SMT Open Assessment

MSc Computing 2009

University of York

Author: Y3872644

# Section 1

1. This depends on whether “paths” is considered in terms of basis paths (that is, linearly independent paths) or whether “paths” is interpreted as meaning total paths.

In the case of basis paths, this can be determined by calculating the cyclomatic complexity of the programme, as this figure represents the number of linearly independent paths through the control flow graph (McCabe, 1976). After graphing the control flow () this can be calculated as follows where is the number of edges and is the number of nodes, is the number of connected components (1) and is the number of return statements:

Therefore the number of linearly independent paths is finite (4 paths). Review of the code confirms the following 4 paths (identified by their statement line numbers):

* 1 🡪 2 🡪 3 🡪 4
* 1 🡪 2 🡪 3 🡪 6 🡪 7 🡪 14 🡪 15 🡪 16
* 1 🡪 2 🡪 3 🡪 6 🡪 7 🡪 8 🡪 9 🡪 10 🡪 7 🡪 14 🡪 15 🡪 16
* 1 🡪 2 🡪 3 🡪 6 🡪 7 🡪 8 🡪 9 🡪 10 🡪 11 🡪 7 🡪 14 🡪 15 🡪 16

In the case of total paths however, there are an infinite number. This is because any unit with an indefinite loop contains an infinite number of total paths (one for each iteration of the loop). In this specific case, the loop is indefinite because there is no upper restriction on the values that determine the upper bound of the loop counter and therefore the number of possible execution paths is theoretically unbounded. (Note however, in practice, the range of values that can be specified within the precision of a *double* will provide an upper bound on the path number).

1. The following test set provides 100% statement coverage (that is, each statement of the programme is executed).

 T1 = (a = -1.0, b = 0.0, c = 0.0, d = 0.0, e = 0.0), calc = 0.0

 T2 = (a = 2.0, b = 1.0, c = 0.0, d = 1.0, e = 0.0), calc = 3.1

T1 executes the following path:

 1 🡪 2 🡪 3 🡪 4

T2 executes the following path

 1 🡪 2 🡪 3 🡪 6 🡪 7 🡪 8 🡪 9 🡪 10 🡪 11 🡪 7 🡪 14 🡪 15 🡪 16

This test set does not provide 100% branch coverage. 100% branch coverage requires that each decision has been evaluated to both true and false. In particular, the above test set has only executed the true branch of the decision on line 10 and not the false branch. Therefore only 75% branch coverage is achieved.

START

FINISH

FINISH

Figure 1 - Control flow graph for open assessment code

1. The matrix in Table 1 shows which tests are required to achieve 100% MC/DC coverage for the decision on line 3. All possible combinations of A,C,D,E outcomes are shown then the truth table is extended to show which tests are required to demonstrate A,C,D and E’s independence. By using the table to identify the conditions required to show each variables independence, a minimal test set can be created that achieves 100% MC/DC coverage.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **A** | **C** | **D** | **E** | **Outcome** | **A** | **C** | **D** | **E** |
| **01** | 1 | 1 | 1 | 1 | 1 |  |  |  |  |
| **02** | 1 | 1 | 1 | 0 | 1 |  |  |  |  |
| **03** | 1 | 1 | 0 | 1 | 1 |  |  |  |  |
| **04** | 1 | 1 | 0 | 0 | 1 |  |  |  |  |
| **05** | 1 | 0 | 1 | 1 | 1 |  |  |  |  |
| **06** | 1 | 0 | 1 | 0 | 1 |  |  |  |  |
| **07** | 1 | 0 | 0 | 1 | 1 |  |  |  |  |
| **08** | 1 | 0 | 0 | 0 | 1 | 16 |  |  |  |
| **09** | 0 | 1 | 1 | 1 | 1 |  |  |  |  |
| **10** | 0 | 1 | 1 | 0 | 1 |  |  |  |  |
| **11** | 0 | 1 | 0 | 1 | 1 |  |  |  |  |
| **12** | 0 | 1 | 0 | 0 | 1 |  | 16 |  |  |
| **13** | 0 | 0 | 1 | 1 | 1 |  |  |  |  |
| **14** | 0 | 0 | 1 | 0 | 1 |  |  | 16 |  |
| **15** | 0 | 0 | 0 | 1 | 1 |  |  |  | 16 |
| **16** | 0 | 0 | 0 | 0 | 0 | 8 | 12 | 14 | 15 |

A’s independence shown by:

16, 8

C’s Independence shown by:

16, 12

D’s Independence shown by:

16, 14

E’s Independence shown by:

16, 15

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A** | **C** | **D** | **E** | **Outcome** |
| **01** | 1 | **X** | **X** | **X** | 1 |
| **02** | 0 | 1 | **X** | **X** | 1 |
| **03** | 0 | 0 | 1 | **X** | 1 |
| **04** | 0 | 0 | 0 | 1 | 1 |
| **05** | 0 | 0 | 0 | 0 | 0 |

Table 1 - Truth table showing which combination of outcomes demonstrate condition independence

Note, in practice the effective truth table is greatly reduced because of the programme’s use of the short-circuiting Boolean operator ||. (; an **X** indicates the evaluation was short-circuited).

Table 2 - Reduced truth table

The following test set provides 100% MC/DC coverage. As required by the TCA/DO178B standard, this test set ensures each decision takes on each outcome at least once, each condition in each decision takes on all possible outcomes at least once, each exit point is reached at least once and each condition is shown to independently affect the decision’s outcome.

The first five tests provide 100% coverage for the conditions in the decision on line 3 (the table above shows which conditions are being tested).

T1 = (a = -1.0, b = 1.0, c = 1.0, d = 1.0, e = 10.0), calc = 0.0 //Test 08 in

T2 = (a = 1.0, b = 1.0, c = -1.0, d = 1.0, e = 10.0), calc = 0.0 //Test 12 in

T3 = (a = 1.0, b = 1.0, c = 1.0, d = -1.0, e = 10.0), calc = 0.0 //Test 14 in

T4 = (a = 1.0, b = 1.0, c = 1.0, d = 1.0, e = 11.0), calc = 0.0 //Test 15 in

T5 = (a = 1.0, b = 1.0, c = 1.0, d = 1.0, e = 10.0), calc = 43.0 //Test 16 in

T6 = (a = 12.0, b = 2.0, c = 1.0, d = 1.0, e = 1.0), calc = 51.1

The T6 test case extends the test set so that it provides 100% MC/DC coverage for the conditions in the decisions on lines 7 and 10 as well.

1. (Relevant part of statement is coloured).

Definitions of *e*

* Line 1: double calc(double a, double b, double c, double d, **double e**)
	+ Line 9: **e**=e + (y\*d);
	+ Line 14: **e**=e+((((1.0/(a\*a))+(1/(2\*a\*a)))/2.0)\*d);

Computational uses of *e*

* Line 15: res=4.0 \* **e** \* a;
* Line 9: e=**e** + (y\*d);
* Line 14: e=**e**+((((1.0/(a\*a))+(1/(2\*a\*a)))/2.0)\*d);

Predicate uses of *e*

* Line 3: if(a < 0.00 || c < 0.0 || d < 0.0 || **e > 10.0**){

Kills of *e*

*(Variable is killed when the block goes out of scope)*

* Line 4: **return 0.0**;
* Line 16: **return res**;

Definitions of *d*

* Line 1: double calc(double a, double b, double c, **double d**, double e)

Computational uses of *d*

* Line 9: e=e + (y\***d**);
* Line 14: e=e+((((1.0/(a\*a))+(1/(2\*a\*a)))/2.0)\***d**);
* Line 7: for(x = **d**; x<= a - d; x = x + **d**)

Predicate uses of *d*

* Line 3: if(a < 0.00 || c < 0.0 || **d < 0.0** || e> 10.0){
* Line 7: for(x =d; **x<= a - d**; x = x + d)

Kills of *d*

*(Variable is killed when the block goes out of scope)*

* Line 4: **return 0.0**;
* Line 16: **return res**;
1. The following table (Table 3) lists the simple, definition clear, path segments that need to be exercised for 100% all-uses coverage and 100% all du-path coverage of variable *e*.Four *c*orresponding all-uses test cases that target these path segments are also listed.

|  |  |  |
| --- | --- | --- |
| **Strategy** | **Path segments for variable *e*** | **Corresponding segment test cases** |
| All-Uses | 1 🡪 2 🡪 3 🡪 6 🡪 7 🡪 8 🡪 91 🡪 2 🡪 3 🡪 6 🡪 7 🡪 149 🡪 10 🡪 7 🡪 1414 🡪 15 | T1 = (a = 2.0, b = 0.0, c = 1.0, d = 1.0, e = 1.0), calc = 11.1T2 = (a = 0.0, b = 0.0, c = 1.0, d = 1.0, e = 1.0), calc = NaN[[1]](#footnote-1)T3 = (a = 2.0, b = 0.0, c = 1.0, d = 1.0, e = 1.0), calc = 11.1T4 = (a = 2.0, b = 0.0, c = 1.0, d = 1.0, e = 1.0), calc = 11.1 |
| Alldu-path segments | 1 🡪 2 🡪 3 🡪 6 🡪 7 🡪 8 🡪 91 🡪 2 🡪 3 🡪 6 🡪 7 🡪 149 🡪 10 🡪 7 🡪 149 🡪 10 🡪 11 🡪 7 🡪1414 🡪 15 | Table 3 – A test set that exercises an all-uses strategy for variable *e* but not an all du-paths. |

The test set in Table 3 exercises a complete all-uses strategy of *e* but not an all du-paths. It provides 100% all-uses coverage because it exercises all of the all-uses path segments. It does not achieve 100% du-path coverage because it does not provide coverage of the du-path segment 9 🡪 10 🡪 11 🡪 7 🡪 14.

1. While it is relatively trivial to design test sets that target specific outcomes from more simplistic conditions such as those in the decision on line 3, more complex conditions (especially when part of a nested decision) are much harder to target. For example, if a test case is required that executes the false branch on line 10, it is not immediately obvious which set of inputs would be required produce this result, nor is the feasibility of certain paths obvious. While ‘dry running’ the algorithm, or using symbolic execution techniques can help reveal the necessary starting values, it is clear that in a larger system the issue of determining exactly which path will be exercised by a given set of inputs will be substantially more difficult.

Another related difficulty encountered, was in how to ensure that the proposed test set is minimal (avoids redundancy). Again, for a decision with simple conditions such as that on line 3, this is not a serious difficulty; a table such as that in can be readily produced which shows the number of test cases required for a minimal test set that achieves 100% coverage. However, for decisions with conditions whose outcome is dependent on intermediary calculated variables such as on line 10, avoiding redundancy in the test set is more difficult. This is especially true if the intermediary variables are modified using loops since it is difficult to visualise how a given number of iterations might affect these intermediary variables.

1. Randomised techniques can be successfully applied to white box testing. Consider a key disadvantage of randomised black box testing; the code cannot be inspected therefore it is impossible to know the true code coverage of the testing performed. For example, a programme has the decision “If(x = 6)”; with randomised black box testing, the tester has no knowledge of the values that result in the outcome of this decision evaluating to true or false and so they may execute many test cases but never cause the condition in this decision to evaluate to true; consequently full branch coverage may not be achieved.

This problem is still true even if the generation of random input data is guided by the specification in some way, perhaps for instance by testing with an operational distribution of data as well as an inverted data distribution. While it may be supposed that this provides relatively comprehensive coverage, without access to the code there is no way to verify that this is true.

Randomised white-box testing can help to mitigate this problem as follows. Starting with a test strategy involving the random mutation of well-formed input values (as might be conducted using standard randomised black box testing), testing is performed with the key difference being that access to the code allows structural coverage metrics to be gathered as well.

Once gathered, these metrics could be used to identify paths that were not covered, or are hard to exercise. Then, symbolic execution could be used to inform future test case generation so that coverage is more complete. Clearly, for any non-trivial system this would have to be performed in an automated fashion.

This strategy can also help counter the issue of excessive redundant test cases; a problem with randomised black box testing is that without code coverage information, there is a risk that large sections of the input domain are exercising essentially the same or substantially similar paths. If coverage metrics are gathered, the generated test set can be refined to reduce redundancy.

A difficulty of randomised white-box testing is that it is non-trivial to create an oracle that will work with arbitrary inputs, however this is not a major problem as randomised testing can reveal the presence of some bugs even without an oracle (for example, a programme crash caused by a memory access violation or some other unhandled exception).

# Section 2

1. **Big Bang Testing**: An advantage is that there is no need to produce driver or stub modules when using this method, thereby saving some time and effort over incremental approaches. Using the ‘Big Bang’ method may also be more suitable in this case due to the programme’s small size.

A key disadvantage is that debugging the system is likely to be more time consuming than with other testing methods. For example, if the system is compiled, executed and a fault is then found, there is no way to know which of the 9 modules contains the programming error. Identifying exactly which one is defective may require tracing through up to 9 modules.

Another disadvantage is that even if a programming error is identified in a specific module, after correcting this error the entire programme (modules A – I) must be reloaded into memory. This is more time consuming than compiling and loading a single module (or a subset of the modules).

**Top down Testing**: This approach has the advantage that the programmes executive control (module A) can be tested early on in the development cycle. Also, since the programme has no cycles in its dependency graph, top down testing would be relatively uncomplicated.

This method would require more effort to be expended on the development of stubs which are recognised by Myers, et al (2004) as harder to develop than driver modules. For example, testing module A requires stubs for E, C, G and D to be developed.

Another disadvantage is that by design, the stubs only provide simulated (‘hard-wired’) results to calling modules. It is therefore imperative that the hardcoded results are carefully chosen to provide a realistic test. The limited range or type of results may fail to expose problems that later become visible when the real module is integrated.

**Bottom up Testing**: Using the bottom up testing approach has the advantage that the most used modules (D, I and F) are tested early on. These modules interface with a significant number of other modules and an incremental approach allows any interface errors to be revealed earlier on.

 As with top down testing, this approach allows for potential efficiency gains due to the ability to exploit parallelism in project activities, whereas Big Bang testing allows for none. For instance, testing on any of modules D, I or F could start as soon as implementation has been completed and testing can be performed concurrently with other module testing (even if higher up modules such as module C are not yet fully implemented).

A disadvantage of this approach is that the system as a whole is subjected to the least amount of testing because the upper most modules are added towards the end of the testing process.

In conclusion, the bottom up testing strategy is most suitable for this programme as it allows the most heavily depended on modules to be tested early on and allows greater parallelism within the testing process.

1. **Big** **Bang Testing**: Testing the programme with this strategy requires that all modules A – I are fully implemented. Next, the modules are linked, compiled and executed. This is useful in that problems with module interfaces will be found straight away however the main difficulty here is the lack of rigour, as mentioned previously it will be very difficult to isolate detected defects to specific modules. This problem can be mitigated to some extent by performing unit testing of modules in isolation but as certain defects only manifest themselves when the entire programme is integrated this approach will always be suboptimal this regard. A further difficulty is that no integration testing can be done until all modules are implemented. This is a fundamental limitation of big bang testing and there is no workaround.

**Top down Testing**: Cluster integration begins as follows: starting with the root node of the graph (module A), stub’s representing modules E, D, C and G must be written. There is no single ‘right’ way as to how to order the integration of modules however the starting steps for one possible way is shown in Figure 2 (green colour indicates a stub). Modules would continue to be integrated as soon as higher up modules are completely tested.

Figure 2 - First steps of a top down integration process

In practice this integration order could be modified to take into account issues such as one module not yet being ready for testing or a desire to test more complex modules (e.g. D) earlier on in the process. However a difficulty is encountered due to the amount and type of coupling that exists in the programme (i.e. the programme is not structured in a perfectly top down manner). For instance, writing a stub that simulates the interface and behaviour of D to test A requires also writing stub code to test E and C before they are even integrated themselves.

A potential solution to allow testing of A to begin as soon as possible, is firstly, to implement a subset of stub functionality representing D (just enough to allow testing of A) then to add more functionality later to this stub to allow testing of E and C. This solution has the hazard that by modifying the stub halfway through the integration testing process, there is a risk that previously signed off modules may contain defects that would have been exposed had the full stub been used from the start. Another possible solution is to implement the full stub for D and accept the testing ‘bottleneck’ that this results in; (testing of A is delayed until a full stub representing D is completed that implements functionality for testing E, A and C).

The above mentioned problem also arises when creating a stub for I to test E, since at this point D (which calls I) has not been integrated. This is a well known difficulty in integration testing and finding the optimal order is more commonly referred to as the Class Integration and Test Order Problem (Abdrazik, 2007).

**Bottom Up Testing**: Cluster integration begins as follows: starting with the leaf nodes of the graph, drivers representing higher up modules are written (Figure 3, blue colour indicates a driver module).

Figure 3 - First steps for bottom up integration testing

A driver representing H and D is written to test B, then H is integrated and I is tested using the driver representing D and E. A similar difficulty as described in the top down testing process is encountered; a lack of a perfectly hierarchical structure. This complicates the testing process as it means that, for instance, the driver to test B must incorporate elements of both modules H and D in order to simulate all the necessary interactions.

The exact order of module integration is not completely deterministic and some flexibility is allowed. The only prerequisite to integrating a higher up module is that the subordinate module has had all its interfaces tested.

1. In order to gain confidence that the switching mechanism is bug-free, configuration testing can be performed as follows. Firstly, assuming the configurability requirements for the software are that any combination of modules is permitted, there are 3 modules, each with 2 implementations; this means that the total possible number of combinations is . All possible configuration states are shown in Table 4. Due to the relatively low number of total combinations, it is practical in this case to test the software in all 8 configurations.

The key issue for the testing procedure, is that it verifies as much as possible that the ‘double implementations’ have identical interfaces and have the same logical behaviour; (i.e. the algorithm in B1 should produce identical results as its functional variant B2 given the same set of input parameters). To facilitate this, a test harness could be created to execute a test suite against the 3 modules and check the results against known good values. The rationale being that the test driver executes its tests as the software configuration is varied throughout all 8 possibilities, with the results of calls to the modules’ interfaces checked against an oracle using the test comparator.

In this situation, a defect relating to the switching process would be found when the results from the implementation of 1 module differ from the test oracle (and probably its corresponding paired functional variant), thereby revealing an issue in the module’s functional correctness. It is particular important to use some kind of test oracle to check the logical correctness of a module so as to guard against a situation where the same bug in both functional variants results them ‘confirming’ the same (incorrect) result.

|  |  |  |  |
| --- | --- | --- | --- |
| # | B | F | I |
| 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 2 |
| 3 | 1 | 2 | 1 |
| 4 | 1 | 2 | 2 |
| 5 | 2 | 1 | 1 |
| 6 | 2 | 1 | 2 |
| 7 | 2 | 2 | 1 |
| 8 | 2 | 2 | 2 |

Table 4 - All possible combinations of switchable modules

# Section 3

* 1. **Objectivity**: This is a key personal quality that software reviewers should have. An effective software inspection should be egoless; that is, criticisms should be levelled against the product being inspected rather than against the author. In addition, the author’s purpose is not to defend the software being criticised. This is essential to prevent the review from becoming adversarial in nature, which would degrade its effectiveness.
	2. **Previous inspection experience**: It is important for software reviewers to have previous experience with the exact type of inspection being conducted (e.g. walkthrough, formal inspection etc). Greater experience allows inspectors to work more efficiently as they identify defects more accurately, with lower false positive rates. Cockram (2001) identified the significant impact this attribute has on the quality of reviewers and consequently on inspection effectiveness as a whole.
	3. **Experience of the programming language**: It is crucial that software inspectors have a good deal of familiarity with the programming language used in the software they will be inspecting. The main reason this characteristic is needed is because while some programming languages such as C++, Java and C# share syntactic similarities there are subtle differences. For instance, in Java, array bounds are declared slightly differently than in C++; an inspector with experience of C++ but not Java may incorrectly apply assumptions based on previously held knowledge of a language to the program under inspection.
	4. **Application domain knowledge**: It is important for the reviewer to have some knowledge of the application domain of the system. This is because detecting certain defects at the requirements and design stage requires an understanding of the domain area that comprises the software’s intended use. For example, to provide an effective review of the requirements for a Warehouse Management System (WMS), a reviewer requires understanding of the businesses processes that users perform in a WMS. This allows the reviewer to check that the specified requirements are consistent with their domain knowledge and also that the specification is checked for omissions.
	5. **Communication skills**: Good communication skills are crucial for a software reviewer. This is because some of the greatest benefits of a review depend on the synergistic effects arising from having a group of inspectors meeting to discuss the product, not simply on the skills of individual inspectors. In a sense, the true value of a review is that it represents more than just the sum knowledge and experience of the individual personnel involved. Fagan (1986) referred to this synergistic effect as akin to having an extra “Phantom Inspector” and noted the positive impact it had on inspection effectiveness.

# References

Abdrazik, A. (2007). Using Coupling-Based Weights for the Class Integration and Test Order Problem. *The Computer Journal*. 8 (1), p1.

Cockram, T. (2001). Gaining Confidence in Software Inspection Using a Bayesian Belief Model. *Software Quality Control*. 9 (1), p31-42.

Fagan, M. (1986). Advances in Software Inspections. *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 7, p744-751.

IEEE (2008). IEEE 754-2008 Standard for Floating-Point Arithmetic. p41.

McCabe, T. (1976). A Complexity Measure. *IEEE Transactions on software engineering*. SE-2 (4), p308.

Myers G, et al (2004). The Art of Software Testing, Second Edition. Wiley. p111.

1. The result is NaN because this combination of input values causes the division on line 6 to have a divisor of zero. This is an “invalid operation” (IEEE, 2008) as defined by IEEE 754-2008. [↑](#footnote-ref-1)