

Structural Coverage Metrics

Executive Summary

This paper provides a discussion of structural test coverage metrics, looking at the practicality of their use for real software developments. It concludes that some metrics are unsuitable for real use, and recommends suitable combinations of structural coverage metrics for testing a range of integrity levels. It should be remembered that structural coverage based on control flow analysis is only a good starting point for thorough testing, and that other criteria for designing tests based on data flow and required functionality should be included in an effective testing strategy.

IPL is an independent software house founded in 1979 and based in Bath. The company provides a comprehensive range of software services and also supplies the AdaTEST and Cantata software testing and verification packages. This paper has been developed from investigations made during the analysis of requirements for these packages. IPL was accredited to ISO9001 in 1988, and gained TickIT accreditation in 1991. AdaTEST and Cantata have been produced to these standards.

Copyright

This document is the copyright of IPL Information Processing Ltd. It may not be copied or distributed in any form, in whole or in part, without the prior written consent of IPL.

IPL

Eveleigh House

Grove Street

Bath

BA1 5LR

UK

Phone: +44 (0) 1225 444888

Fax: +44 (0) 1225 444400

email ipl@iplbath.com



Certificate Number FM1589

*Last Update: 06/01/97 12:29
File: COVERAGE.DOC*

1. Introduction

Measurement of structural coverage of code is a means of assessing the thoroughness of testing. There are a number of metrics available for measuring structural coverage, with increasing support from software tools. Such metrics do not constitute testing techniques, but a measure of the effectiveness of testing techniques.

A coverage metric is expressed in terms of a ratio of the metric items executed or evaluated at least once to the total number of metric items. This is usually expressed as a percentage.

$$\text{Coverage} = \frac{\text{items executed at least once}}{\text{total number of items}}$$

There is significant overlap between the benefits of many of the structural coverage metrics. It would be impractical to test against all metrics, so which metrics should be used as part of an effective testing strategy?

This paper describes and discusses a selection of structural coverage metrics which are based on control flow analysis. Other structural coverage metrics, such as those based on data flow analysis, are not considered by this paper. For those interested in other metrics, Beizer [1] and Ntafos [4] describe a good selection.

The reason for limiting this paper to structural coverage metrics, and in particular metrics based on control flow, is that such metrics are most suitable for automated collection and analysis. Automated collection and analysis is considered essential if a metric is to be used in a real software development.

The authors have been involved with the development of a range of packages for the testing and verification of software for a number of years. Support for coverage analysis was a high level requirement, but decisions had to be made as to which metrics should be included. This paper has been developed from investigations made during the analysis of requirements for these packages.

In sections 3 to 9 of this paper, metrics are described and assessed against a number of evaluation criteria (which are specified in section 2). Specific attention is given to the practical use of each metric and the feasibility of achieving 100% coverage.

The investigations conducted were based on static analysis and code reading, so the assessment is mostly qualitative. However, a subjective score has been given for each metric against the evaluation criteria (5=high, 1=low). Simple examples are given to illustrate specific points. Data collected from an investigation of real code (summarised in annex A) is used to support the analysis.

Section 10 summarises conclusions and makes recommendations to enable developers to apply structural coverage metrics in a practical way in real software developments. There are many equivalent names for each structural coverage metric. The names used in this paper are those considered to be most descriptive. Equivalent alternative names are listed annex B. References are given in annex C.

2. Evaluation Criteria

The first evaluation criteria is **automation**. To be of use on a real software development, which may involve tens of thousands or hundreds of thousands of lines of code, a metric must be suitable for automated collection and analysis.

A metric should also be **achievable**. It should be possible and practical to achieve 100% coverage (or very close to 100% coverage) of a metric. Any value less than 100% requires investigation to determine why less than 100% has been achieved.

- (a) If it is the result of a problem in the code, the problem should be fixed and tests run again.
- (b) If it is the result of a problem in the test data, the problem should be fixed and tests run again.
- (c) If it is because 100% coverage is infeasible, then the reasons for infeasibility must be ascertained and justified.

Infeasibility occurs because the semantics of the code constrain the coverage which can be achieved, for example: defensive programming, error handling, constraints of the test environment, or characteristics of the coverage metric. Infeasibility should be the only reason for metric values of less than 100% to be accepted.

When 100% coverage is infeasible, the effort required for investigation and to take appropriate action is important. This will depend on the frequency at which coverage of less than 100% occurs and on how **comprehensible** the metric is. To be **comprehensible** the relationship between a metric, design documentation and code should be simple.

Software has to be retested many times throughout its life. Test data required to achieve 100% coverage therefore has to be **maintainable**. Changes required of test data should not be disproportionate in scale to changes made to the code.

An ideal criteria against which a coverage metric should be assessed is its **effectiveness** at detecting faults in software. To measure the **effectiveness** of each coverage metric would require extensive data collection from software tested using the entire range of coverage metrics. The size of such a data collection would require orders of magnitude more effort than the investigation described in annex A.

As the investigation was based on static analysis and code reading, the actual **effectiveness** of each metric could not be quantified. For the purposes of this paper, **effectiveness** is assumed to be a function of **thoroughness**. The **thoroughness** with which test data designed to fulfil a metric actually exercises the code is assessed. A higher thoroughness score is attributed to metrics which demand more rigorous test data to achieve 100% coverage.

3. Statement Coverage

Statement Coverage = s/S

where:

s = Number of statements executed at least once.

S = Total number of executable statements.

Statement coverage is the simplest structural coverage metric. From a measurement point of view one just keeps track of which statements are executed, then compares this to a list of all executable statements. Statement coverage is therefore suitable for **automation**.

Statement coverage is easily **comprehensible**, with the units of measurement (statements) appearing directly in the code. This makes analysis of incomplete statement coverage a simple task.

It is practical to achieve 100% statement coverage for nearly all code. An investigation of real code (as described in annex A) showed no infeasible statements. 100% statement coverage was **achievable** for all modules analyzed. However, statement coverage is not a very good measure of test **thoroughness**. Consider **the following fragment of code**:

Example 3a

```
1. if CONDITION then
2.     DO_SOMETHING;
3. end if;
4. ANOTHER_STATEMENT;
```

Full statement coverage of example 3a could be achieved with just a single test for which CONDITION evaluated to true. The test would not differentiate between the code given in example 3a and the code given in example 3b.

Example 3b

```
1. null;
2. DO_SOMETHING;
3. null;
4. ANOTHER_STATEMENT;
```

Another criticism of statement coverage, is that test data which achieves 100% statement coverage of source code, will often cover less than 100% coverage of object code instructions. Beizer [1] quantifies this at about 75%.

Test data for statement coverage is **maintainable** by virtue of its simplicity and **comprehensible** relationship to the code.

Automation	5
Achievable	5
Comprehensible	5
Maintainable	5
Thoroughness	1

4. Decision coverage

Decision coverage = d/D

where:

d = Number of decision outcomes evaluated at least once.

D = Total number of decision outcomes.

To achieve 100% decision coverage, each condition controlling branching of the code has to evaluate to both true and false. In example 4a, decision coverage requires two test cases.

Example 4a

```
1. if CONDITION then
2.     DO_SOMETHING;
3. else
4.     DO_SOMETHING_ELSE;
5. end if;
```

Test CONDITION

- 1 True
- 2 False

Not all decision conditions are as simple, decision conditions are also in case or switch statements and in loops. However, this does not present an obstacle to **automation**.

The units of measurement (decision conditions) appear directly in the code, making decision coverage **comprehensible** and investigation of incomplete decision coverage straight forward. An investigation of real code (as described in annex A) showed no infeasible decision outcomes. 100% decision coverage was **achievable** for all modules analyzed.

Test data designed to achieve decision coverage is **maintainable**. Equivalent code to example 4a, shown in example 4b, would not require changes to test data for decision coverage.

Example 4b

```
1. if not CONDITION then
2.     DO_SOMETHING_ELSE;
3. else
4.     DO_SOMETHING;
5. end if;
```

For structured software, 100% decision coverage will necessarily include 100% statement coverage. The weakness of decision coverage becomes apparent when non-trivial conditions are used to control branching. In example 4c, 100% decision coverage could be achieved with two test cases, but without fully testing the condition.

Example 4c

```
1. if A and B then
2.     DO_SOMETHING;
3. else
4.     DO_SOMETHING_ELSE;
5. end if;
```

<u>Test</u>	<u>A</u>	<u>B</u>
1	True	True
2	False	True
	Untested	
	True	False
	False	False

For a compound condition, if two or more combinations of components of the condition could cause a particular branch to be executed, decision coverage will be complete when just one of the combinations has been tested. Yet compound conditions are a frequent source of code bugs.

The **thoroughness** of test data designed to achieve decision coverage is therefore an improvement over statement coverage, but can leave compound conditions untested.

Automation	5
Achievable	5
Comprehensible	5
Maintainable	5
Thoroughness	2

5. LCSAJ Coverage

An LCSAJ is defined as an unbroken linear sequence of statements:

- (a) which begins at either the start of the program or a point to which the control flow may jump,
- (b) which ends at either the end of the program or a point from which the control flow may jump,
- (c) and the point to which a jump is made following the sequence.

Hennell [3] gives a full explanation and some examples to help illustrate the definition of an LCSAJ.

LCSAJ coverage = l/L

where:

l = Number of LCSAJs exercised at least once.

L = Total number of LCSAJs.

LCSAJs depend on the topology of a module's design and not just its semantics, they do not map onto code structures such as branches and loops. LCSAJs are not easily identifiable from design documentation. They can only be identified once code has already been written. LCSAJs are consequently not easily comprehensible.

Automation of LCSAJ coverage is a bit more difficult than automation of decision coverage. However, it is relatively easily achieved.

Small changes to a module can have a significant impact on the LCSAJs and the required test data, leading to a disproportionate effort being spent in maintaining LCSAJ coverage and maintaining test documentation. Unfortunately this dependence cannot be illustrated with trivial examples. In examples 5a LCSAJs are marked as vertical bars.

Example 5a

```

          | | | | | |
          | | |
    | | | | | |
    | | | | |
    | | |
    | |
          |
          | |
          | |
    | |
    | | | | |
    | | | | |
    | | | | |
    | | |
    | | | |
    1.  if A then
    2.    STATEMENT;
    3.  end if;
    4.  if B then
    5.    if C then
    6.      STATEMENT;
    7.    else
    8.      STATEMENT;
    9.    end if;
   10. else
   11.   if D then
   12.     STATEMENT;
   13.   else
   14.     STATEMENT;
   15.   end if;
   16. end if;
   18. if E then
   19.   STATEMENT;
   20. end if;

```

Suppose condition B were to be negated and the two nested 'if-else' constructs were to swap positions in the code. Condition A would then be combined in LCSAJs with condition D, whereas condition E would be combined in LCSAJs with condition C. The code would be effectively the same, but the LCSAJs against which LCSAJ coverage is measured would have changed.

A similar problem occurs with case or switch statements, where LCSAJs lead into the first alternative and lead out of the last alternative, as shown in example 5b.

Example 5b

```

          | | | | | | |
          | | | |
    | | | | | | |
    | | | | | |
    | |
    |
          |
          |
          |
          | |
          | |
    | |
    | | | |
    | | | |
    | | |
    | | |
    1.  if A then
    2.    STATEMENT;
    3.  end if;
    4.  case B
    5.    B1:
    6.      STATEMENT;
    7.    B2:
    8.      STATEMENT;
    9.    B3:
   10.      STATEMENT;
   11.  end case
   12. if C then
   13.   STATEMENT;
   14. end if;

```

To achieve LCSAJ coverage, condition A must be tested both true and false with each branch of the case, whereas condition C need only be tested true and false with the last case and one other case. If the sequence of the case branches were modified, or a default (others) case were appended to the case statement, the LCSAJs against which coverage is measured would again change significantly.

Many minor changes and reorganisations of code result in large changes to the LCSAJs, which will in turn have an impact on the test data required to achieve LCSAJ coverage. Test data for LCSAJ coverage is therefore not easily **maintainable**. Minor reorganisations of code have little or no impact on the test data required by other coverage metrics discussed in this paper.

A large proportion of modules contain infeasible LCSAJs and as a result, achieving 100% LCSAJ coverage for other than very simple modules is frequently not **achievable**. Hedley [2] provides data on some FORTRAN code, with an average of 56 LCSAJs per module, in which 12.5% of LCSAJs were found to be infeasible.

An experimental investigation of code, as described in annex A, with an average of 28 LCSAJs per module, showed 62% of modules to have one or more infeasible LCSAJs. Each LCSAJ which has not been covered has to be analyzed for feasibility. The large amount of analysis required for infeasible LCSAJs is the main reason LCSAJ coverage is not a realistically **achievable** test metric.

Hennell [3] provides evidence that testing with 100% LCSAJ coverage as a target is more effective than 100% decision coverage. Test data designed to achieve 100% LCSAJ coverage is therefore more **thorough** than test data for decision coverage. However, like decision coverage, LCSAJ coverage can be complete when just one of the combinations of a compound condition has been tested (as demonstrated in example 4c).

Automation	4
Achievable	1
Comprehensible	1
Maintainable	2
Thoroughness	3

6. Path Coverage

Path Coverage = p/P

where:

p = Number of paths executed at least once.

P = Total number of paths.

Path coverage looks at complete paths through a program. For example, if a module contains a loop, then there are separate paths through the module for one iteration of the loop, two iterations of the loop, through to n iterations of the loop. The **thoroughness** of test data designed to achieve 100% path coverage is higher than that for decision coverage.

If a module contains more than one loop, then permutations and combinations of paths through the individual loops should be considered. Example 6a shows the first few test cases required for path coverage of a module containing two 'while' loops.

Example 6a

```
1. while A loop
2.     A_STATEMENT;
3. end loop;
4. while B loop
5.     ANOTHER_STATEMENT;
6. end loop;
```

<u>Test</u>	<u>A</u>	<u>B</u>
1	False	False
2	(True, False	False)
3	(True, (True,	False) False)
4	(True, False	True,
		False)

etc.

It can be seen that path coverage for even a simple example can involve a large number of test cases. A tool for **automation** of path coverage would have to contend with a large (possibly infinite) number of paths. Although paths through code are readily identifiable, the sheer number of paths involved prevents path coverage from being **comprehensible** for some code.

As for LCSAJs, it must be considered that some paths are infeasible. Beizer [1], Hedley [2] and Woodward [6] conclude that only a small minority of program paths are feasible. Path coverage is therefore not an **achievable** metric. To make path coverage **achievable** the metric has to be restricted to **feasible path coverage**.

Feasible Path Coverage = f/F

where:

f = Number of paths executed at least once.

F = Total number of feasible paths.

Extracting the complete set of feasible paths from a design or code is not suitable for **automation**. Feasible paths can be identified manually, but a manual identification of feasible paths can never ensure completeness other than for very simple modules. For this reason path coverage was not included in the investigation described in annex A.

Both path coverage and feasible path coverage are not easily **maintainable**. The potential complexity and quantity of paths which have to be tested means that changes to the code may result in large changes to test data.

Automation	1
Achievable	1 (feasible 3)
Comprehensible	2

Maintainable	2 (feasible 1)
Thoroughness	4

7. Condition Operand Coverage

Condition Operand Coverage = c/C

where:

c = Number of condition operand values evaluated at least once.

C = Total number of condition operand values.

Condition operand coverage gives a measure of coverage of the conditions which could cause a branch to be executed. Condition operands can be readily identified from both design and code, with condition operand coverage directly related to the operands. This facilitates **automation** and makes condition operand coverage both **comprehensible** and **maintainable**.

Condition operand coverage improves the **thoroughness** of decision coverage by testing each operand of decision conditions with both true and false values, rather than just the whole condition. However, condition operand coverage is only concerned with condition operands, and does not include loop decisions.

A weakness in the **thoroughness** of condition operand coverage is illustrated by examples 7a and 7b.

In example 7a, 100% condition operand coverage requires test data with both true and false values of operands A and B.

Example 7a

```

1. if A and B then
2.     DO_SOMETHING;
3. else
4.     DO_SOMETHING_ELSE;
5. end if;

```

Example 7b

```

1. FLAG:= A and B;
2. if FLAG then
3.     DO_SOMETHING;
4. else
5.     DO_SOMETHING_ELSE;
6. end if;

```

Condition operand coverage is vulnerable to flags set outside of decision conditions. As a common programming practice is to simplify complex decisions by using Boolean expressions with flags as intermediates, the **thoroughness** of condition operand coverage is therefore not as good as it could be. Equivalent code in example 7b can be tested to 100% condition operand coverage by only testing with true and false values of FLAG, but A or B need not have been tested with both true and false values.

Thoroughness can be improved by including all Boolean expressions into the coverage metric. The term **Boolean expression operand coverage** refers to such a development of condition operand coverage.

Boolean Expression Operand Coverage = e/E

where:

e = Number of Boolean operand values evaluated at least once.

E = Total number of Boolean operand values.

Applying Boolean expression operand coverage to example 7b, in order to achieve 100% coverage, test cases are required in which each of A, B and FLAG have values of true and false.

There were no infeasible operand values in the real code investigated (see annex A). 100% Boolean expression operand coverage was therefore **achievable** for all modules investigated.

Automation	4
Achievable	5
Comprehensible	5
Maintainable	5
Thoroughness	2 (Boolean 3)

8. Condition Operator Coverage

Condition Operator Coverage = o/O

where:

o = Number of condition combinations evaluated at least once.

O = Total number of condition operator input combinations.

Condition operator coverage looks at the various combinations of Boolean operands within a condition. Each Boolean operator (and, or, xor) within a condition has to be evaluated four times, with the operands taking each possible pair of combinations of true and false, as shown in example 8a.

Example 8a

```

1. if A and B then
2.     DO_SOMETHING;
3. end if;

```

<u>Test</u>	<u>A</u>	<u>B</u>
1	True	False
2	True	True
3	False	False

4 False True

As for condition operand coverage, Boolean operators and operands can be readily identified from design and code, facilitating **automation** and making condition operator coverage both **comprehensible** and **maintainable**. However, condition operator coverage becomes more complex and less **comprehensible** for more complicated conditions. **Automation** requires recording of Boolean operand values and the results of Boolean operator evaluations.

As for condition operand coverage, achieving condition operator coverage will not be meaningful if a condition uses a flag set by a previous Boolean expression. Examples 7a and 7b illustrated this point. **Boolean expression operator coverage** improves upon the thoroughness of condition operator coverage by evaluating coverage for all Boolean expressions, not just those within branch conditions.

Boolean Expression Operator Coverage = x/X

where:

x = Number of Boolean operator input combinations evaluated at least once.

X = Total number of Boolean operator input combinations.

The **thoroughness** of Boolean expression operator coverage is higher than for condition operand coverage, in that sub-expressions of all compound conditions will be evaluated both true and false.

The investigation of code, described in annex A, identified two infeasible operand combinations which prevented 100% condition operand coverage being **achievable**. Both of these operand combinations occurred in a single module. The general form of the infeasible combinations is given in example 8b.

Example 8b

```
1. if (VALUE=N1) or (VALUE=N2) then
2.   DO SOMETHING;
3. end if;
```

Test	=N1	=N2
1	True	False
2	False	True
3	False	False
	Infeasible	
	True	True

The infeasible operand combinations were both due to mutually exclusive sub-expressions, which (assuming $N1 \neq N2$) could never both be true at the same time.

Infeasible operand combinations are rare, are readily identifiable during design, and do not depend upon the topology of the code. Boolean expression operator coverage is much more **achievable** than LCSAJ coverage.

Automation 4

Achievable 4

Comprehensible	4
Maintainable	5
Thoroughness	3 (Boolean 4)

9. Boolean Operand Effectiveness Coverage

Boolean Operand Effectiveness Coverage = b/B

where:

b = Number of Boolean operands shown to independently influence the outcome of Boolean expressions.

B = Total number of Boolean operands.

To achieve Boolean operand effectiveness coverage, each Boolean operand must be shown to be able to independently influence the outcome of the overall Boolean expression. The straight forward relationship between test data and the criteria of Boolean operand effectiveness coverage makes the metric **comprehensible** and associated test data **maintainable**. This is illustrated by example 9a.

Example 9a

```
1. if (A and B) or C then
2.     DO_SOMETHING;
3. end if;
```

<u>Test</u>	<u>A</u>	<u>B</u>	<u>C</u>
1	true	true	false
2	false	true	false

(Tests 1 and 2 show independence of A)

3	true	true	false
4	true	false	false

(Tests 3 and 4 show independence of B)

5	false	false	true
6	false	false	false

(Tests 5 and 6 show independence of C)

It is worth noting that there are other sets of test data which could have been used to show the independence of C.

There were no infeasible operand values in the real code investigated (see annex A), and only two infeasible operand combinations, neither of which obstructed the criteria of Boolean operand effectiveness coverage. 100% Boolean operand effectiveness coverage was therefore **achievable** for all modules investigated.

Boolean operand effectiveness coverage is only concerned with the operands and will not always identify expressions which are using an incorrect operator. Research by Boeing

[7],[8] has shown that for single mutations to operators in a Boolean expression, Boolean Operand effectiveness coverage is as thorough as Boolean expression operator coverage, but that it is less thorough for multiple mutations. As multiple mutations are unlikely, we conclude that the **thoroughness** of test data designed to achieve 100% Boolean operand effectiveness coverage is about the same as the **thoroughness** of Boolean expression operator coverage.

Automation of Boolean operand effectiveness coverage requires the state of all Boolean operands in a Boolean expression to be recorded each time the expression is evaluated. The ability of an operator to independently affect the outcome will not necessarily be demonstrated by adjacent evaluations of the expression (as in example 9a).

Automation	3
Achievable	5
Comprehensible	5
Maintainable	5
Thoroughness	4

10. Conclusions

The use of any coverage metric as an aid to program testing is beneficial for software quality. However, some metrics are more **achievable** than others, even with the benefit of tools.

The size of real software developments means that **automation** should be regarded as a prerequisite for the use of any coverage metric. Provided that **automation** is available from a tool, the difficulty that the tool suppliers have encountered in implementing the tool is irrelevant to software developers.

To be practical for use on a real software development, a coverage metric must be **achievable**. Developers should not have to expend effort on justification of large numbers of infeasible metric items.

Not all software developers will have the experience and understanding of coverage metrics to work with metrics which are not **comprehensible**. Effective use of a coverage metric requires that all members of a development team can understand how to use the metric, and do not make mistakes as a result of complexities of the metric.

A large proportion of the life cycle cost of software is expended on maintenance. Even during development, modules of code will usually undergo a number of changes. Each time code is changed, it has to be retested; **maintainability** of test data is consequently a major consideration when selecting a coverage metric.

Table 1 provides a summary of the evaluation criteria and the scores assigned for each coverage metric investigated.

Coverage Metric	Evaluation Criteria				
	Auto- mation	Achieve- able	Comprehen- sible	Maintain- able	Thorough- ness
Statement	5	5	5	5	1
Decision	5	5	5	5	2
LCSAJ	4	1	1	2	3
Path	1	1	2	2	4
Feasible Path	1	3	2	1	4
Condition Operand	4	5	5	5	2
Boolean Expression Operand	4	5	5	5	3
Condition Operator	4	4	4	5	3
Boolean Expression Operator	4	4	4	5	4
Boolean Operand Effectiveness	3	5	5	5	4

Table 1. Summary of Evaluation Criteria

From these criteria it is concluded that LCSAJ coverage and path coverage are not practical metrics for use in real software developments. However, these metrics should not be ruled out of further research. In practice, many testing strategies are based on the identification of feasible paths in support of other metrics, but without actually using path coverage as a metric.

The remaining criteria of **thoroughness** can now be used to rank the metrics. Testing software which requires higher integrity should include coverage metrics of higher **thoroughness**. (The design of test data should also consider data flow and required functionality, as concluded by Weiser [5]).

From this criteria, it is concluded that statement coverage is a valid starting point, but should only be used by itself for software of low integrity. Decision coverage by itself is still not particularly thorough, but it does include some coverage of loops which is not required by any of the other metrics.

Coverage metrics which consider all Boolean expressions should be used in preference to those which only consider conditions. They offer improved thoroughness of test data with no disadvantages.

None of the coverage metrics considered in this paper have been assigned a **thoroughness** score of 5. The conceptual **thoroughness** of 5 is only a target for structural coverage. The structural coverage metrics discussed in this paper should never be used as the sole objective when designing test data. Criteria other than control flow based structural coverage should also be considered when designing test data, such as boundary conditions, data flows and functional requirements.

Annex A Experimental Data

This annex describes an investigation of code using a combination of automated analysis tools and manual analysis.

A total of 77 modules from two projects were analyzed. Project 1 was a real time control system, from which 47 modules comprising a user interface and a communications subsystem were taken. Project 2 was a design tool, from which 30 modules comprising a diagram editor were taken. Both projects used the 'C' language.

Analysis consisted of the identification of branches, LCSAJs and compound conditions (including Boolean expressions outside of branch conditions). Statements, branches, LCSAJs and compound conditions were assessed for feasibility. The data collected is summarised in table 2.

Data Item	Project 1	Project 2	Overall
Number of Modules	47	30	77
Statements			
Minimum	8	14	8
Maximum	81	125	125
Total	1769	1046	2815
Mean/module	38	35	37
Infeasible statements	0	0	0
LCSAJs			
Total	1474	700	2174
Mean/module	31	23	28
Infeasible LCSAJs	126	76	202
Modules with Infeasible LCSAJs	29 (62%)	19 (63%)	48 (62%)
Decisions			
Total	460	207	667
Mean/module	10	7	9
Infeasible decision outcomes	0	0	0
Compound Conditions & Boolean Expressions			
Total	52	41	93
Mean/module	1.11	1.37	1.21
Infeasible operand values	0	0	0
Infeasible operand combinations	0	2	2
Modules with infeasible operand combinations	0 (0%)	1 (3%)	1 (1.3%)

Table 2. Summary of Investigation Data

Annex B Alternative names

There are many equivalent names for each structural coverage metric. The names used in this paper are those considered to be most descriptive. Equivalent alternative names are listed in this annex.

Coverage metrics are sometimes referred to as Test Effectiveness Ratios, abbreviated to TER. This abbreviation appears in a number of the alternative names.

Statement Coverage:

- C1
- TER1
- TER-S

Decision Coverage:

- C2
- Branch Coverage
- TER2
- TER-B

LCSAJ Coverage:

- TER3
- TER-L

Path Coverage:

- TER-P

Condition Operand Coverage:

- Branch Condition Coverage
- BC Coverage

Boolean Expression Operand Coverage:

- Branch Condition and Flag Coverage
- BCF Coverage

Condition Operator Coverage:

- Branch Condition Combination Coverage
- BCC Coverage
- Multiple Condition Coverage

Boolean Expression Operator Coverage:

- Branch Condition Combination and Flag Coverage
- BCCF Coverage

Boolean Operand Effectiveness Coverage:

- Boolean Effectiveness Coverage
- Modified Condition Decision Coverage

Annex C References

- [1] B.Beizer
 "Software Testing Techniques", Second Edition,
 Van Nostrand Reinhold 1990.
- [2] D.Hedley, M.A.Hennell
 "The Causes and Effects of Infeasible Paths in Computer Programs",
 Proceedings 8th International Conference on Software Engineering, IEEE, London
 1985.
- [3] M.A.Hennell, D.Hedley, I.J.Riddell
 "Assessing a Class of Software Tools",
 Proceedings IEEE 7th International Conference on Software Engineering, Orlando,
 pp 266-277, 1984.
- [4] S.C.Ntafos
 "A Comparison of Some Structural testing Strategies",
 IEEE Transactions on Software Engineering, Vol 14, No 6, pp 868-874, June
 1988.
- [5] M.D.Weiser, J.D.Gannon, P.R.McMullin.
 "Comparison of Structural Test Coverage Metrics",
 IEEE Software, Vol 2, No 2, pp 80-85, March 1985.
- [6] M.R.Woodward, D.Hedley, M.A.Hennell
 "Experience with Path Analysis and Testing of Programs",
 IEEE Transactions on Software Engineering, VOL SE-6, No 3, pp 278-286, May
 1980.
- [7] J.J Chilenski, S.P. Miller
 "Applicability of Modified Condition Decision Coverage to Software Testing"
 Boeing Company and Rockwell International Corporation, 1993.
- [8] J..J Chilenski
 Presentation
 Boeing Company, 1995