such that F-element 1F is not at the beginning of L-trace **L1**, is selected. That is, L-sequence 1F 5F is selected. (Since there are four occurrences of this L-sequence in L-trace **L1**, the occurrence that generates shortest reversible L-prefixes is used). Reversing element 5F requires that initial L-sequence 1F 5T be generated. This initial L-sequence is itself not a subsequence of an existing L-sequence that starts with 1F. In addition, the LCSAJ coverage matrix indicates that this initial L-sequence, if generated, will cause at least one unexercised LCSAJ to be exercised. Thus, we should generate reversible L-prefix 1F 5F 6F 1F 5T. However, the subpath associated with this reversible L-prefix is infeasible[30].

On the other hand, the subpath associated with initial L-sequence 1F 5T is feasible (in fact, it is actually exercised in step **(10.2)**). Since reversible L-prefix 1F 5F 6F 1F 5T cannot be generated, and at least one unexercised LCSAJ could have been exercised if this reversible L-prefix could have been generated, this is another example of **relative infeasibility.**

---

[29]bufpos > 0 at line 20; bufpos value remains unchanged and is then tested at line 33 to determine if 1 ≤ bufpos. Since 1 ≤ bufpos can never have a *False* outcome, that the subpath associated with reversible L-prefix 6F 1T 2T 3T 4F is infeasible.

[30]Bufpos (at line 44) is not yet equal to maxpos.

(5) The longest L-sequence that starts with decision element 5F in L-trace **L1** is now selected, i.e., L-sequence 5F 6F. (Since there are four occurrences of this L-sequence in L-trace **L1**, the occurrence that generates shortest reversible L-prefixes is used). Reversing 6F requires that initial L-sequence 5F 6T be generated. This initial L-sequence is itself not a subsequence of an existing L-sequence that starts with 5F. In addition, the LCSAJ coverage matrix indicates that this initial L-sequence, if generated, will cause at least one unexercised LCSAJ to be exercised. Thus, we should generate reversible L-prefix **RL3** = 1T 2F 6F 1F 5F 6T.

However it can be shown (details omitted) that the subpath associated initial L-sequence 5F 6T is infeasible, imping that LCSAJ Descriptor 5F 6T can never be generated. Moreover, since this LCSAJ Descriptor is the **only** LCSAJ Descriptor associated with a particular unexercised LCSAJ, namely LCSAJ 20, no input exists that will cause this LCSAJ to be exercised. ("-" in Table 4.3).

(6) Next, L-sequence 3F 4T (in L-trace **L1**) is selected, i.e. the longest L-sequence that starts with decision element 3F. (Since there are two occurrences of this L-sequence in L-trace **L1**, the occurrence that generates shortest reversible L-prefixes is used). Decision element 4T is reversed, giving initial L-sequence 3F 4F. This initial L-sequence is itself not a subsequence of an existing L-sequence that starts with F-element 3F. Moreover, the LCSAJ coverage matrix indicates that this initial L-sequence, if generated, will cause at least one unexercised LCSAJ to be exercised. Thus, we should generate reversible L-sequence,

RL4 = 1T 2F 6F 1F 5F 6F 1T 2T 3F 4F

However, it can be shown (details omitted) that the subpath associated with initial L-sequence 3F 4F is infeasible, implying that LCSAJ Descriptor 3F 4F can never be generated. Since this LCSAJ Descriptor is the **only** LCSAJ Descriptor associated with a particular unexercised LCSAJ, namely LCSAJ 9, no input exists that will cause LCSAJ 9 to be exercised ("-" in Table 4.3).

(7) L-sequence *4F is now selected. (Since there are four occurrences of this L-sequence in L-trace **L1**, the occurrence that generates shortest reversible L-prefixes is used). Decision element 4F is reversed, giving initial L-sequence *4T. However, this initial L-sequence is an L-sequence in an existing L-trace, namely L-trace **L2**. Thus, decision element 4F (in L-sequence *4F) is not reversed.

(8) Next, L-sequence 4F 6T is selected. That is, the longest L-sequence that starts with decision element 4F. L-sequence 4F 6T is the only such L-sequence. Moreover, if decision element 6T is reversed 4F 6F is the corresponding initial L-sequence. However, this initial L-sequence is an L-sequence in L-traces **L1** and **L2**. Thus, decision element 6T (in L-sequence 4F 6T) is not reversed.

All longest (not previously selected) L-sequences in L-trace **L1** have been selected.

(9) We now apply the strategy to L-trace **L2**. The longest L-sequence that starts with 1F such that F-element 1F is also at the beginning of L-trace **L2**, is 1F 5F. Reversal of 5F (element 1F is the leftmost reversed element in this L-sequence) requires that reversible L-prefix 1F 5T be generated. However, it can be shown (details omitted) that the subpath associated with reversible L-prefix 1F 5T is infeasible. Thus, LCSAJ Descriptor 1F 5T can never be generated. However, this LCSAJ Descriptor **is not** the only LCSAJ Descriptor associated with a particular unexercised LCSAJ, namely LCSAJ 18 (the other LCSAJ Descriptor is 1F 5T), implying that it is still possible for LCSAJ 18 to be exercised. (In fact, it is exercised in step **(10.2)**).

We note that all longest (not previously selected) L-sequences in L-trace **L2** have already been selected (in steps **(1)** to **(8)** inclusive).

(10) All longest (not previously selected) L-sequences in existing L-traces, namely L-traces **L1** and **L2** have been selected. Note that some LCSAJs have not yet been exercised. For example, LCSAJs 6, 18, 21 and 22 are not exercised. Of these, it can be shown (details

omitted) that no input exists that will cause LCSAJ 21 to be exercised ("-" in Table 4.3). Moreover, the following inputs suffice to cause LCSAJs 6, 18 and 22 to be exercised:

(10.1) Input a*! generates L-trace

L3 = 1F 5F 6F 1T 2T 3F 4T *4F **6F 1T 2F** 6T. ("3" in Table 4.3).

(10.2) Input AAAA! generates L-trace

L4 = 1F 5F 6F 1F 5F 6F 1F 5F 6F **1F 5T** 6T to be generated ("4" in Table 4.3).

At this stage, note that all LCSAJs that can be exercised, now have actually been exercised. Moreover, initial L-sequences 6F 1T 2F and 1F 5F that could not be generated in steps **(3.1)** and **(4)** respectively, have now been generated in L-traces **L3** and **L4** respectively (highlighted in bold).

We note that 6 of the 24 LCSAJs for the program in Fig.4.2 can never be exercised ("-" 's in Table 4.3). Thus, any attempt to exercise subpaths with these LCSAJ will fail. We made three such attempts, namely in steps **(3.2)**, **(5)**, and **(6)**.

Of the remaining subpaths generated (by the strategy), two are infeasible (steps **(3.1)** and **(4)**), and two are feasible (steps **(0)** and step **(1)**). In addition, 15 of the 18 LCSAJ that can be exercised, were actually exercised when L-traces **L1** and **L2** were generated. The remaining LCSAJs that could be exercised, were exercised after relative infeasibility was taken into account (in step **(10)**).

The Pascal program in the APPENDIX is equivalent to the program in Fig. 4.2. Moreover, the program in the APPENDIX generates L-traces and lists all exercised LCSAJs for the program in Fig. 4.2, whenever the *former* is executed on data. In this example, we used the program in the APPENDIX to generate L-traces as well as to list all exercised LCSAJs for the program in Fig. 4.2.

4.4 SUMMARY

The *left to right Adaptive LCSAJ Test Strategy* was modified in order that at least one unexercised LCSAJ is exercised, whenever a decision element is reversed and the corresponding L-trace is determined. This modification yielded the *modified Adaptive LCSAJ Test Strategy*. In addition, we proved that the *modified Adaptive LCSAJ Test Strategy* achieves LCSAJ coverage (modulo feasibility). Further, we demonstrated the applicability of this strategy with a well-known example, the Naur Line Editor [Good75, Prob84].

In the next and final Chapter, we present our conclusions and give directions for future research.

CHAPTER 5

CONCLUSIONS AND SUGGESTIONS FOR FUTURE RESEARCH

We introduced a new adaptive test strategy for LCSAJs and proved that it achieves LCSAJ coverage (modulo feasibility). This testing strategy, in contrast with existing LCSAJ coverage testing strategies, has no *a priori* test path selection phase. That is, no attempt is made to select a set of paths satisfying a common criteria of code coverage, namely LCSAJ coverage.

Instead, our strategy - an extension of the path prefix testing strategy [Myer86, Prat87] - modifies one previously exercised path to obtain a new subpath (determined by reversible L-prefixes) to be exercised. Moreover, this subpath (if exercised) causes at least one unexercised LCSAJ to be exercised.

We are hoping that the subpaths (associated with reversible L-prefixes) to be exercised will be feasible, since the corresponding paths (associated with L-traces) would have already been exercised. If this happens then the computational effort spent in analyzing paths (associated with L-traces) can be utilized in causing LCSAJs in corresponding subpaths (associated with reversible L-prefixes) to be exercised.

This testing strategy does not *a priori* select unexercised LCSAJs to be exercised since the subpaths (associated with reversible L-prefixes) determine the next unexercised LCSAJ to be exercised. Thus, we avoid many of the problems associated with existing LCSAJ

coverage testing strategies (see Sec. 2.4). As a consequence, our approach is felt to be a more pragmatic approach to LCSAJ coverage testing than currently available.

This LCSAJ coverage testing strategy, however, has one perceived drawback --- relative infeasibility. In particular, if relative infeasibility occurs too frequently, it suggests that our strategy (in practice) offers no significant advantages over other LCSAJ coverage testing strategies. Nonetheless, as can be seen from our example in Sec. 4.3, there appears to be a straight forward approach to overcoming this problem

5.1 FUTURE DIRECTIONS

Our LCSAJ coverage testing strategy appears to be highly effective for small programs. However, it requires an experimental study as to its actual cost benefit, especially with respect to large-scale systems. The degree to which relative infeasibility hampers our approach can then be determined.

Currently, this testing strategy is manually implemented. A software tool is needed to help automate this adaptive testing strategy. Tools that automatically generate LCSAJ coverage matrices, determine which reversible L-prefixes need to be generated, etc., need to be developed.

We should also extend our results by relaxing some of the assumptions in Sec. 3.1. For example, we should include programs with GOTO constructs, other control constructs (e.g. CASE), as well as programs with more than one entry and exit point.. Myers [Myer86] indicates that his adaptive testing strategy for branch coverage is valid for programs containing GOTO constructs. Since our test strategy is an extension of his strategy, we expect similar results.

Finally, adaptive white-box testing strategies should be developed for other criteria of code coverage (e.g. chains of LCSAJs [Wood80]. However, such chains are known to be highly infeasible [Wood80].

REFERENCES

Chow78 T. Chow, "Testing Software Design Modeled by Finite State Machines", *IEEE Trans. Software Eng.*, Vol. SE-4, No. 3, pp. 178-187, 1978.

Clar76 L. Clark, "A System to Generate Test Data and Symbolically Execute Programs", *IEEE Trans. Software Eng.*, Vol. SE-2, No. 3, pp. 215-222, 1976.

Clar85 L. Clark *et al.*, "A Comparison of Data Flow Criteria for Test Data Selection", *8th Int'l Conf. on Software Eng.*, U.S.A., pp. 244-251, 1985.

DeMi78 R. DeMillo, R. Lipton and F. Sayward, "Hints on Test Data Selection : Help for the Practicing Programmer", *IEEE Computer*, Vol. 11, No. 4, pp. 34-41, 1978.

Dura80 J. W. Duran and S. C. Ntafos, " An Evaluation of Random Testing", *IEEE Trans. Software Eng.*, Vol. SE-10, No. 4, pp. 438-444, 1980.

Fair85 R. Fairly, *Software Engineering Concepts*, New York , McGraw-Hill, 1985.

Fost80 K. A. Foster, "Error Sensitive Test Case Analysis (ESTCA)", *IEEE Trans. Software Eng.*, Vol. SE-6, No. 3, pp. 258-264, 1980.

Good75 J. Goodenough and S. Gehart,"Toward a Theory of Test Data Selection", *IEEE Trans. Software Eng.*, Vol. SE-1, No. 2, pp. 156-173, 1975.

Grog80 P. Grogono, *Programming in Pascal*, Reading, Mass., Addision-Wesley, 1980.

Hedl85 D. Hedley and M. A. Hennel, "The Causes and Effects of Infeasible Paths in Computer Programs", *8th Int'l Conf. on Software Eng.*, U.S.A., pp. 259-266, 1985.

Henn84 M. Hennell, D. Hedley and L. J. Riddell, "Assessing a Class of Software Tools", *7th Int'l Conf. on Software Eng.*, pp. 266-277, 1984

Hopc79 J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages and Computation*, Reading, Mass. , Addision-Wesley, 1979.

Howd75 W. E. Howden, "Methodology for the Generation of Program Test Data", *IEEE Trans. on Computers*, Vol. C-24, pp. 554-559, 1975.

Howd77 --------, "Symbolic Testing and the DISSECT Symbolic Evaluation System", *IEEE Trans. Software Eng.*, Vol. SE-3, No. 4, pp 266-278, 1977.

Howd80 --------, "Functional Program Testing", *IEEE Trans. Software Eng.*, Vol. SE-6, No. 2, pp. 162-169, 1980.

Howd86 --------, "A Functional Approach to Program Testing and Analysis", *IEEE Trans. Software Eng.*, Vol. SE-12, No. 10, pp. 997-1005, 1986.

Howd87 ---------, *Functional Program Testing & Analysis*, New York, McGraw-Hill, 1987.

Huan75 J. C. Huang, " An Approach to Program Testing", *ACM Computing Surveys*, Vol. 7, No. 3, pp. 113-128, 1975.

Huan78 --------, "Program Instrumentation and Software Testing", *IEEE Computer* , Vol. 11, No. 4, pp. 34-42, 1978.

Ince87 D. C. Ince, " The Automatic Generation of Test Data", *The Computer Journal*, Vol. 30, No. 1, pp.63-69, 1987.

King76 J. C. King, " Symbolic Execution and Program Testing", *Communications of the ACM*, Vol. 19, No. 17, pp. 385-394, 1976.

Kore87 B. Korel, "The Program Dependence Graph in Static Program Testing", *Inf. Proc. Letters*, Vol. 24, No. 2, pp. 103-108, 1987.

Kund79 S. Kundu, "SETAR - A New Approach to Test Case Generation", *Software Testing Infotech State of the Art Rep.*, pp. 163-186, 1979.

Lask83 J. Laski and B. Korel, " A Data Flow Oriented Program Testing Strategy", *IEEE Trans. Software Eng.*, Vol. SE-9, No. 3, pp. 347-354, 1983.

Mill78 E. Miller and W. Howden, *Tutorial : Software Testing and Validation Techniques*, New York, IEEE, 1978.

Mins67 M. Minsky, *Computation : Finite and Infinite Machines*, Englewood Cliffs, N.J., Prentice-Hall, 1967.

Myer79 G. J. Myers, *The Art of Software Testing*, New York, Wiley, 1979.

Myer86 J. P. Myers, *Software Testing : A New Methodology and a Theory of Complexity*, PhD Thesis, Univ. of Denver, 1986.

Ntaf84 S. Ntafos, "On Required Element Testing", *IEEE Trans. Software Eng.*, Vol. SE-10, No. 6, pp. 795-803, 1984.

Paig77 M. R. Paige, " On Partitioning Program Graphs", *IEEE Trans. Software Eng.*, Vol. SE-3, No. 6, pp. 386-393, 1977.

Prat87 R. E. Prather and J. P. Myers, " The Path Prefix Software Testing Strategy", *IEEE Trans. Software Eng.*, Vol. SE-13, No. 7, pp. 761-765, 1987.

Prob81 R. L. Probert, " PROBE - A Semi-Automated Software Validation System", *Proc. Cdn. Inf. Proc. Soc. Conf.*, 3.4.1-3.4.6,1981.

Prob82a --------, "Optimal insertion of Software Probes in Well-Delimited programs", *IEEE Trans. Software Eng.*, Vol. SE-8, No. 1, pp. 34-42, 1982.

Prob82b --------, "Life-Cycle/Grey-Box Testing", *Congressus Numerantium*, Vol. 34, pp. 87-117, 1982.

Prob84 R. L. Probert and H. Ural, "High-Level Testing and Example Directed Development of Software Specification", *Journal of System and Software*, Vol. 4, pp. 317-325, 1984.

Rapp85 S. Rapps and E. Weyuker, "Selecting Software Test Data Using Data Flow Information", *IEEE Trans. Software Eng.*, SE-11, No. 4, pp. 367-375, 1985.

Sari84 B. Sarikaya and G.v. Bochmann, "Synchronization and Specification Issues in Protocol Testing", *IEEE Trans. on Communications*, Vol. Com.32, No. 4, pp. 389-395, 1984.

Ural88 H. Ural and B. Yang, "A Structural Test Selection Criteria", *Inf. Proc. Letters*, Vol. 28, No. 3, pp. 157-163, 1988.

Weyu84 E. Weyuker, " The Complexity of Data Flow Criteria for Test Data Selection", *Inf. Proc. Letters*, Vol. 19, No. 2, pp. 103-109, 1984.

Whit80 L. J. White and E. Cohen," A Domain Strategy for Computer Program Testing", *IEEE Trans. Software Eng.*, Vol. SE-6, No. 3, pp. 347- 357, 1980.

Wood76 M. Woodward, D. Hedley and M. Hennel, " On Program Analysis", *Inf. Proc. Letters*, Vol. 5, No. 5, pp. 136-140, 1976.

Wood80 --------, "Experience with Path Analysis and Testing of Programs", *IEEE Trans. Software Eng.*, Vol. SE-6, No. 3, pp. 278-286, 1980.

Wood84 M. Woodward, "An Investigation into Program Paths and their Representation", Technical Manuscript.

Yang88 B. Yang, *A Test Selection Strategy based on Input-Output Relation Analysis*, MCS Thesis, Univ. of Ottawa, 1988.

APPENDIX

The Pascal program in this APPENDIX is equivalent to the program in Fig. 4.2. Moreover, the program in this APPENDIX generates L-traces and lists all exercised LCSAJs for the program in Fig. 4.2, whenever the former is executed on data.

```
program naur;

const
  bl     = ' ';   (* blank *)
  nl     = '*';   (* new line character *)
  et     = '!';   (* end of text character *)
  maxpos =  2 ;
  tvalue = 'T';
  fvalue = 'F';


var
  k, bufpos, fill : integer;
  alarm           : boolean;
  cw              : char  ;
  buffer          : array [ 1 .. maxpos ] of char;
  trace, lfile,
  infile, outfile : text;

(*--------------------------*)

procedure incharacter( var cw : char );

(* get input character *)

begin
  read( infile, cw);
end;

(*--------------------------*)
```

```
procedure outcharacter( cw : char );

(* Display character in buffer *)

begin
 if ( cw = nl ) then
  writeln(outfile)
 else
  write(outfile, cw);
end;

(*-------------------------*)

procedure printelement( num  : integer;
                        val  : char );

(* display decision element *)

begin
  write(trace,num:1, val:1, ' ':1);
end;

(*-------------------------*)

procedure printasterisk;


(* ENDWHILE delimiter encountered *)

begin
  write(trace,'*':1);
end;

(*-------------------------*)

procedure printstartpt( num : integer);


(* print LCSAJ start line *)

begin
  write(lfile, ' ( ', num:1,' , ');
end;

(*-------------------------*)
```

```
procedure printendpt( num : integer);

(* print LCSAJ end line *)

begin
  write(lfile, num:1,' , ');
end;

(*--------------------------*)

procedure printjumppt( num : integer);

(* print LCSAJ target line *)

begin
  write(lfile, num:1,' ),  ');
end;

(*--------------------------*)

begin(* of main program *)
  assign(trace,"a:trace.n"); assign(lfile,"lfile.n");
  assign(infile,"a:input.n"); assign(outfile,"outfile.n");
  reset(infile);
  rewrite(trace); rewrite(lfile); rewrite(outfile);
  printstartpt(1);
  alarm  := false;
  bufpos := 0;
  fill   := 0;

  repeat
    incharacter( cw );
    if not( ( cw = bl) or ( cw = nl) or (cw = et )) then
      begin
        printendpt(18);
        printjumppt(42);
      end;

    if ( cw = bl ) or ( cw = nl ) or (cw = et ) then
       begin
         printelement(1, tvalue);
         if not( bufpos <> 0 ) then
           begin
             printendpt(20);
             printjumppt(41);
           end;

         if ( bufpos <> 0 ) then
           begin
             printelement(2, tvalue);
             if not((( fill + bufpos ) < maxpos ) and
               ( fill <> 0)) then
               begin
```

```
          printendpt(22);
          printendpt(27);
        end;

      if (( fill + bufpos ) < maxpos ) and
        ( fill <> 0 ) then
        begin
          printelement(3, tvalue);
          outcharacter(bl);
          fill := fill + 1;
          printendpt(26);
          printjumppt(32);
          printstartpt(32);
        end
      else
        begin
          printelement(3,fvalue);
          printstartpt(27);
          outcharacter(nl);
          fill := 0;
        end;

      k := 1;
      while ( k <= bufpos ) do
        begin
          printelement(4, tvalue);
          outcharacter( buffer[k] );
          k := k + 1;
          printasterisk;
          printendpt(37);
          printjumppt(33);
          printstartpt(33);
        end;
      printelement(4,fvalue);
      printendpt(33);
      printjumppt(38);
      printstartpt(38);
      fill := fill + bufpos;
      bufpos := 0;
    end;
  printendpt(41);
  printjumppt(52);
  printstartpt(52);
 end
else
 begin
  printelement(1, fvalue);
  printstartpt(42);
  if not( (bufpos = maxpos)) then
    begin
      printendpt(44);
      printjumppt(46);
    end;
```

```
      if bufpos = maxpos then
        begin
         printelement(5, tvalue);
         alarm := true;
         printendpt(45);
         printjumppt(51);
        end
      else
        begin
          printelement(5,fvalue);
          printstartpt(46);
          bufpos := bufpos + 1;
          buffer[ bufpos ] := cw;
        end;
    end;
    if not( ( alarm or ( cw = et)) ) then
     begin
      printelement(6, fvalue);
      printendpt(52);
      printjumppt(16);
      printstartpt(16);
     end
    else
      begin
       printelement(6, tvalue);
       printendpt(53);
       printjumppt(-1);        (* -1 denotes the entry point to *)
      end;                     (* the operating system.        *)
 until (alarm or ( cw = et ));
end.
```