Rejoinder to Comments of Reviewer 1

Comment 1: Page 4: Section 1.1.1: Maybe clarify the concepts of error, failure, fault, testing, debugging…?

Response: We have now incorporated the following in Section 1.1.1, Page 4, Paragraph 1.

An error in software is a mistake committed by a programmer. A failure of software occurs during execution of a test case when an unacceptable deviation in the behavior of the software is observed from what is expected. A fault (or defect, or bug) is the cause of a failure of a test case, which may in turn consist of one or more errors. An error in the software can manifest as a fault and in turn can cause a failure. Debugging is carried out to determine the location of the fault in the code causing a failure. Testing is the process of determining if an execution of the software with certain inputs (called test data) causes the software to fail.

Comment 2: Page 5, Figure 1.1: Please give a better explanation of the figure.

Response: The following has been incorporated in Section 1.1.1, Page 5, Paragraph 3 to give a more elaborate explanation of Fig. 1.1.

Figure 1.1 represents a typical testing process [1] using the notations of the UML activity diagram. As shown in the diagram, the different activities of the testing process are design of test cases, running of tests, debugging, and correction of errors. These activities have been drawn using rounded rectangles. A filled circle followed by an arrow indicates the initial activity or the first activity in the process. A transition from one activity to another is dented by an arrow symbol. The following activities are complex and consist of several sub-activities: design of test suite, debugging, and error correction. For example, design of test suite has the subactivities: determining the test strategies to use, designing the test scenarios for each identified test strategy, and then determining the test case for each test scenario. These activities in the diagram are marked with a rake symbol. Presence of a rake symbol in an activity symbol signifies that the corresponding activity is complex and contains subactivities. The rectangles in the diagram denote artifacts. Code, design document, test suite, and corrected program are artifacts and have been denoted using rectangles.

Comment 3: Page 11, Figure 1.2: Same comment as to Figure 1.1.

Response: The following has been incorporated in Page 11, Paragraph 2 to give a more elaborate explanation of Fig. 1.2.

The activity diagram in Figure 1.2 depicts the various activities that take place during test coverage analysis and the order in which they occur. As shown in the diagram, the main steps in the test coverage analysis are code analysis, code instrumentation, and running of test cases, and coverage computation. Each of these activities have been annotated with a rake symbol indicating that these are complex activities and contain several subactivities. The different artifacts have been drawn in rectangles. Execution trace is generated by executing the annotated source code. This is analyzed to produce the coverage report.

Comment 4: Page 19, line 12 from bottom: "if', should be in italics?

Response: The mistake has been corrected in the revised Thesis.

Comment 5: Page 23, Figure 2.4: It is not obvious which node the def-use refer to.

Response: We have incorporated the following in in Section 2.2.2, Page 24, paragraph 3.

The definition-use graph representation, corresponds to the `test_gcd` procedure in Figure 2.1.a, and CFG in Figure 2.1.b. Vertex `4` in the graph, representing statement `y=y-x;` at line `4` of `test_gcd`, is labelled def(`4`)={`y`} for *def* set, and use(`4`)={`x`,`y`} for *use* set.

Comment 6: Page 28, paragraph 2, line 5: "analysis is such...", Should be "in"?

Response: The mistake has been corrected in the revised Thesis.

Comment 7: Page 35, Last sentence of first paragraph in section 2.2.8, last sentence" "Data members..." Please explain.

Response: We have now incorporated the following in the revised Thesis in page 35.

Data dependencies from data members of parameter objects need to be distinguished from that of the data members of the object corresponding to the callsite. Therefore, instead of parameter vertices of SDG, data members have been represented in control dependence of corresponding callsite vertices. Data members of parameter objects, in contrast, have been represented in control dependence of respective parameter vertices.

Comment 8: Page 37, bullet list: is this list complete, or are there other possibilities?

Response: The elements listed correspond to all possibilities including dynamically typed objects at parameter vertices and callsite vertices. Therefore, it is the complete list.

.

Comment 9: Page 37, Paragraph below bullet list: Is the reference to Figure 2.9 correct?

Response: The error is corrected by removing vertex `Fi2` in control dependence of vertex `E13`, from graph in Figure 2.9.

Comment 10: Page 41, Figure 2.10: What is the meaning of the arrows? Explain or discuss the figure.

Response: The following has now been added in Section 2.3.2, Page 43, Paragraph 1, with reference to the Figure 2.10.

An arrow in the figure represents the subsumption relation between a coverage criterion at the left (tail/source) of the arrow to a coverage criterion at the right (head/target) of the arrow. An arrow for example, from the condition/decision coverage to condition coverage in the figure, indicates that condition coverage is subsumed by condition/decision coverage.

Comment 11: Page 47, line 2: This is part of an introduction, make clear here introduction what you mean by "effective" test coverage metrics.

Response: We have now added the following in page 47.

Effectiveness of test coverage metric indicates the extent to which faults detected when full coverage is achieved by the test suite during unit, integration, and system level testing of object-oriented programs.

Comment 12: Page 54, line 2 above section 3.3: "had" -> "have".

Response: The mistake has been corrected in the revised Thesis.

Comment 13: Page 56, line 6 from bottom: "been" -> "be".

Response: The mistake has been corrected in the revised Thesis.

Comment 14: Page 79, line 1: "It can be seen from Figure 4.9 ...". Please explain.

Response: We have now added the following in page 79.

The test case in Figure 4.10.a uses an instance created of an object of `Book` class, and invokes class methods `issueBook()`, `reserveBook()`, and `returnBook()` on the object. The data member `bookState`, is defined and used respectively, with no intervening definition, between pair of statements `C76` and `C78`, `C78` and `C82`, and `C82` and `C84`. These correspond to class level interactions by class methods `Book()` with `reserveBook()`, `reserveBook()` with `issueBook()`, and `issueBook()` with `returnBook()`, for class `Book`, using class data member `bookState` of the class. The corresponding associations of the def/use are represented in the du-pairs of level `0` DUT of the data member `bookState` of class `Book`, shown in Figure 4.2. The exercise of the du-pairs resulting from execution of the test case, is recorded with '`X`' marks at the corresponding cells in the DUT, and is shown in Figure 4.9. The total number of cells in the DUT is `15`, out of which `3` cells could be marked as exercised, based on the coverage achieved by the test case.

Comment 15: Page 83, line 3 from bottom: "If the code and the state model ...": Please explain why.

Response: We have now added the following in Page 85, Paragraph 2.

Our CDC metric is an extension of the state-based metric. The state behavior of classes is represented in an UML state model. It is well known that the events of state transition correspond to invocation of specific class methods, and the state variables correspond to specific data members of the class. These correspondences hold good for a class if the state model of the class in correct, accurate and consistent with the code of the class.

Comment 16: Page 87, Figure 4.12: Box "State representation" has only outgoing arrows. Is that correct?

Response: There was an inadvertent mistake in Figure 4.12, page 89.

Now "Parse source code" activity is connected to "State representation" input data object, with bidirectional arrow.

Comment 17: Page 88, Table 4.1: Text: maybe "considered" -> "analysed"?

Response: The mistake has been corrected in the revised Thesis.

Comment 18: Page 91-92: A better explanation of the figures would be appreciated.

Response: We have now added the following in Section 4.4.3, Page 94.

The differences in the coverage observed in our class level test coverage metrics are shown in Figure 4.13. Figure 4.13.c shows the distribution of differences in coverage observed on the samples using statement coverage with DCC, together with absolute value of the corresponding coverages using the test coverage metrics. The boxplot shows the spread of the difference in coverages observed on the sample programs with increments in test execution at instants of almost equal intervals.

The trend shows that the spread increases rapidly, until the upper limit of the highest quartile touches 100% at about 35% test execution. Narrowing of the spread of the distribution is observed at a slow rate, with increase in test execution. A thin Q1 and Q4, with a large Q3, and smaller Q2 is observed at saturation of the coverages, at the end of the tests. The differences in the test coverage of branch coverage with statement coverage, and DU coverage with statement coverage is shown in Figures 4.13.a, and 4.13.b. A comparison of the observed results reported in Figure 4.13.c, with boxplots in 4.13.a, and 4.13.b, implies that statement coverage saturates at first. Coverage using statement coverage corresponds to most of the samples, in the Q4 part of the boxplots. Saturation of statement coverage compares closely with branch coverage. Comparison with DU coverage shows saturation of the coverages below complete coverage within 45% test execution. Subsequent test execution in Figure 4.13.c shows, increment using class level coverages in DCC. A comparison of statement coverage with coverage observed using SBDC is shown in Figure 4.13.d. The boxplot shows trends similar to 4.13.c. except that coverage using SBDC saturates at around 80% test execution.

Figure 4.14 shows a view of the DCC and SBDC coverages observed on the subjects injected with mutants, of that of Figure 4.13. The spread of the coverage of the subjects in the boxplots, are reported at increments of equal intervals during the course of the test execution. A spread at an instant in average coverage of the test samples in Figure 4.14.a, and 4.14.b, corresponds to one percent increase in test execution. A maximum of about 90% of the injected faults was observed to be detected until saturation of DCC at 72% coverage and at 100% test execution.

A view of the strong measure of coverage computed using DCC and SBDC criteria, at the corresponding test executions is shown in Figure 4.15. Figure 4.15.a shows the distribution of differences in coverage observed on the samples using weak measure and strong measure of DCC. The absolute value of the corresponding coverages is also shown in the plot. Coverage computed using strong metric was observed to be much lower than weak metric by the coverage criteria at every instant in the plot. The distribution of differences in the coverage by SBDC criteria is shown in Figure 4.15.b. An increasing spread in the middle quartiles is observed in both the boxplots. A delay in the saturation was also observed in case of strong test coverage metric. An extent of 46% coverage at saturation was observed using strong measure of DCC criterion by the end of the test execution. Strong SBDC was observed to saturate at an extent of 80%. An intermittent increase in coverage was observed until 90% of test execution. An extent of about 85% of injected faults was detected at the corresponding test execution.

Comment 19: Page 92, paragraph below figure 4.15: A reference to Coburtura tool is missing.

Response: The omission has now been addressed in the revised Thesis.

Comment 20: Page 97, line 6-7, Why is the metric "inadequate"?

Response: We have now included the following sentence in page 99, line 7.

The procedural metrics are inadequate as even with 100% coverage, the metrics could detect at most 45% of the seeded bug.

Comment 21: Page 103, line 6: "had been tested" -> "have been tested".

Response: The mistake has now been corrected in the revised Thesis.

Comment 22: Page 129, line 3 below the heading "Inter-class-Data-dependence-seQuence (IDQC) coverage": What is an inter-class data dependence forest (IDDF)? This is not defined in the list of abbreviations.

Response: The mistake has now been corrected in the revised Thesis, and Section 5.1 now contains definition of Inter-class data dependence forest (IDDF) at page 121.

Comment 23: Page 135 and 137, Figures 5.2 and 5.3: I used some time to understand you discussion in section 2.3.4 with reference to these tables. Maybe this could be better explained?

Response: The following has now been incorporated in page 144.

We have discussed a set of basic data flow coverage criteria, used in procedural test coverage metrics, in Section 2.3.4, under Chapter 2 titled "Basic Concepts". The APDU criterion proposed by Alexander et. al. [66] is an extension of such data flow paths coverage, into criteria for integration test coverage of object-oriented programs. Table 5.2 presents experimental results on criteria defined in our interclass data dependence coverage metrics (IDDC), and experimental results on APDU is reported in Table 5.3.

Rejoinder to Comments of Reviewer 2

Comment 1: The definition of CDT given in section 4.2.1 -- "A CDT is essentially the parse-tree of the MDG" - seems to be loose. It is not clear how a graph (MDG) is parsed to get its CDT. The explanation given in the same paragraph to construct the CDT from MDG looks fine.

Response: The statement "A CDT is essentially the parse-tree of the MDG", in the definition of Section 4.2.1 has been replaced with the following.

A CDT is a connected subgraph of an MDG, consisting of only control dependence edges. A CDT is a spanning tree for the vertex set of the MDG that it includes.

Comment 2: Different control and data dependence coverage criteria are defined in section 4.3. The justification for defining the way it is done for each of those would help understand better.

Response: We have now incorporated the following in Section 4.3, Page 75, Paragraph 2.

A summary to justify the criteria defined in CDC metrics are as follows:

- Data dependence coverage measures the extent to which the data dependences in a class are exercised by the test suite.
- Inter-method dependence coverage (MDC) criterion is based on the extent of coverage of the dependences across methods of a class.
- Interaction of the class methods resulting in exercise of the def-use pairs represented on a DUT are measured by MDC criterion.
- Inter-statement dependence coverage (SDC) criterion is based on the extent of coverage of the dependences among statements belonging to different methods of a class.
- SDC criterion measures the extent of coverage of du-pairs of an extended representation of DUT called statement-level DUT (SDUT).
- MDC for a class (MDCC) and SDC for a class (SDCC) measure the extent to which the du-pairs of the DUTs and SDUTs of a class are exercised by a test suite, respectively.
- Control dependence coverage of a class (CDCC) measures control dependence sequence coverage (CDSC) of every class method, and data dependence coverage of a class (DDCC) measures SDCC respectively.
- Dependence coverage of a class (DCC) measures combined CDCC and DDCC
- State-based dependence coverage of a class (SBDC), measures coverage of a subset of the control and data dependences in DCC that is relevant to specific states of the class.

Comment 3: It is noted that some publicly available Java classes are used for experimentation in section 4.4.2. Whether the research community uses similar kind of classes for experimentation? If not, then why these were chosen for the experiments may be clarified.

Response: The subject programs listed are taken from the sources mentioned in a few related papers.

The paper references and the sources of the subject programs are the following:

- M. J. Harrold and G. Rothermel, "A coherent family of analyzable graphical representations for object-oriented software," Department of Computer and Information Science, The Ohio State University, Technical Report OSU-CISRC-11/96-TR60, Nov 1996.
- W. W. A. Vincenzi, J. Maldonado and M. Delamaro, "Coverage testing of Java programs and components," Science of Computer Programming, vol. 56, no. 1-2, pp. 211–230, 2005.
- Milos Gligoric, "Comparing Non-adequate Test Suites using Coverage Criteria", ISSTA'13, 2013
- C. Zhang. R. Sharma. M. A. Alipour. D. Marinov. M. Gligoric, A. Groce, "Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites," ACM Transactions on Software Engineering and Methodology, vol. 24, no. 4, Aug 2015.
- Chua Hock-Chua, "NTU," http://www.ntu.edu.sg/home/ehchua, 2010, NTU.
- Justin Wetherell's Algorithms: https://code.google.com/archive/p/java-algorithms-implementation/

The toy class Book, is a simple program that has been used to explain the concepts in the chapter.

Comment 4: The experiments which are used in section 4.4.5 for comparison with related works should be explained clearly.

Response: We have now incorporated the following in Section 4.4.5, Page 100, Paragraph 3.

We have used the subject programs in Table 4.1 for experimentation. The DepCov prototype tool discussed in section 4.4.1 was used for instrumentation of the sources, construction of program models, and coverage computations. The mutation operators of μJava tool were used for injecting mutants into the instrumented source code. We have created separate pools of mutated versions of sources using method-level and class-level mutants per subject program. Standard JUnit test cases per subject were selected using a random enumeration on the set of available test cases. Test cases were executed on randomly selected mutated version of the sources. Average coverage across all subjects was measured per criterion. Mutants that are detected are recorded, at every instant of test execution. All mutated versions and test cases are executed. We have selected more mutants from pool of method-level mutants, at first. Selection of class-level mutants was increased later in the test execution.

Comment 5: The issue raised in the second bullet point for chapter 4 is valid in this case also for various integration test coverage metric and may be justified.

Response: We have now incorporated the following in Section 5.2, Page 131, Paragraph 1.

A summary to justify the criteria defined in IDDC metrics are as follows:

- Consequent class entry (CClE) vertex, antecedent call site (ACa) vertex, antecedent method entry (AME) vertex, and context call site (CCa) vertex, are represented in an extended JSysDG (eJSysDG).
- Inter-class control dependence coverage (ICC) criterion measures the extent to which the CDS of the MDGs of AMe vertices are exercised, by the test suite.
- Inter-class data dependence coverage (IDC) criterion measures the extent to which class objects that can bind with callsite and parameter objects using polymorphism, are defined and used, during coverage of ICC, by a test suite.
- Inter-class data dependence (IDD) relation is defined across bounded set of objects, and is represented on inter-class data dependence forest (IDDF).
- Inter-class data dependence sequence coverage (IDQC) criterion extends IDC on set of objects represented on IDDF.
- Inter-class composite system state machine (ICSM) is represented, based on UML state model of concurrent states, eJSysDG, and UML state model of individual classes.
- Inter-class state coverage (ISC), inter-class state transition pair coverage (ITC), and inter-class state transition path coverage (ITPC), criteria are defined based on coverage of the states, transition-pairs, and transition-paths represented on ICSM, respectively by a test suite.

Comment 6: Please clarify whether the multiclass programs considered for experimentation in section 5.3 are real-life or toy programs that are whether these are commercial and used anywhere?

Response: The programs used in the literature have only been considered for experimental studies. These are not commercial software, and are mostly toy applications. We have added this in Section 5.3.1, Page 138, and Paragraph 1.

Comment 7: Experimental results have been compared with four existing works which are relatively old--the latest one being published in 2011. Is there any recent work to compare with? It would be better to do so, if available.

Response: Our proposal for integration test coverage is novel and there is no other published work on integration coverage.

Data dependence-based coverage analysis has not received attention from researchers. The paper which is moderately related is Modeling and coverage analysis of programs with exception handling

ESF Najumudheen, R Mall, D Samanta, "Modeling and coverage analysis of programs with exception handling", Proceedings of the 12th Innovations on Software Engineering Conference, pp. 1-11, ISEC 2019

Though not directly comparable, we have briefly compared it with our approach. The following is included in section 5.3.5, page 142.

Exception handling constructs are not directly included in eJSysDG, therefore exception coverage is not possible to compared using IDDC metrics. This is a potential extension of our work.

Comment 8: The experiments which are used in section 5.3.5 for comparison are not explained and should be done.

Response: We have now included the following in Page 142.

The programs that we have experimented with are shown in Table 5.1. The table shows the characteristics of each programs, including program name, number of use cases and scenarios provided, number of classes that it contained, total number of individual states of the classes, lines of code (LOC), number of JUnit test cases provided, and number of mutants created using the μJava tool. We have used our IDepCov prototype tool, including implementation of our coverage criteria proposed in IDDC, and also implementation of the Cov_PC, STRAP, All-uses of DUT, and APDU coverage criteria from the related work. At first we have created instrumented version of the source codes using our IDepCov tool. Using μJava mutation testing tool we then created the specified number of mutated versions of the instrumented sources. We have executed the standard JUnit test cases, implementing the use case scanarios, on the mutated versions of the corresponding sources to compute coverage achieved by respective criteria, and observe on whether the test case fails, or injected defect is detected (or the mutant is killed). Both these observations are recorded in our results of experimentation, for comparison with related coverage criteria.

We maintained pools of mutated versions of the sources for the respective subjects. We created a random order of the JUnit test cases per subject and picked the test cases in that order to execute on a randomly selected mutated version of the corresponding sources during the course of test execution. We executed every test case provided with a subject at least once on a version of the subject. We executed every version of every subject, and computed average coverage achieved using each coverage criterion, and recorded the number of mutants detected, at the instants of tests, over the entire duration of the test execution.

Equivalent mutants were carefully removed from the test pool, by observing on change in coverage per coverage criterion and per subject. Saturation of coverage by a criterion which could not achieve 100% coverage was inferred, when no change in its coverage was observed over a long time, while change is coverage was observed for another coverage criterion.

Answers to Questions for Viva

Question 1: In your research work, in particular in your experimental studies (sections 4.4 and 5.3.2) you have used a Java framework, and the sample programs that you have analyzed are Java programs. Can you comment on feasibility of your proposed metrics for other programming languages that are more common in safety critical applications, such as Ada, and may be C++?

Response: The results presented in the Thesis are generic and can be applied to any type of programs. However, the results presented in the Thesis are based on JSDG (Java System Dependence Graph). For other languages, suitable dependence model proposed in the literature need to be used.

Question 2: From table 4.1 we see that most of the programs analyzed are relatively small programs. How do you think your proposed metrics will scale up to be practical for testing large scale industrial software development projects?

Response: We have proposed two broad categories of metrics: unit and integration. The unit test metrics are invariant of program size for well-written programs. However, the integration coverage metric may involve significant computations for large programs and a parallel computation of the metrics may be required.

Question 3: In the first part of the abstract in your thesis you refer to safety critical applications and safety standards for developing safety critical software. According to e.g. IEC 61508 part 3, Annex B, table B.1 it is highly recommended (HR), at least for SIL2, SIL3, and SIL4 that use of dynamic objects and dynamic variables should be restricted. a) Will this requirement exclude or restrict the use of certain object-oriented languages for safety critical applications? b) Will 100% test coverage always guarantee a safe software system?

Response:

a) Java , in its current form, is not appropriate as a whole for the development of high integrity systems (SIL2, SIL3, and SIL4) that require rigour and predictability of language, compilation systems, and tools.

However, due to the high popularity of Java, there have been efforts to specify safety-critical subsets of Java such as Safety-Critical Java (SCJ).

b) As is widely accepted, any amount of testing cannot guarantee safety of a system. However, a stronger metric can reduce the bug density. It would be interesting to study how the bug density would decrease with stronger testing.

Question 4: In section 5.3.1 (page 133) you have selected subject programs from a variety of application domains and assert that this is expected to remove any bias in the experimentation. Can you substantiate this assumption?

Response:

Programs for different application domains tend to use specific constructs and contain specific type of bugs. For example, embedded controller software may use large number of decision statements with many atomic clauses in the decision logic. Therefore, the bugs typically concern the decision logic. Similarly, a mathematical application may use large number arithmetic expressions, and consequently exhibit bugs in in arithmetic expressions. By considering programs from different application domains, we expect that the bias towards specific types of bugs would be reduced.

Question 5: In section 6.2 Directions for Future Research, last paragraph (on page 144) you say that "Such an approach could involve design heuristics for assigning weights to edges". Can you mention a few possible ideas?

Response: The dependence graph-based representations for object-oriented program, uses multiple type of edges to represent the various dependences arising in an object-oriented program. These include, call dependences, class dependences, interface dependences, intra-class data and control dependences, class member and data member dependences, and so on. Experimental studies can be conducted to determine the types of bugs in a required class of applications. Depending on the extent of bugs attributable to different dependences, the weights of the dependencies may be fixed.