DOT/FAA/AR-07/20

Air Traffic Organization
Operations Planning
Office of Aviation Research
and Development
Washington, DC 20591

Object-Oriented Technology Verification Phase 3 Report—Structural Coverage at the Source-Code and Object-Code Levels

August 2007

Final Report

This document is available to the U.S. public through the National Technical Information Service (NTIS) Springfield, Virginia 22161.

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

This document is disseminated under the sponsorship of the U.S. Department of Transportation in the interest of information exchange. The United States Government assumes no liability for the contents or use thereof. The United States Government does not endorse products or manufacturers. Trade or manufacturer's names appear herein solely because they are considered essential to the objective of this report. This document does not constitute FAA certification policy. Consult your local FAA aircraft certification office as to its use.

This report is available at the Federal Aviation Administration William J. Hughes Technical Center's Full-Text Technical Reports page: actlibrary.tc.faa.gov in Adobe Acrobat portable document format (PDF).

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT/FAA/AR-07/20		

4. Title and Subtitle	5. Report Date
OBJECT-ORIENTED TECHNOLOGY VERIFICATION PHASE 3 REPORT—STRUCTURAL COVERAGE AT THE SOURCE-CODE AND OBJECT-CODE LEVELS	August 2007
	6. Performing Organization Code
7. Author(s) John Joseph Chilenski and John L. Kurtz	8. Performing Organization Report No.
9. Performing Organization Name and Address The Boeing Company P.O. Box 3707 Seattle, WA 98124-2207	10. Work Unit No. (TRAIS)
	11. Contract or Grant No.
12. Sponsoring Agency Name and Address U.S. Department of Transportation Federal Aviation Administration Air Traffic Organization Operations Planning Office of Aviation Research and Development Washington, DC 20591	13. Type of Report and Period Covered Final Report
	14. Sponsoring Agency Code AIR-120

15. Supplementary Notes

The Federal Aviation Administration Airport and Aircraft Safety R&D Division COTR was Charles Kilgore.

16. Abstract

The Federal Aviation Administration (FAA) sponsored this 3-year, three-phase research to provide information for developing FAA policy and guidance for the use of Object-Oriented Technology in Aviation (OOTiA) systems and to support harmonization with international certification authorities on the use and verification of OOTiA. This Report, (Phase 3), documents the results of an investigation into issues and acceptance criteria for the use of structural coverage analysis (SCA) at the source-code (SC) versus object-code (OC) or executable object-code (EOC) levels within object-oriented technology (OOT) in commercial aviation to satisfy objectives 5-8 of table A-7 in RTCA DO-178B/EUROCAE ED-12B.

The intent of SCA is to provide an objective assessment (measure) of the completeness of the requirements-based tests and supports the demonstration of the absence of unintended function. An analysis of several OOT features and the satisfaction of DO-178B/EUROCAE ED-12B Table A-7 Objective 5 for modified condition decision coverage indicates that either a mix of SC coverage and OC/EOC coverage, or coverage of the OC/EOC with SC to OC/EOC traceability, may be required for software Levels A-C. This differs from the current practice where the coverage analysis is conducted against either the SC or OC/EOC, and SC to OC/EOC traceability is only needed for Level A software.

This research effort identified four issues requiring additional work. Three of these issues, identified in Phase 2, concern whether different levels of coverage for different software levels should be allowed for verification of data coupling and control coupling to satisfy DO-178B/EUROCAE ED-12B Table A-7 Objective 8. The remaining issue, identified in this Phase 3, concerns the proper mix of SC and OC/EOC coverage analyses to satisfy DO-178B/EUROCAE ED-12B Table A-7 Objectives 5-8. The adequate testing of polymorphism with dynamic binding and dispatch came up again during this Phase 3 investigation. Defining adequate testing for this OOT feature is still an active research area with no definitive answer yet.

17. Key Words	18. Distribution Statement
Object-oriented technology, Data coupling, Control coupling, Verification, Structural coverage analysis, Source code, Object code, Executable object code	This document is available to the U.S. public through the National Technical Information Service (NTIS) Springfield, Virginia 22161.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	67	

Form DOT F 1700.7 (8-72) Reproduction of completed page authorized

TABLE OF CONTENTS

		Page
EXECUTIVE SUMMARY		ix
1.	INTRODUCTION	1
1.1	Purpose	1
1.2	Background	1
1.3	Document Overview	2
1.4	Related Activities and Documentation	2
2.	STRUCTURAL COVERAGE AT THE SC AND OC LEVELS	2
2.1	The High-Level View of Coverage Analysis	2
2.2	The OOT Features View of Coverage Analysis	18
2.2.1	Methods Tables	18
2.2.2	Constructors and Initializers	21
2.2.3	Destructors and Finalizers	25
2.2.4	Object-Oriented Technology Features Conclusions	32
3.	MODIFIED CONDITION DECISION COVERAGE AND OBC COMPARISON	33
3.1	One-Condition Decisions	34
3.2	Two-Condition Decisions	34
3.3	Three-Condition Decisions	35
3.4	Beyond Three-Condition Decisions	44
3.5	Modified Condition Decision Coverage Versus OBC Conclusions	46
4.	RESULTS	47
5.	OBJECT-ORIENTED TECHNOLOGY VERIFICATION ISSUES REQUIRING FURTHER WORK	48
6.	REFERENCES	49
APPENDICES		
	A—Fault Injection Example	
	B—Fault Injection With Minimum-Sized Object-Code Branch Coverage Test Sets	

LIST OF FIGURES

Figure		Page
1	Requirements/Implementation Overlap	3
2	Five-Process Life Cycle	4
3	Five-Level Life Cycle Artifacts Overlap	5
4	Five-Level Life Cycle Artifacts Overlap Map	6
5	Methods Tables Within a Class Hierarchy	19
6	Class Methods Table With Offset	19
7	Java Constructor Example	21
8	Copy Constructor Example	23
9	Initializer Constructor Example	24
10	Airplane Class Hierarchy	27
11	C++ Destructor-Compiler Destroying Member Objects	27
12	Finally Block Example	28
13	Finally Example Code	29
14	One-Condition Predicate Graph	35
15	Two-Condition Predicate Graphs	36
16	One With Two-Condition Combinations	37
17	Two With One-Condition Combinations	37
18	Predicate Graph for (A OR B) AND C	38
19	Covering Path Sets for (A OR B) AND C	39
20	Predicate Graph for (A AND B) OR C	39
21	Covering Path Sets for (A AND B) OR C	40

22 Modified Condition Decision Coverage Versus OBC Fault Detection—2(1T,2F) AND <3>(2T,1F) Families 42

23 Modified Condition Decision Coverage Versus OBC Fault Detection—<3>(2T,2F) Family 43

24 Modified Condition Decision Coverage Versus OBC Fault Detection—All Three-Condition Expressions 44

25 Modified Condition Decision Coverage Versus OBC Test Set Size Equivalence in Expressions 45

26 Modified Condition Decision Coverage Versus OBC Test Set Fault Detection Equivalence in Expressions 45

27 Modified Condition Decision Coverage Versus OBC Fault Detection—All Four-Condition Expressions 46

LIST OF TABLES

Table		Page
1	Life Cycle Artifact Overlap Subdomains	7
2	Potential Pair-Wise Subdomain Problems	9
3	Subdomain Resolutions	12
4	Example Bytecodes for Java Constructor—Airplane();	22
5	Example Bytecodes for Java Constructor—Airplane(747);	22
6	Bytecode for the Constructor Public Airplane747();	24
7	Bytecode for the Constructor Public Airplane747(Int);	25
8	Java Bytecodes for Subroutine Branching for Finally Clauses	29
9	Java Bytecodes for the Finally Example	30
10	Alternate Java Bytecodes for the Finally Example	31
11	Comparison of Three-Condition Pattern Families	43

LIST OF ACRONYMS

ASF	Associative shift fault
CAST	Certification Authorities Software Team
DA	Design artifacts
ENF	Expression negation fault
EOC	Executable object code
EOCC	Executable object-code coverage
FAA	Federal Aviation Administration
HLR	High-level requirement
LHS	Left-hand side
LLR	Low-level requirement
MCDC	Modified condition decision coverage
OBC	Object-code branch coverage
OC	Object code
OCC	Object-code coverage
OOT	Object-oriented technology
OOTiA	Object-Oriented Technology in Aviation
ORF	Operator reference fault
RCC	Receiver-classes criterion
RHS	Right-hand side
SC	Source code
SCA	Structural coverage analysis
SCC	Source-code coverage
TMC	Target-methods criterion
VNF	Variable negation fault

EXECUTIVE SUMMARY

Object-oriented technology (OOT) has been used extensively throughout the non-safety-critical software and computer-based systems industry. OOT has also been used in safety-critical medical and automotive systems and is now being used in the commercial airborne software and systems domain. However, as with any new technology, there are concerns and issues relating to its adoption within safety-critical systems. Previous Federal Aviation Administration (FAA) research and two OOT-in-Aviation (OOTiA) workshops with industry indicate that there are some areas of OOT verification that are still a concern in safety-critical systems.

The FAA sponsored this 3-year, three-phase research to provide information for developing FAA policy and guidance for the use of OOTiA systems and to support harmonization with international certification authorities on the use and verification of OOTiA. Phase 1 gathered information on the current state of the industry with respect to the use and verification of OOTiA through an industry survey. Phase 2 addressed the verification of data coupling and control coupling (satisfaction of Objective 8 of RTCA DO-178B/EUROCAE ED-12B Table A-7) in OOT software. Phase 3 (this Report) addresses structural coverage at the source-code (SC) level versus object-code (OC) or executable object-code (EOC) levels (satisfaction of Objectives 5-8 of DO-178B/ED-12B Table A-7) for OOT software.

This report documents the results of an investigation into issues and acceptance criteria for the use of structural coverage analysis (SCA) at the SC level versus OC or EOC levels when using OOT methods and tools in commercial aviation as specified by Objectives 5-7 of Table A-7 in DO-178B/EUROCAE ED-12B. The intent of the SCA is to provide an objective assessment (measure) of the completeness of the requirements-based tests and supports the demonstration of the absence of unintended function.

The results of this research show and this report identifies that certain features of OOT require source-code coverage (SCC) analysis, certain features require OC or EOC coverage analysis, and certain features require a combination of both. A combination of these features indicates that a combination of SCC and OC or EOC coverage analyses are needed for OOT software to satisfy Objectives 5-8 of Table A-7 in DO-178B/EUROCAE ED-12B when the following OOT features are used:

- Method tables
- Constructors and initializers
- Destructors, finalizers and “finally” blocks

This report also shows that, if a combination of SCC analysis and OC or EOC coverage analysis is undesirable from an applicant’s perspective, coverage of the OC or EOC can be used with the addition of SC to OC or EOC traceability. This traceability would need to apply to Levels B and C software, where it currently does not under DO-178B/EUROCAE ED-12B

The previous results, either SCA of both SC and OC/EOC or additional SC to OC or EOC traceability for all three levels requiring SCA (A through C), are changes beyond the DO-178B that needed to meet the intent of SCA for OOT software.

Finally, this Report shows that object-code branch coverage (OBC) of short-circuited logic at the OC or EOC level is not equivalent in the general case to modified condition decision coverage (MCDC) at the SC level. Certain deficiencies in automated object-code coverage (OCC) analyzers contributing to part of the problem are identified. Further analyses to identify these deficiencies are identified. To cover the primary difference between MCDC and OBC requires that the independence of each condition be demonstrated. Note that the results concerning OBC of short-circuited logic apply equally to both OOT and non-OOT software.

OOT issues concerning Objectives 5-8 of Table A-7 in DO-178B/EUROCAE ED-12B requiring further investigation beyond the current task are identified. The majority of these issues were identified in the Phase 2 Report.

1. INTRODUCTION.

1.1 PURPOSE.

This Report documents the results of an investigation into issues and acceptance criteria for the use of structural coverage analysis (SCA) at the source-code (SC) versus object-code (OC) and executable object-code (EOC) levels within object-oriented technology (OOT) in commercial aviation as required by Objectives 5-7 of Table A-7 in RTCA DO-178B/EUROCAE ED-12B (DO-178B hereinafter) [1]. The intent of SCA is to provide an objective assessment (measure) of the completeness of the requirements-based tests and supports the demonstration of the absence of unintended function [1]. This investigation is the third of a three-phase research task undertaken by The Boeing Company on behalf of the Federal Aviation Administration (FAA). The results of the investigation provide information to the FAA for developing policy and guidance for the use of Object-Oriented Technology in Aviation (OOTiA) systems and to support harmonization with international certification authorities on the use and verification of OOTiA. The results also provide guidance for future research tasks.

This investigation was guided, in part, by the results from an industry survey conducted during the first phase of this task [2]. The results concerning current and proposed interpretations and practices for the SCA of software at the SC, OC, and EOC levels to satisfy the objectives of DO-178B [1] with the clarifications given in DO-248B/EUROCAE ED-94B (DO-248B hereinafter) [3] and how those interpretations and practices relate to OOT led to the approach taken in this investigation.

1.2 BACKGROUND.

DO-178B requires SCA in Objectives 5-8 of Table A-7 [1].

OOT has been used extensively throughout the non-safety-critical software and computer-based systems industry. OOT has also been used in safety-critical medical and automotive systems and is now being used in the commercial airborne software and systems domain [2 and 4]. However, as with any new technology, there are concerns and issues relating to its adoption within safety-critical systems. Previous FAA research [2, 4, and 5] and two OOTiA workshops with industry (see http://shemesh.larc.nasa.gov/foot/ for more information) indicate that there are some areas of OOT verification that are still a concern in safety-critical systems [6].

The FAA requested a 3-year, three-phase research task to investigate OOTiA verification. Phase 1 gathered information on the current state of the industry with respect to the use of OOTiA and the current and proposed verification practices for the resulting OOT software through an industry survey [2]. Phase 2 addressed the verification of data coupling and control coupling (satisfaction of Objective 8 of DO-178B Table A-7 [1]) in OOT software [7]. The Phase 1 results provided input to the Phase 3 work on SCA at the SC level versus OC and EOC levels (satisfaction of Objectives 5-7 of DO-178B Table A-7 [1]) for OOT software reported herein.

1.3 DOCUMENT OVERVIEW.

- Section 1 provides the purpose, background, and general overview of this Report.
- Section 2 discusses SCA performed at the SC versus OC and EOC levels from both a high-level perspective as well as a detailed implementation perspective.
- Section 3 discusses modified condition decision coverage (MCDC) at the SC level versus object-code branch coverage (OBC) at the OC or EOC level when only short-circuit logic expressions are used.
- Section 4 summarizes the results of the study.
- Section 5 identifies issues for further study.
- Section 6 provides a list of references used in this Report.
- Section 7 identifies activities and documents related to the work reported herein.

1.4 RELATED ACTIVITIES AND DOCUMENTATION.

There is one related activity and its associated documents that relate directly to the issues addressed herein:

- The joint FAA/NASA Object-Oriented Technology in Aviation project workshops and the associated documentation at http://shemesh.larc.nasa.gov/foot/.

2. STRUCTURAL COVERAGE AT THE SC AND OC LEVELS.

One of the ongoing debates in the commercial airborne software domain is whether SCA should be performed at the SC, OC, or EOC levels. Coverage analysis from a high level is discussed in section 2.1 to determine if SCA on SC, OC, or EOC offers advantages over the others. In section 2.2, coverage analysis is discussed from a detailed level when considering specific OOT features and their implementation within programming languages.

2.1 THE HIGH-LEVEL VIEW OF COVERAGE ANALYSIS.

All processes need an exit criterion assessing the adequacy and completeness of the work performed by that process. Within DO-178B, requirements and structural coverage analyses are used to ensure that the requirements-based testing process adequately exercises a program's functions and structure [1]. Requirements coverage analysis determines which requirements were and were not tested. SCA determines which software structures were and were not exercised and supports the demonstration of the absence of unintended function. The reason that both requirements coverage and structural coverage are needed is graphically depicted in figure 1.

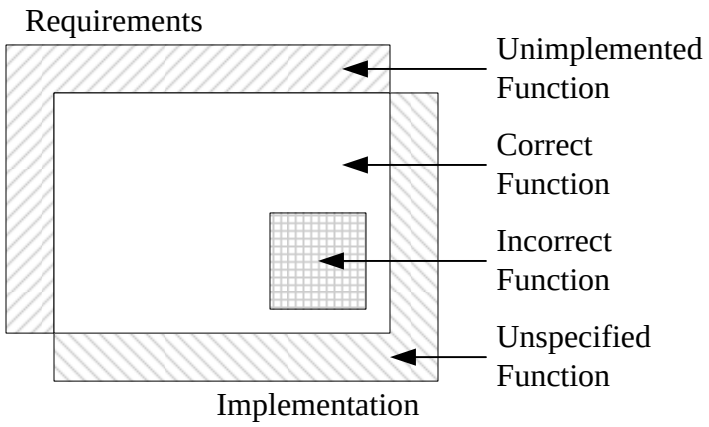


Figure 1. Requirements/Implementation Overlap

In figure 1, the requirements are shown as overlapping the implementation. Where the two overlap, there are parts where the implementation is in agreement with the requirements (i.e., correct) and parts where it is not (i.e., incorrect). Where the requirements do not have an overlap with the implementation is where the implementation fails to use a requirement. Requirements-based test coverage analysis will generally identify these defects (unimplemented function), but SCA generally will not. Where the implementation does not have an overlap with the requirements is where the implementation provides a capability beyond the requirements (unspecified function, possibly unintended). Requirements-based test coverage analysis will generally not identify these defects, but SCA generally will.

To better understand whether SCA is better performed using SC, OC, or EOC, consider how the intermediate life cycle artifacts between the requirements and the implementation fit into an analysis of overlaps as in figure 1. For this analysis, a simplified five-level software-process life cycle model and corresponding artifacts are derived from DO-178B [1]:

- Requirements process—produces the high-level requirements (HLR). Note that the HLR consists of both traceable and derived requirements [1].

- Design process—produces the low-level requirements (LLR) and architecture from the HLR. Note that the LLR consists of both traceable and derived requirements [1]. The LLR and architecture together will hereinafter be referred to as the design artifacts (DA).

- Coding process—produces SC and/or OC from the requirements (HLR, LLR) and architecture. The SC can either be in a high-level language (e.g., Ada, C, C++, Java) or in an assembly language. The SC and OC can either be generated within this life cycle or reused from a library (e.g., commercial-off-the-shelf, operating systems, glue code, microcode). The SC and OC can be generated by either manual or automated means.

- Compile process—produces the OC from the SC. Note that a compiler will generally be used to translate high-level SC into OC, and an assembler will generally be used to translate assembly language SC into OC.

- Link process—produces the EOC from the OC.

The process and artifact flows for this simplified life cycle are depicted in figure 2.

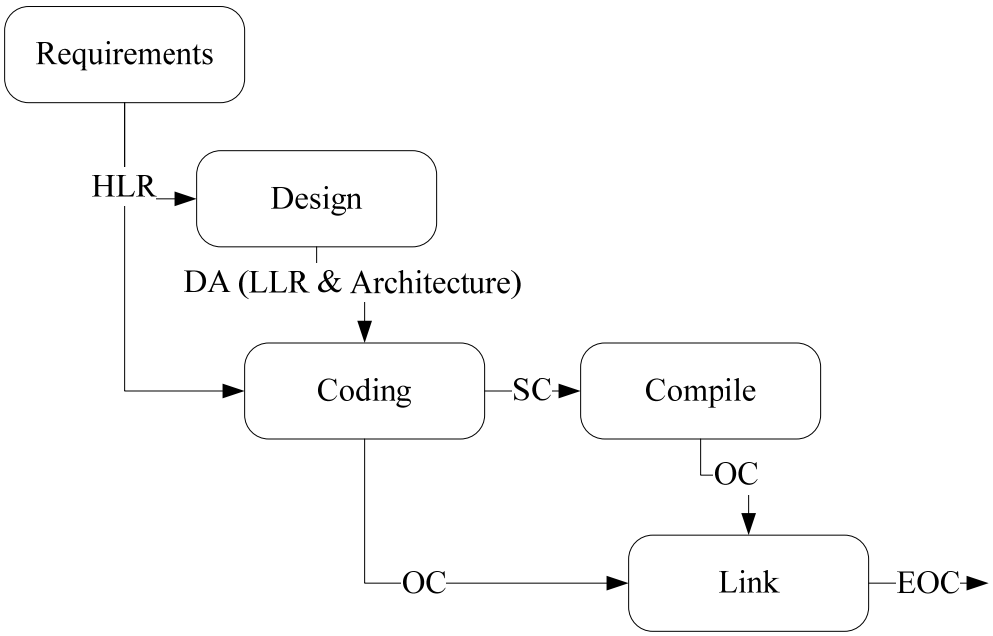


Figure 2. Five-Process Life Cycle

The analysis of the overlap of the five life cycle artifacts (HLR, DA (LLR and architecture), SC, OC, EOC) is depicted in figure 3.

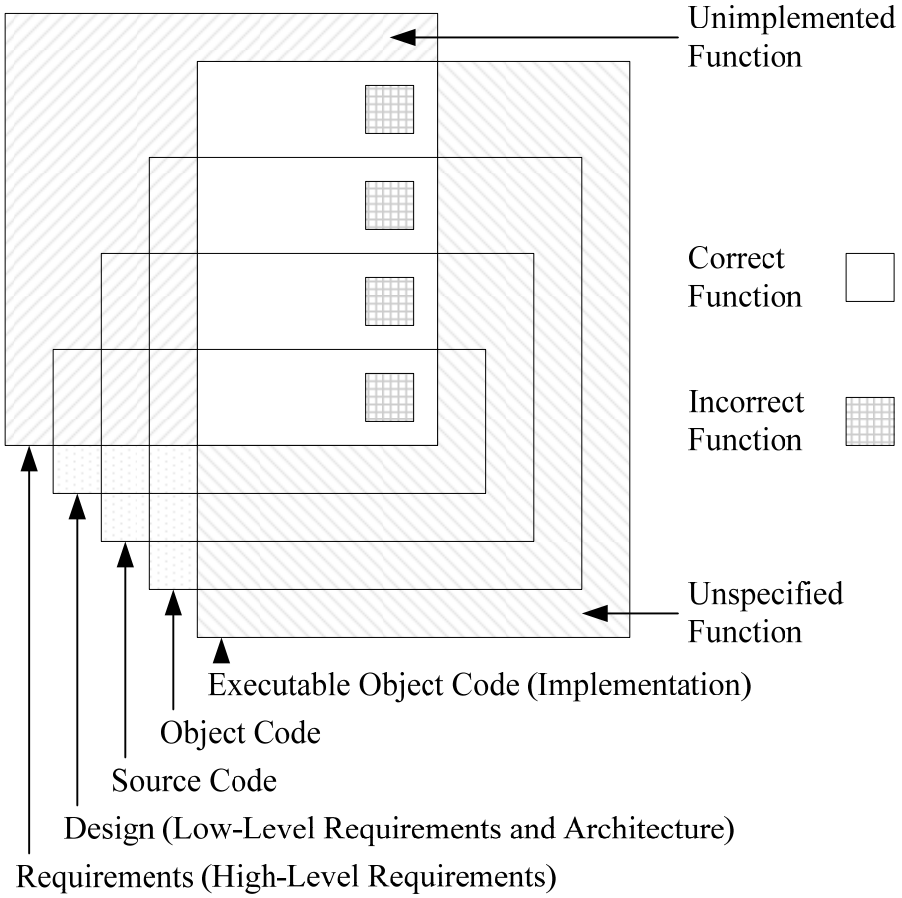


Figure 3. Five-Level Life Cycle Artifacts Overlap

Note that in figure 3 there are more subdomains than in figure 1. The four major subdomains from figure 1 are still present in figure 3 (unimplemented function, correct function, incorrect function, unspecified function), but the addition of the DA, SC, and OC has divided these subdomains further and added some new ones for partially specified and implemented functions. Note that figure 3 is using a simple rectilinear Venn diagram representation, so all overlapping subdomains cannot be represented. The subdomains in figure 3 can be represented by the Venn diagram of figure 4. Because the correctness of an implemented function is not important to the analysis of the advantages of SCA performed at the three levels (SC, OC, and EOC), the implemented function subdomains in figure 4 no longer distinguish between correct and incorrect function and have been collapsed into a single implemented subdomain.

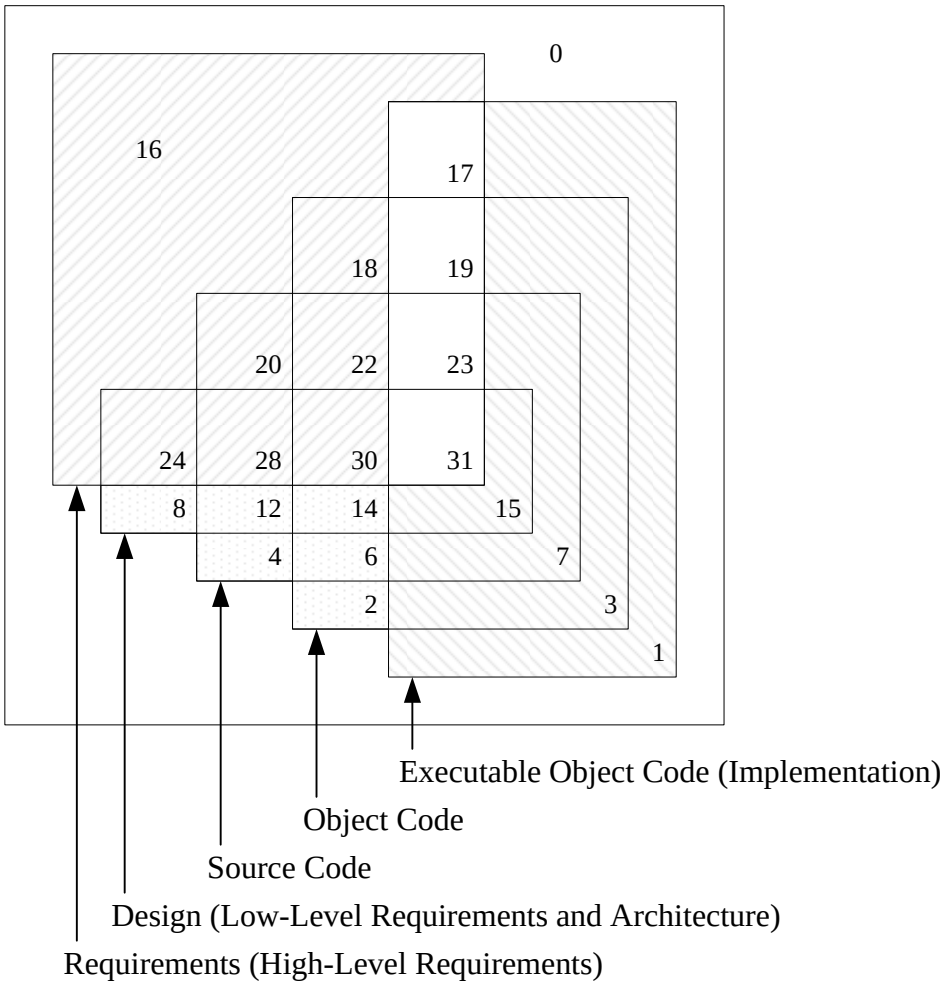


Figure 4. Five-Level Life Cycle Artifacts Overlap Map

The subdomains in figure 4 can be represented in the truth table of table 1. In table 1, the first column identifies the subdomain from figure 4. The second column identifies whether the HLR specify, or ask for, a function. The third column identifies whether the DA (LLR and Architecture) specifies a function. The fourth column identifies whether the SC specifies a function. The fifth column identifies whether the OC specifies a function. The sixth column identifies whether the EOC (implementation) provides a function. The seventh column identifies the major subdomain from figure 4. Note that table 1 has more subdomains identified than figure 4 because the representation technique used in figure 4 is incapable of representing all subdomains.

Table 1. Life Cycle Artifact Overlap Subdomains

Subdomain No.	HLR Specify, or ask for, Function	DA Specifies Function	SC Specifies Function	OC Specifies Function	EOC Provides Function	Major Subdomain
0	False	False	False	False	False	Null
1	False	False	False	False	True	Unspecified
2	False	False	False	True	False	Incomplete
3	False	False	False	True	True	Unspecified
4	False	False	True	False	False	Incomplete
5	False	False	True	False	True	Unspecified
6	False	False	True	True	False	Incomplete
7	False	False	True	True	True	Unspecified
8	False	True	False	False	False	Incomplete
9	False	True	False	False	True	Unspecified
10	False	True	False	True	False	Incomplete
11	False	True	False	True	True	Unspecified
12	False	True	True	False	False	Incomplete
13	False	True	True	False	True	Unspecified
14	False	True	True	True	False	Incomplete
15	False	True	True	True	True	Unspecified
16	True	False	False	False	False	Unimplemented
17	True	False	False	False	True	Implemented
18	True	False	False	True	False	Unimplemented
19	True	False	False	True	True	Implemented
20	True	False	True	False	False	Unimplemented
21	True	False	True	False	True	Implemented
22	True	False	True	True	False	Unimplemented
23	True	False	True	True	True	Implemented
24	True	True	False	False	False	Unimplemented
25	True	True	False	False	True	Implemented
26	True	True	False	True	False	Unimplemented
27	True	True	False	True	True	Implemented
28	True	True	True	False	False	Unimplemented
29	True	True	True	False	True	Implemented
30	True	True	True	True	False	Unimplemented
31	True	True	True	True	True	Implemented

An analysis of potential problems for each of the pair of artifacts corresponding to the subdomains in table 1 and figure 4 is given in table 2. This pair-wise analysis simplifies the subdomain analysis for the subdomains in figure 4, which will follow in table 3. This analysis assumes that all HLR, when present, are correct and necessary but may not be complete. All artifacts are assumed to comply with previous existing artifacts. This means that the potential problems being considered in this simplified analysis are:

- Missing HLR
- Missing DA
- Missing SC
- Missing OC
- Missing EOC
- Extraneous DA
- Extraneous SC
- Extraneous OC
- Extraneous EOC

In table 2, the first column provides a number identifying the pair being considered. The second column identifies whether the HLR specify, or ask for, a function. The third column identifies whether the DA specifies a function. The fourth column identifies whether the SC specifies a function. The fifth column identifies whether the OC specifies a function. The sixth column identifies whether the EOC (implementation) provides a function. The seventh column provides the analysis for the potential problems represented by the artifact pair. In column seven, the potential source of the problem from the simplified life cycle model in figure 2 is identified, along with the potential cause of the problem. In columns 2 through 6, "F" means False, "T" means True, and "-" means "do not care" for the purpose of a pair-wise analysis (i.e., "does not matter").

Since this analysis is based on pairings, three of columns 2 through 6 will always have a "do not care" entry while the other two will have one of the combinations (FF, FT, TF, TT). The (FF) combination means that there is not a potential problem. Nothing earlier in the life cycle was requested, and nothing later in the life cycle was provided. The (TT) combination also means that there is no problem, because something earlier in the life cycle was requested and something later in the life cycle complying with the request was provided. The (FT) combination may mean that there is a potential problem because nothing was requested earlier in the life cycle but something was provided later in the life cycle. Note that there are cases where this is not a potential problem (e.g., a derived LLR will not have an HLR, SC generated directly off of HLR will not have a DA). The (TF) combination may also mean that there is a potential problem because something was requested earlier in the life cycle, but nothing complying with the request was provided later in the life cycle. Note that there are cases where this is not a potential problem (e.g., SC generated directly off of HLR will not have DA).

Table 2. Potential Pair-Wise Subdomain Problems

Pair No.	HLR	DA	SC	OC	EOC	Analysis
1	F	F	–	–	–	Not an issue. Nothing appears in the HLR and no DA were generated.
2	F	T	–	–	–	Potential traceability problem. DA were generated with nothing being asked for in the HLR. Potential requirements process error: missing HLR. Potential design process error: extraneous DA.
3	F	–	F	–	–	Not an issue. Nothing appears in the HLR and no SC was generated.
4	F	–	T	–	–	Potential traceability problem. SC was generated with nothing being asked for in the HLR. Potential requirements process error: missing HLR. Potential coding process error: extraneous SC.
5	F	–	–	F	–	Not an issue. Nothing appears in the HLR and no OC was generated.
6	F	–	–	T	–	Potential traceability problem. OC was generated with nothing being asked for in the HLR. Potential requirements process error: missing HLR. Potential coding process error: extraneous OC. Potential compile process error: extraneous OC.
7	F	–	–	–	F	Not an issue. Nothing appears in the HLR and no EOC was generated.
8	F	–	–	–	T	Potential traceability problem: EOC was generated with nothing being asked for in the HLR. Potential requirements process error: missing HLR. Potential link process error: extraneous EOC.
9	–	F	F	–	–	Not an issue. Nothing appears in the DA and no SC was generated.
10	–	F	T	–	–	Potential traceability problem: SC was generated with nothing being specified in the DA. Potential design process error: missing DA. Potential coding process error: extraneous SC.
11	–	F	–	F	–	Not an issue. Nothing appears in the DA and no OC was generated.
12	–	F	–	T	–	Potential traceability problem: OC was generated with nothing being specified in the DA. Potential design process error: missing DA. Potential coding process error: extraneous OC. Potential compile process error: extraneous OC.
13	–	F	–	–	F	Not an issue. Nothing appears in the DA and no EOC was generated.

Table 2. Potential Pair-Wise Subdomain Problems (Continued)

Pair No.	HLR	DA	SC	OC	EOC	Analysis
14	–	F	–	–	T	Potential traceability problem: EOC was generated with nothing being specified in the DA. Potential design process error: missing DA. Potential link process error: extraneous EOC.
15	–	–	F	F	–	Not an issue. Nothing appears in the SC and no OC was generated.
16	–	–	F	T	–	Potential traceability problem: OC was generated with nothing being specified in the SC. Potential coding process error: missing SC. Potential coding process error: extraneous OC. Potential compile process error: extraneous OC.
17	–	–	F	–	F	Not an issue. Nothing appears in the SC and no EOC was generated.
18	–	–	F	–	T	Potential traceability problem: EOC was generated with nothing being asked for in the SC. Potential coding process error: missing SC. Potential link process error: extraneous EOC.
19	–	–	–	F	F	Not an issue. Nothing appears in the OC and no EOC was generated.
20	–	–	–	F	T	Potential traceability problem: EOC was generated with nothing being asked for in the OC. Potential coding process error: missing OC. Potential compile process error: missing OC. Potential link process error: extraneous EOC.
21	T	F	–	–	–	Potential traceability problem: HLR without complying DA were generated. Potential design process error: missing DA.
22	T	T	–	–	–	Not an issue. HLR with complying DA were generated.
23	T	–	F	–	–	Potential traceability problem: HLR without complying SC was generated. Potential coding process error: missing SC.
24	T	–	T	–	–	Not an issue. HLR with complying SC was generated.
25	T	–	–	F	–	Potential traceability problem: HLR without complying OC was generated. Potential coding process error: missing OC. Potential compile process error: missing OC.
26	T	–	–	T	–	Not an issue. HLR with complying OC was generated.
27	T	–	–	–	F	Potential traceability problem: HLR without complying EOC was generated. Potential link process error: missing EOC.

Table 2. Potential Pair-Wise Subdomain Problems (Continued)

Pair No.	HLR	DA	SC	OC	EOC	Analysis
28	T	–	–	–	T	Not an issue. HLR with complying EOC was generated.
29	–	T	F	–	–	Potential traceability problem: DA without complying SC was generated. Potential design process error: extraneous DA. Potential coding process error: missing SC.
30	–	T	T	–	–	Not an issue. DA with complying SC was generated.
31	–	T	–	F	–	Potential traceability problem: DA without complying OC was generated. Potential design process error: extraneous DA. Potential coding process error: missing OC. Potential compile process error: missing OC.
32	–	T	–	T	–	Not an issue. DA with complying OC was generated.
33	–	T	–	–	F	Potential traceability problem: DA without complying EOC was generated. Potential design process error: extraneous DA. Potential link process error: missing EOC.
34	–	T	–	–	T	Not an issue. DA with complying EOC was generated.
35	–	–	T	F	–	Potential traceability problem: SC without complying OC was generated. Potential coding process error: extraneous SC. Potential compile process error: missing OC.
36	–	–	T	T	–	Not an issue. SC with complying OC was generated.
37	–	–	T	–	F	Potential traceability problem: SC without complying EOC was generated. Potential coding process error: extraneous SC. Potential link process error: missing EOC.
38	–	–	T	–	T	Not a problem. SC with complying EOC was generated.
39	–	–	–	T	F	Potential traceability problem: OC without complying EOC was generated. Potential coding process error: extraneous OC. Potential compile process error: extraneous OC. Potential link process error: missing EOC.
40	–	–	–	T	T	Not a problem. OC with complying EOC was generated.

The 40 potential pair-wise subdomain problems in table 2 applied to the subdomains in table 1 is given in table 3. In table 3, the first column provides a number identifying the subdomain from figure 4 and table 1 being considered. The second column identifies whether the HLR specify, or ask for, a function. The third column identifies whether the DA specifies a function. The fourth column identifies whether the SC specifies a function. The fifth column identifies

whether the OC specifies a function. The sixth column identifies whether the EOC (implementation) provides a function. The seventh column provides the analysis for the subdomain.

The analysis in table 3 is assembled by combining all of the analyses for matching pairs from table 2. For example, for the analysis of subdomain 0, all of the entries in tables 1 and 3 are False (FFFFF). The analyses from table 2 for all of the pair-wise combinations ((FF---), (F-F--), (F--F-), (F---F), (-FF--), (-F-F-), (-F--F), (--FF-), (--F-F), (---FF)) show that there are no potential problems, therefore, the analysis for subdomain 0 in table 3 shows that there is not a problem. For the analysis of subdomain 1, the entries in tables 1 and 3 are (FFFFT). The analyses from table 2 for the pair-wise combinations ((FF---), (F-F--), (F--F-), (-FF--), (-F-F-), (--FF-)) show that there are no potential problems. However, the pair-wise combinations ((F---T), (-F--T), (--F-T), (---FT)) show that there is a potential traceability problem, as well as a potential error in the requirements process leading to missing HLR, a potential error in the design process leading to missing DA, a potential error in the coding process leading to missing SC, a potential error in either the coding or compile process leading to missing OC, and a potential error in the link process leading to extraneous EOC, respectively. Note that, as mentioned previously, none of these potential errors may be actual errors because there may be valid reasons for EOC existing on its own (e.g., deactivated EOC).

Table 3. Subdomain Resolutions

Subdomain No.	HLR	DA	SC	OC	EOC	Analysis
0	F	F	F	F	F	Not an issue. Nothing appears in the HLR, DA, SC, OC, or EOC.
1	F	F	F	F	T	Potential traceability problem: EOC was generated without anything being asked for, or specified, in the HLR, DA, SC or OC. Potential requirements process error: missing HLR. Potential design process error: missing DA. Potential coding process error: missing SC. Potential coding process error: missing OC. Potential compile process error: missing OC. Potential link process error: extraneous EOC.
2	F	F	F	T	F	Potential traceability problem: OC without complying EOC was generated without anything being asked for, or specified, in the HLR, DA, or SC. Potential requirements process error: missing HLR. Potential design process error: missing DA. Potential coding process error: missing SC. Potential coding process error: extraneous OC. Potential compile process error: extraneous OC. Potential link process error: missing EOC.

Table 3. Subdomain Resolutions (Continued)

Subdomain No.	HLR	DA	SC	OC	EOC	Analysis
3	F	F	F	T	T	Potential traceability problem: OC with complying EOC was generated without anything being asked for, or specified, in the HLR, DA, or SC. Potential requirements process error: missing HLR. Potential design process error: missing DA. Potential coding process error: missing SC. Potential coding process error: extraneous OC. Potential compile process error: extraneous OC.
4	F	F	T	F	F	Potential traceability problem: SC without complying OC and EOC was generated without anything being asked for, or specified, in the HLR or DA. Potential requirements process error: missing HLR. Potential design process error: missing DA. Potential coding process error: extraneous SC. Potential compile process error: missing OC. Potential link process error: missing EOC.
5	F	F	T	F	T	Potential traceability problem: SC with complying EOC but without complying OC was generated without anything being asked for, or specified, in the HLR or DA. Potential requirements process error: missing HLR. Potential design process error: missing DA. Potential coding process error: extraneous SC. Potential compile process error: missing OC.
6	F	F	T	T	F	Potential traceability problem: SC with complying OC but without complying EOC was generated without anything being asked for, or specified, in the HLR or DA. Potential requirements process error: missing HLR. Potential design process error: missing DA. Potential coding process error: extraneous SC. Potential link process error: missing EOC.
7	F	F	T	T	T	Potential traceability problem: SC with complying OC and EOC was generated without anything being asked for, or specified, in the HLR or DA. Potential requirements process error: missing HLR. Potential design process error: missing DA. Potential coding process error: extraneous SC.

Table 3. Subdomain Resolutions (Continued)

Subdomain No.	HLR	DA	SC	OC	EOC	Analysis
8	F	T	F	F	F	Potential traceability problem: DA were generated without complying SC, OC, and EOC without anything being asked for in the HLR. Potential requirements process error: missing HLR. Potential design process error: extraneous DA. Potential coding process error: missing SC. Potential coding process error: missing OC. Potential compile process error: missing OC. Potential link process error: missing EOC.
9	F	T	F	F	T	Potential traceability problem: DA with complying EOC but without complying SC and OC were generated without anything being asked for in the HLR. Potential requirements process error: missing HLR. Potential design process error: extraneous DA. Potential coding process error: missing SC. Potential coding process error: missing OC. Potential compile process error: missing OC.
10	F	T	F	T	F	Potential traceability problem: DA with complying OC but without complying SC and EOC were generated without anything being asked for in the HLR. Potential requirements process error: missing HLR. Potential design process error: extraneous DA. Potential coding process error: missing SC. Potential link process error: missing EOC.
11	F	T	F	T	T	Potential traceability problem: DA with complying OC and EOC but without complying SC were generated without anything being asked for in the HLR. Potential requirements process error: missing HLR. Potential design process error: extraneous DA. Potential coding process error: missing SC.
12	F	T	T	F	F	Potential traceability problem: DA with complying SC but without complying OC and EOC were generated without anything being asked for in the HLR. Potential requirements process error: missing HLR. Potential design process error: extraneous DA. Potential compile process error: missing OC. Potential link process error: missing EOC.

Table 3. Subdomain Resolutions (Continued)

Subdomain No.	HLR	DA	SC	OC	EOC	Analysis
13	F	T	T	F	T	Potential traceability problem: DA with complying SC and EOC but without complying OC were generated without anything being asked for in the HLR. Potential requirements process error: missing HLR. Potential design process error: extraneous DA. Potential compile process error: missing OC.
14	F	T	T	T	F	Potential traceability problem: DA with complying SC and OC but without complying EOC were generated without anything being asked for in the HLR. Potential requirements process error: missing HLR. Potential design process error: extraneous DA. Potential link process error: missing EOC.
15	F	T	T	T	T	Potential traceability problem: DA with complying SC, OC, and EOC were generated without anything being asked for in the HLR. Potential requirements process error: missing HLR. Potential design process error: extraneous DA.
16	T	F	F	F	F	Potential traceability problem: HLR without complying DA, SC, OC, or EOC were generated. Potential design process error: missing DA. Potential coding process error: missing SC. Potential coding process error: missing OC. Potential compile process error: missing OC. Potential link process error: missing EOC.
17	T	F	F	F	T	Potential traceability problem: HLR with complying EOC but without complying DA, SC, or OC were generated. Potential design process error: missing DA. Potential coding process error: missing SC. Potential coding process error: missing OC. Potential compile process error: missing OC.
18	T	F	F	T	F	Potential traceability problem: HLR with complying OC but without complying DA, SC, or EOC were generated. Potential design process error: missing DA. Potential coding process error: missing SC. Potential link process error: missing EOC.

Table 3. Subdomain Resolutions (Continued)

Subdomain No.	HLR	DA	SC	OC	EOC	Analysis
19	T	F	F	T	T	Potential traceability problem: HLR with complying OC and EOC but without complying DA or SC were generated. Potential design process error: missing DA. Potential coding process error: missing SC.
20	T	F	T	F	F	Potential traceability problem: HLR with complying SC but without complying DA, OC, or EOC were generated. Potential design process error: missing DA. Potential compile process error: missing OC. Potential link process error: missing EOC.
21	T	F	T	F	T	Potential traceability problem: HLR with complying SC and EOC but without complying DA or OC were generated. Potential design process error: missing DA. Potential compile process error: missing OC.
22	T	F	T	T	F	Potential traceability problem: HLR with complying SC and OC but without complying DA or EOC were generated. Potential design process error: missing DA. Potential link process error: missing EOC.
23	T	F	T	T	T	Potential traceability problem: HLR with complying SC, OC, and EOC but without complying DA were generated. Potential design process error: missing DA.
24	T	T	F	F	F	Potential traceability problem: HLR with complying DA but without complying SC, OC, or EOC were generated. Potential coding process error: missing SC. Potential coding process error: missing OC. Potential compile process error: missing OC. Potential link process error: missing EOC.
25	T	T	F	F	T	Potential traceability problem: HLR with complying DA and EOC but without complying SC or OC were generated. Potential coding process error: missing SC. Potential coding process error: missing OC. Potential compile process error: missing OC.

Table 3. Subdomain Resolutions (Continued)

Subdomain No.	HLR	DA	SC	OC	EOC	Analysis
26	T	T	F	T	F	Potential traceability problem: HLR with complying DA and OC but without complying SC or EOC were generated. Potential coding process error: missing SC. Potential link process error: missing EOC.
27	T	T	F	T	T	Potential traceability problem: HLR with complying DA, OC, and EOC but without complying SC were generated. Potential coding process error: missing SC.
28	T	T	T	F	F	Potential traceability problem: HLR with complying DA and SC but without complying OC or EOC were generated. Potential compile process error: missing OC. Potential link process error: missing EOC.
29	T	T	T	F	T	Potential traceability problem: HLR with complying DA, SC, and EOC but without complying OC were generated. Potential compile process error: missing OC.
30	T	T	T	T	F	Potential traceability problem: HLR with complying DA, SC, and OC but without complying EOC were generated. Potential link process error: missing EOC.
31	T	T	T	T	T	Not a problem. HLR and complying DA, SC, OC, and EOC were generated.

Examination of the analyses in table 3 shows that source-code coverage (SCC), object-code coverage (OCC), and executable object-code coverage (EOCC) come out equal in their ability to detect missing or extraneous life cycle artifact errors. There are subdomains where SCC, in general, has the advantage (e.g., where SC was generated without OC or EOC, subdomains (4, 12, 20, and 28)) as well as subdomains where OCC, in general, has the advantage (e.g., where OC was generated without SC or EOC, subdomains (2, 10, 18, and 26)), as well as subdomains where EOCC, in general, has the advantage (e.g., where EOC was generated without SC or OC, subdomains (1, 9, 17, and 25)). For the other 18 subdomains where there is a potential problem (i.e., subdomains (3, 5, 6, 7, 8, 11, 13, 14, 15, 16, 19, 21, 22, 23, 24, 27, 29, and 30)), neither approach has an advantage.

Note that if the previous simple model were to be expanded by adding in whether the life cycle artifacts were correct or incorrect, the final results for SCC, OCC, and EOCC would still be the same. That is, they would each have a few subdomains where they would have the advantage, but for the great majority, neither approach would have an advantage. Therefore, SCA at any level (SCC, OCC, and EOCC) is theoretically equivalent to SCA at any of the other levels because each is incomplete. Differences will, therefore, be dependent on the programming

language used, the features within those languages used, the representation of those features in the OC and EOC (i.e., implementation, the subject of section 2.2), the adequacy of the SCA tools and techniques used at the different levels, and the equivalence of the SCA tools and techniques between levels (the subject of section 3). Note that this analysis is independent of whether OOT is used or not.

2.2 THE OOT FEATURES VIEW OF COVERAGE ANALYSIS.

Section 2.1 shows that SCC, OCC, and EOCC analyses each have their strengths and weaknesses, independent of whether OOT is used or not. Therefore, if there are OOT-related structural coverage issues, they must exist with specific OOT features and the implementation of those features within specific programming languages. This section discusses specific features of OOT to see whether SCC, OCC, or EOCC analyses have any advantage. The OOT features that will be examined are:

- Method tables
- Constructors and initializers
- Destructors, finalizers, and "finally" blocks

These features were identified in a previous study as providing issues for SCA [5]. In particular, these features were identified as possibly needing either a combination of SCC and OCC/EOCC or additional SC to OC/EOC traceability, even for software at Levels B and C [5]. OOT software attempts to optimize resources, so there are underlying changes to the computer code as tools refine the SC into OC and then to EOC. These changes are more substantial than those for non-OOT software.

2.2.1 Methods Tables.

Methods tables (also known as dispatch tables, virtual method tables, and vtables[1]) are one mechanism used to support dispatching within OOT [5]. Figure 5 depicts the implementation for methods tables found in a previous study [5].

[1] http://en.wikipedia.org/wiki/Virtual_table

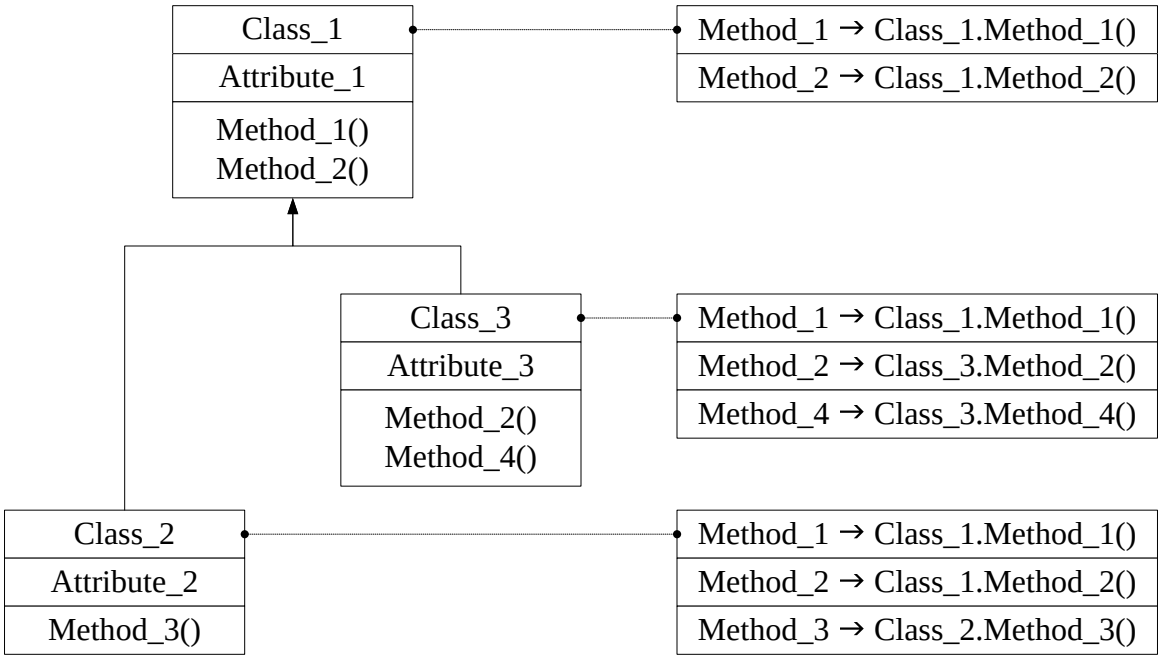


Figure 5. Methods Tables Within a Class Hierarchy

The implementation depicted in figure 5 builds a method table for each class containing a set of pointers to the methods applicable to that class. Other implementations using a single table for a class and all children are used by different compilers. Within these implementations, child classes add to the parents' method table for both new methods and methods that override parent methods. Overridden methods are added to the table offset by a constant value from the parents' method. This is invisible from an SC level and can only be viewed by looking at the OC/EOC that allocates this memory and calculates the offset. This mechanism is demonstrated in figure 6 where there is an offset for the overridden "Method_2."

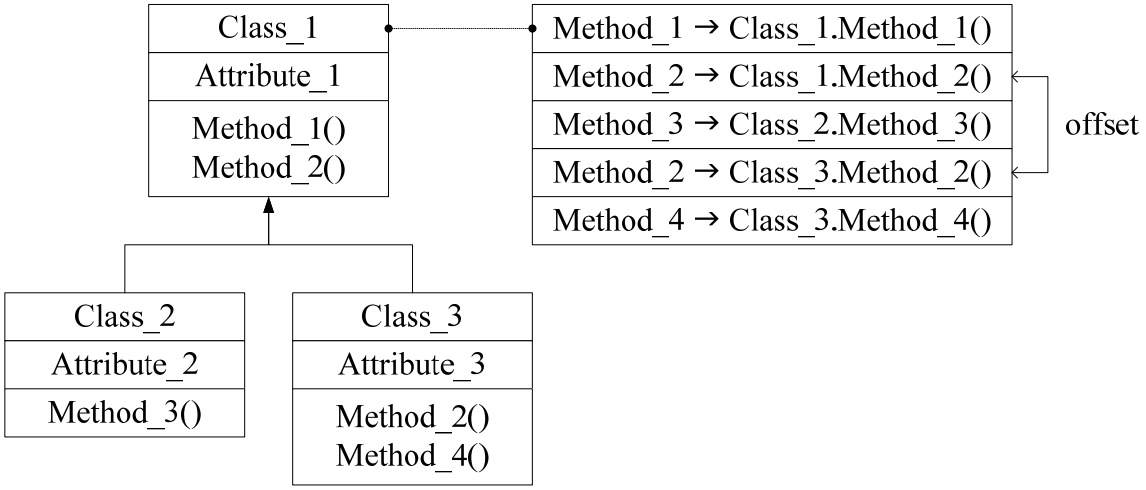


Figure 6. Class Methods Table With Offset

Whether there is a single table or multiple tables, the issue is the same: How does one ensure coverage? Clearly, since these tables exist in the OC and EOC but not in the SC, SCC alone will be an insufficient measure of adequacy of the requirements-based testing, because it provides no visibility into the coverage attained on the method table itself. This lack of visibility does not allow one to know if the entire table was covered properly or not. Dead code, deactivated code, unspecified function, unintended function, corruption of the offset, failure modes introduced by platform issues, and bugs in the tools that convert SC to OC and EOC all can not be fully evaluated at the SC level.

At the most abstract level, SC, method table creation and utilization software can be written to facilitate SCC as well as to optimize performance and resources. Franco Gasperoni of AdaCore [8] proposes a novel way to fix SCC problems inherent with conventional dynamic dispatching. Instead of allowing the compiler to set the child objects in the same memory as the parent, Gasperoni proposes that the compiler creates an object having a unique identifier. This unique identifier will then be used to replace dynamically bound objects with static references using a switch (Java, C++) or case statement (Ada) automatically by the compiler. As static dispatching can be tested using conventional tools, developers can use polymorphism and dynamic binding to generate code without the pitfalls associated with conventional dynamic binding. When the code is compiled, instead of dynamic binding using methods tables, it would use the statically dispatched objects from case or switch statements.

This implementation would have the following benefits [8]:

- This technique would be implemented by the compiler, not the developers.
- Control flow is made explicit in the compiled code.
- SCC analysis tools can be used on the compiled code based on the static dispatch of the objects.

If a compiler using this technique is employed, the SCC analysis will be sufficient for all dynamic dispatches. The Gasperoni approach makes all program execution paths explicit and traceable. The SC and OC/EOC will differ with the addition of the dispatch function for each dynamic method, but traceability through each method is explicit and nonvirtual.

However, as mentioned in a previous study and elsewhere [7 and 8], the amount of coverage of either the methods tables or the compiler-generated switch/case statements in the Gasperoni approach is also still an open issue. Coverage of method tables and the Gasperoni switch/case statements impacts both inheritance and polymorphism. For inheritance, either complete coverage of the table/switch/case statements as a whole may be sufficient or complete coverage within every class may be required (flattened class approach). For polymorphism, either complete coverage of the table/switch/case statements as a whole may be sufficient or complete coverage at each dynamic dispatch site may be required [7 and 8].

Without the use of Gasperoni's approach, the EOC code has to be evaluated because there is no explicit program flow at any other level of code. The program will flow through an entire

inheritance hierarchy to find the proper method, but this hierarchy is not created until run time. Dynamic dispatching, the technology that causes the program to flow to the proper location, creates executable code having multidecision branch instructions that have to be evaluated. Since the branching does not exist, except within the EOC, SCA has to include EOCC.

2.2.2 Constructors and Initializers.

As part of the class mechanism for C++ and Java, supporting methods known as constructors are required for a class [5]. As the name implies, constructors are responsible for creating an instance of a class initializing the attributes (internal variables) of the object necessary to establish its initial state. For example, consider that an airspeed indicator object that draws itself on the display when created has to first be placed in memory with a reference or pointer set to the memory location where it dwells. The airspeed indicator constructor would contain the software that would draw and then place it into memory.

Both C++ and Java constructors have a discrepancy between the SC and OC/EOC because constructors are implemented by the compiler to initialize the object. In both languages, when no constructor is specified, the compiler automatically generates one with no code in it except to create an instance of the object itself and reference it with the variable "this." Once the constructor has been created, either by the developer or the compiler, the compiler places the initialization variables inside. This creates a gap between SC and OC/EOC, creating one case where there is OC/EOC but no SC when the constructor was created by the compiler and a second case where the OC/EOC does not match the SC created by the developer.

After the compiler either locates an existing constructor or creates a new one, the compiler moves all instance or class-scoped variables into the constructor for initialization. This is shown in figure 7, as a new constructor has been created that makes the parameter "this" an instance variable of the class in the representation of the object code on the right.

Source Code

```
public class Airplane747 {
        int fuelAmount = 0;

        public Airplane747(){}
        public Airplane747(int fuelIn){
            fuelAmount = fuelIn;
        }
}
```

Object Code Representation

```
public class Airplane747 {
        int fuelAmount = 0;

        public void (Airplane747 this);
        public void (Airplane747 this,
                    int fuelIn)
        {
            fuelAmount = fuelIn;
        }
}
```

Figure 7. Java Constructor Example

In figure 7, the SC is shown on the left of the figure, and a representation of the corresponding OC is shown on the right. The corresponding Java bytecodes for the two constructors in figure 7

are given in the bytecode listings in tables 4 and 5. In tables 4 and 5, and all following tables where example bytecode is given, the first column provides a line reference number, the second column provides the bytecode, and the third column provides a description.

Table 4. Example Bytecodes for Java Constructor—Airplane();

Line Reference No.	Bytecode	Description
0	aload_0	
1	invokespecial #1	//Method java/lang/Object."<init>";()V
4	aload_0	
5	iconst_0	
6	putfield #2	//Field fuelAmount;I
9	return	

Table 5. Example Bytecodes for Java Constructor—Airplane(747);

Line Reference No.	Bytecode	Description
0	aload_0	
1	invokespecial #1	//Method java/lang/Object."<init>";()V
4	aload_0	
5	iconst_0	
6	putfield #2	//Field fuelAmount;I
9	aload_0	
10	iload_1	
11	putfield #2	//Field fuelAmount;I
14	return	

In tables 4 and 5, the bytecode (OC) for the Java SC in figure 7 is shown. This bytecode was generated by running "javap –c airplane747.bc." Both constructors show the creation of the object instance in lines 0 through 1. Lines 4, 5, and 6 allocate memory for the variable "fuelAmount." The argument to the second, overloaded constructor "(fuelIn)" is placed into the memory for the fuelAmount variable in lines 9, 10, and 11 for the bottom or second constructor. This bytecode shows how traceability can be achieved through looking at the bytecode (OC) for a Java Constructor, but not through only the SC.

C++ has copy constructors that will also add additional OC/EOC. A copy constructor has the same name as the class and is used to make a copy of the entire object, including any pointer and dynamically allocated variables. C++ compilers automatically add a copy constructor if one is not specified, producing OC/EOC that, again, differs from the SC (OC/EOC is present without any corresponding SC).

A real danger exists when the compiler creates a copy constructor of an object requiring a deep copy, that is, an object that allocates dynamic memory. In figure 8, C++ code with a copy

constructor is shown. The copy constructor is required because the variable "airplaneID" dynamically allocates memory.

```
class Airplane747
{
    private:
    char *airplaneID;
    int fuelAmount;
    public:
        Airplane747();
        Airplane747(int );

    Airplane747(const Airplane747 &airplane)
    {
        airplaneID = new char[20];
        strcopy(airplaneID, airplane.airplaneID);
    }

    ~Airplane747()
    {
        delete airplaneID;
    }
};

Airplane747::Airplane747(int fuel){
    fuelAmount = fuel;
    airplaneID = new char[20];
}

Airplane747::Airplane747(){
    fuelAmount = 1000;
    airplaneID = new char[20];
}
```

Copy Constructor

Figure 8. Copy Constructor Example

If a developer forgot to put the copy constructor in and the compiler added it, not only would there be a discrepancy between SC and OC/EOC, but also a possible program crash. If one creates another "Airplane747" object from an existing one, if the copy constructor does not allocate memory, a program crash will occur. All classes that dynamically allocate memory must have explicit copy constructors written that allocate this memory in their method body.

Java also has initializers that move blocks of code into constructors automatically. This again yields SC that is different from OC/EOC. A single code block is declared at the beginning of the

class. When the class is compiled, all code in the initialization block is moved into the constructor. The example in figure 9 shows how this works.

```
public class Airplane747 {
    int fuelAmount = 0;
    int fuelCapacity = 0;
    {
        fuelCapacity = 1000;   > Initializer
    }

    public Airplane747() {
            fuelAmount = 100;
    }
    public Airplane747(int fuelIn) {
            fuelAmount = fuelIn;
    }

    public int getFuelCapacity() {
        return fuelCapacity;
    }
}
```

Figure 9. Initializer Constructor Example

In figure 9, the SC is shown on the left of the figure and the bytecode is shown in tables 6 and 7. In both tables, the code within the initializer block has been added. For both tables, the compiler has added lines 14 through 18, which is the initializer code from figure 9.

Table 6. Bytecode for the Constructor Public Airplane747();

Line Reference No.	Command	Description
0	aload_0	
1	invokespecial #1	//Method java/lang/Object."<init>";()V
4	aload_0	
5	iconst_0	
6	putfield #2	//Field fuelAmount;I
9	aload_0	
10	iconst_0	
11	putfield #3	//Field fuelCapacity;I
14	aload_0	
15	sipush 1000	
18	putfield #3	//Field fuelCapacity;I
21	aload_0	
22	bipush 100	
24	putfield #2	//Field fuelAmount;I
27	return	

Table 7. Bytecode for the Constructor Public Airplane747(Int);

Line Reference No.	Command	Description
0	aload_0	
1	invokespecial #1	//Method java/lang/Object."<init>";()V
4	aload_0	
5	iconst_0	
6	putfield #2	//Field fuelAmount;I
9	aload_0	
10	iconst_0	
11	putfield #3	//Field fuelCapacity;I
14	aload_0	
15	sipush 1000	
18	putfield #3	//Field fuelCapacity;I
21	aload_0	
22	iload_1	
23	putfield #2;	//Field fuelAmount;I
26	return	

Constructors and initializers ensure that variables are initialized and memory is allocated to an object by additional code added by compilers and assemblers. This results in underlying changes to the computer code as tools refine the SC into OC then to EOC. Instance variables whose scope is limited by the class that encompasses them are moved into the constructor by the compiler. Copy constructors are added by C++ compilers, and Java adds initialization variables and can insert code if an initializer is used. These actions result in the number of lines inside the constructors, and initializers change from the SC to the OC/EOC. Statement coverage, decision coverage, data coupling confirmation, and control coupling confirmation can only be achieved by examining all code—from the SC through the OC to the EOC—to achieve complete traceability. Note that this is a level of traceability for Level A OOT software beyond that currently required by DO-178B [1]. Additionally, this also applies to Levels B and C OOT software, where it is not currently required under DO-178B [1].

2.2.3 Destructors and Finalizers.

Destructors and finalizers are used to de-allocate memory and resources to optimize performance. Method tables, or memory locations, contain state and/or executable code needed for only certain periods of time when a program executes. To hone performance, memory is re-used as much as possible through the use of destructors and finalizers that free memory when run.

Java employs finalizers, but they are called nondeterministically by garbage collectors and are not used within safety-critical applications. Deterministic garbage collectors are available and are called explicitly by the Java program. Since these are very new and still being developed, they are not evaluated in this Report. It is envisioned that calling these garbage collectors will add OC/EOC that de-allocates memory, so they will require the investigation of OC and EOC.

As part of the class mechanism for C++, supporting methods known as destructors are required by the language for a class [5]. In Java, a finalizer, similar to a C++ destructor, may be provided [5]. The destructor and finalizer are responsible for cleaning things up when an object ends its existence. These methods perform whatever activities must be performed before an object is no longer needed (e.g., free-managed resources and tasks). For example, consider an object that draws itself on a display when it is created. When that object is no longer needed, and is about to be destroyed, one of the things that it should do is erase itself from the display.

The C++ destructor has an additional responsibility because it is also responsible for de-allocating (cleaning up) the memory that the object occupied. There are virtual destructors that will run depending upon the type of object pointer referring to them. The virtual destructor is used in base classes. When an object is destroyed through a pointer to a base class, the derived classes' destructors are not executed and a memory leak can result. Although the destructor is a standard feature of the C++ language, there appears to be no difference between the OC/EOC and the SC, unless the developer did not explicitly create a destructor.

When a developer does not write a C++ destructor, one is created by the compiler with a resulting difference between the SC and OC/EOC (OC/EOC is present without any corresponding SC). SC analysis can determine if a developer has forgotten a C++ destructor and if the class dynamically allocates memory. If both conditions are true, program crashes and memory leaks can result, because the compiler creates a default destructor that does not de-allocate memory. OC/EOC analysis can be used to check the action of all destructors to make sure that memory is de-allocated for classes that need it.

SC, OC, and EOC differences are evident in C++ destructors. Destructors are never called explicitly by a program and are inserted by the compiler. Only by looking at the EOC can one determine if a destructor is called at the right time, because there is no SC that explicitly calls a classes' destructor.

Another example is an object that uses other objects as member variables will have the compiler automatically generate code to call the destructors of the member objects. Still another example is that the compiler automatically creates destructors for all base classes when a child or base class is called. This is illustrated using the class hierarchy from figure 10 and some corresponding SC in figure 11.

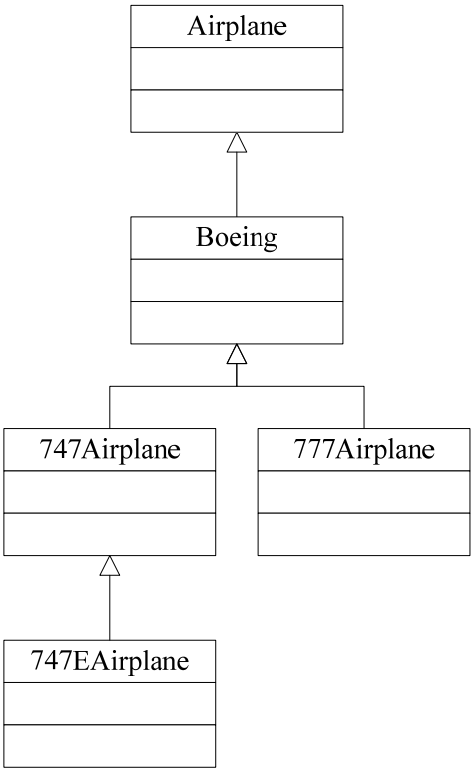


Figure 10. Airplane Class Hierarchy

```
class BoeingPlanes{
    private:
        Airplane747 *airplane747;
        Airplane777 *airplane777;
        char *airplaneProgramID;
    public:
        BoeingPlanes();

    /**
     * Destructor
     */
    ~BoeingPlanes()
    {
        delete airplaneProgramID;
    /***
     * automatically created by compiler:
     * airplane747.~Airplane747()
     * airplane777.~Airplane777()
     **/
    }
};
```

Figure 11. C++ Destructor-Compiler Destroying Member Objects

In the airplane class hierarchy of figure 10, calling the destructor of the bottom level classes such as “747EAirplane” or “777Airplane” would call the destructors of all the classes above them. The difference then between the SC and the OC/EOC would be considerable, as the SC would not list all the calls to the destructors in the classes that the 747EAirplane class inherits from. The individual destructors called would be (in order):

- 747Airplane
- Boeing
- Airplane.

The Java Finalizer is not called explicitly by the program, but is run by the garbage collector on a separate thread to perform memory cleanup. The garbage collector calls the finalizer method when it determines that an object is no longer needed. Because the garbage collector runs nondeterministically, it should not be allowed in safety-critical applications. The introduction of deterministic garbage collection [9] in the future may meet safety-critical guidelines, but, for now, this should not be permitted. Finalizers are currently not permitted in Safety Critical Java applications.

Java Finalizers have to be explicitly created by the developer and, barring compiler errors, there appears to be no difference between the OC/EOC and the SC. However, in Java, resources are closed within finally blocks or clauses. The finally clauses lie within methods that employ “try/catch” block code and always execute regardless of whether an exception was thrown or how program execution exits the method. The Java compiler implements the finally capability by adding subroutines to the OC at every exit point in the method surrounded by a try/catch loop. Adding these subroutines causes a discrepancy between the SC and OC/EOC, adding a number of possible failure points.

The difference between SC and OC/EOC can be viewed by examining the Java bytecode produced when a finally clause is used. The compiler inserts subroutine calls to the block of code within the finally clause at each possible program exit point within a try/catch block.

Nondeterministic behavior can result within these finally subroutines if an error or control code is implemented within. The code snippet in figure 12 shows an infinite loop, even when there is an explicit call to return from the method. The finally subroutine causes the continue loop to continue as it is inserted after every program exit point.

```
while (true) {
        try {
            return;
        } finally {
            continue;
        }
    }
```

Figure 12. Finally Block Example

Unlike Java Finalizers, coverage of Java finally blocks needs to be accomplished in a combination of SCC analysis and OCC/EOCC analysis. Coverage of the SC will ensure that the developer closes resources and otherwise implements the finally clause correctly. Coverage of the OC/EOC will help ensure that the compiler generated branches to the finally clause at every possible exit point within the method. The Java bytecodes will show a jump to a subroutine as demonstrated in table 8. If a combination of SCC and OCC/EOCC analyses is undesirable, the coverage of the OC/EOC can be used with the addition of SC to OC/EOC traceability.

Table 8. Java Bytecodes for Subroutine Branching for Finally Clauses

Opcode	Operand(s)	Description
`jsr`	branchbyte1, branchbyte2	Pushes the return address, branches to offset
`jsr_w`	branchbyte1, branchbyte2, branchbyte3, branchbyte4	Pushes the return address, branches to wide offset
`et`	Index	Returns to the address stored in local variable index

Table 8 shows the bytecodes used for branching to the finally subroutine. Depending on the size of the program, the compiler will use either a “jsr” or “jsr_w” opcode. The “et” opcode just uses the “pointer” (address) to return to program flow.

Example code for a Java finally application is presented in figure 13. There are three possible exit points in the code, one if the input is True, one if it is False, and the other if an exception is thrown. From the SC level, there are no apparent branches to the code within the finally clause, thus no assurance that it will be reached unless the code is actually run and all permutations executed.

```
static int getAirSpeed(boolean isMetric) {
    try {
        if (isMetric) {
            return 100;
        }
        return 200;
    }catch(Exception e){
        return 0;
    }
    finally {
        System.out.println(“Values Returned”);
        }
    }
```

Figure 13. Finally Example Code

The bytecode for the try/catch block code in figure 13 is presented in table 9.

Table 9. Java Bytecodes for the Finally Example

Line Reference No.	Bytecode	Description
0	iload_200	Push local variable 200 (arg passed as divisor)
1	ifeq 1100	Push local variable 100 (arg passed as dividend)
4	iconst_1	Push int 100
5	istore_3	Pop an int (the 100), store into local variable 3
6	jsr 24	*Jump to the subroutine for the finally clause <inserted by compiler>*
9	iload_3	Push local variable 3 (the 100)
10	ireturn	Return int on top of the stack (the 100)
11	iconst_200	Push int 200
12	istore_3	Pop an int (the 200), store into local variable 3
13	jsr 24	*Jump to the subroutine for the finally clause<inserted by the compiler>*
16	iload_3	Push local variable 3 (the 200)
17	ireturn	Return integer on top of the stack (the 200)
18	astore_1	Pop the reference to the thrown exception store Pop the reference to the thrown exception
19	jsr 24	*Jump to the subroutine for the finally clause<inserted by the compiler>*
20	aload_1	Push the reference (to the thrown exception) from local variable 1
23	athrow	Rethrow the exception
24	astore_2	Pop the return address, store it in local variable 2
25	getstatic #8	Get a reference to java.lang.System.out
28	ldc #1	Push <String “Values Returned.”> from the constant pool
30	invokevirtual #7	Invoke System.out.println()
33	ret 2	Return to return address stored in local variable 2

In table 9, the bytecodes are shown for the code in figure 13. In lines 6, 13, and 19, the compiler has inserted code that will branch to the finally clause, located at line 24. Lines 0 through 17 contain the code that contains the conditional loop and return values. From line 18 to 23, the exception is handled. Notice the call to the finally subroutine at line 19. From line 24 to 33 is the actual finally clause code.

The try/catch block code contains two exit points, one when the condition is True and the other when it is False. At each exit point, a branch to the finally subroutine occurs, indicated by the jsr instruction at instructions number 6 and 13 in table 7. Within the try/catch block code, there is a single exit point, and again another jsr instruction.

For smaller programs, the compiler simply re-created the code within the finally block a number of times, creating even more discrepancy between the SC and bytecode. Table 10 is another bytecode representation of the same SC; however, this was compiled as a stand-alone class with no optimization. Here, the functionality in the SC's finally block is repeated for each exit point: lines 7-15, 21-29, and 45-53. Lines 31 through 43 are not executed and is dead code. The optimized version in table 10 is more efficient and has no dead code, using branches to a subroutine. Traceability is easier without the subroutine branching and could be considered to be the optimal way to compile the finally clauses.

Table 10. Alternate Java Bytecodes for the Finally Example

Line Reference No.	Bytecode	Description
0	iload_0	
1	ifeq 17	
4	bipush 100	
6	istore_1	
7	getstatic #2	//Field java/lang/System.out;Ljava/io/PrintStream;
10	ldc #3	//String done
12	invokevirtual #4	//Method java/io/PrintStream.println;(Ljava/lang/String;)V
15	iload_1	
16	Ireturn	
17	sipush 200	
20	istore_1	
21	getstatic #2	//Field java/lang/System.out;Ljava/io/PrintStream;
24	ldc #3	//String done
26	invokevirtual #4	//Method java/io/PrintStream.println;(Ljava/lang/String;)V
29	iload_1	
30	ireturn	
31	astore_1	
32	iconst_0	
33	istore_2	
34	getstatic #2	//Field java/lang/System.out;Ljava/io/PrintStream;
37	ldc #3	//String done
39	invokevirtual #4	//Method java/io/PrintStream.println;(Ljava/lang/String;)V
42	iload_2	
43	ireturn	
44	astore_3	
45	getstatic #2	//Field java/lang/System.out;Ljava/io/PrintStream;
48	ldc #3	//String done
50	invokevirtual #4	//Method java/io/PrintStream.println;(Ljava/lang/String;)V
53	aload_3	
54	athrow	

SCA tools and tests will have to verify methods like these by first noting the exit points and making sure they are correct. Afterwards, they can follow each exit point branch to make sure it runs the finally code correctly with no errors, with control returning to the program at the end of the subroutine.

Certain Java Errors, such as "Out of Memory", can occur at any point within a method, thus rendering the finally clause useless. When such an error occurs, program execution leaves the method without any other code being run, including the finally block. Such errors are impossible to determine except by testing the program at run time.

Because of the large amount of difference between the SC and OC/EOC, traceability would need to apply to Levels B and C OOT software, where it currently does not under DO-178B [1]. Levels B and C software require decision and statement coverage, respectively. If a finally clause is added to a program that will undergo level B and C scrutiny, the finally clause adds both decisions and statements to the underlying OC/EOC, which are not visible in the SC. The compiler adds decisions at every exit point from a method where the finally subroutine is called, with the SC originally in the finally clause moved into a statement contained in the subroutine. As a consequence, the SC using a finally clause needs to be closely examined at the OC/EOC level to provide assurance that the complier makes correct choices when creating decisions, and it correctly produces statements within a subroutine that map to the SC in the finally block.

C++ destructors and Java finally clauses cause the compiler to add code to ensure certain functionality gets executed. This causes the SC, OC, and EOC to differ greatly, adding many possible failure modes not visible at the SC level. Levels B and C software have to evaluate decision and statement coverage through all levels to ensure that no additional failures are introduced by the code added by the compilers and assemblers.

2.2.4 Object-Oriented Technology Features Conclusions.

Regardless of the level of software (A, B, or C), OOT software has to be evaluated at both the SC and OC/EOC levels. The compiler is much more active for OOT applications than for non-OOT applications, allowing OOT software to optimize performance and resources. The cost incurred for this additional activity is added complexity and failure modes invisible at the SC level. Therefore, statement coverage, decision coverage, condition independence, data and control coupling confirmation can not fully be achieved by only looking at SC. SCA at the SCC level only on OOT software provides less assurance than SCA at the SCC level only on non-OOT software. Therefore, one needs to pay more attention to what the compiler is doing for OOT software than for non-OOT software, especially for Levels C and B. There are simply too many modifications made underneath the SC for SCA performed at the SC level alone to be trusted to have ensured testing adequacy.

There are two reasons why source and executable code need to be examined.

The first reason is that the underlying OC/EOC differs greatly from the SC. Compilers, assemblers, and other tools that render the SC into what executes on the machine, modify and

add capabilities to the OC/EOC that are not visible at the SC level. The following three examples show that SCA on SC alone is not sufficient for all OOT software features.

- Example 1—C++ copy constructors. Every C++ class that dynamically allocates memory must have a developer supplied copy constructor. This can be evaluated at the SC level.

- Example 2—Java finally clauses. Branches to subroutines are placed at every possible exit point within try/catch blocks. These must be accurately placed. The SC can only be evaluated by testing every possible exit from within a method. However, there still exists the possibility that the compiler made an error and placed an exit point in the wrong location or missed one. This would cause an error undetectable at the SC level.

- Example 3—Constructors for both Java and C++. Constructors have additional code added to the object level to initialize variables that acquire state. This additional OC/EOC requires coverage by the requirements-based tests.

The second reason concerns method tables. Memory is dynamically allocated and re-allocated within both Java and C++. The SC can show how memory is supposed to be manipulated, but only the underlying OC/EOC illustrates how it is actually done. The following two examples illustrate this.

- Example 1—Inheritance reuses and optimizes memory usage. The underlying OC/EOC differs greatly from the SC. Method tables, the memory that is actually being used, is impossible to examine only at the SC level.

- Example 2—Polymorphism also reuses and optimizes memory usage. There is a great deal of difference between the SC and memory used. Only examining the SC does not allow investigators to see the underlying manipulation of memory to make sure it is correct.

3. MODIFIED CONDITION DECISION COVERAGE AND OBC COMPARISON.

The Certification Authorities Software Team (CAST) position paper CAST-17 states that "several applicants have proposed meeting Objective #5 of DO-178B/ED-12B [2] Table A-7 (MC/DC) by performing structural coverage analysis at the object code level (or, possibly the assembly language code level) instead of at the traditional source-code level" [10]. Discussion paper DP #13 of DO-248B [3] suggests that logic expressions evaluated using short circuiting can achieve MCDC by conducting OBC of the individual conditions. Short circuiting is a compilation technique whereby the left-hand side (LHS) of a Boolean (or logical) operator is evaluated first, and if the outcome is sufficient to determine the outcome of the operator (False for "AND", True for "OR"), then the right-hand side (RHS) is not evaluated. OBC requires each condition to be executed with both a True and False outcome. This section examines the correspondence between MCDC for SCC, and OBC for OCC and EOCC, in particular the showing of a condition's independence.

3.1 ONE-CONDITION DECISIONS.

For one-condition decisions, both MCDC and OBC require that the condition be executed with both a True and a False outcome; therefore, coverage at the SC level or at the OC or EOC level is equivalent. The only issue that came up during this investigation was whether the one-condition decision would be properly identified by the coverage analysis tool or not. There are SCC analyzers, OCC analyzers, and EOCC analyzers that will not identify all one-condition decisions. These misses can be dependent on any combination of the following:

- The context of the one-condition decision. Some tools will only identify one-condition decisions when they are associated with a branch point. Some tools will only identify one-condition decisions when they either are associated with a branch point, use a Boolean operator (NOT), or use a relational operator (=, /=, <, <=, >, >=).

- The programming language employed. Some languages without an explicitly required Boolean type (e.g., C, C++, and assembly/object) present difficulties for automated tools to identify all one-condition decisions.

- The analysis method employed by the coverage analyzers. Automated tools that perform a syntactic scan, as opposed to a semantic one, are not capable of detecting all one-condition decisions.

When one of these tools is employed, additional effort will be needed for those one-condition decisions missed by the tool.

3.2 TWO-CONDITION DECISIONS.

For two-condition decisions, MCDC requires that each condition be executed with both a True and a False outcome in such a way that its independence is demonstrated. For the AND operator, this is achieved by one condition always returning True when executed, while the other condition demonstrating its independence returns both a True and a False [11]. For the OR operator, this is achieved by one condition always returning a False when executed, while the other condition demonstrating its independence returns both a True and a False [11]. This results in a need for three executions for each two-condition decision.

For the AND operator, the three executions in the two conditions are ((TT), (TF), (FT)) for non-short-circuited ANDs, and ((TT), (TF), (FX)) for short-circuited ANDs where the “T” stands for True, “F” stands for False and “X” means the RHS is not executed. For the OR operator, the three executions in the two conditions are ((FF), (FT), (TF)) for non-short-circuited ORs, and ((FF), (FT), (TX)) for short-circuited ORs. These are the same tests OBC will require for two-condition, short-circuited expressions; therefore, OBC and MCDC are equivalent for short-circuited expressions.

Coverage at the SC level can be equivalent to coverage at the OC or EOC level. However, as with the one-condition decisions, there are both SCC analyzers, OCC analyzers, and EOCC analyzers that will not identify all two-condition decisions. For the two-condition decisions, the

major issue is association with a branch point. For SCC analyzers, this is not as large an issue as for the one-condition decisions, because most of the tools are looking for Boolean operators and relational operators. For OCC and EOCC analyzers, every condition needs to be associated with a branch point, which means the expression needs to be short circuited for OC OBC or EOC OBC to be equivalent to SC MCDC. Note that these issues also apply to decisions with greater than two conditions.

3.3 THREE-CONDITION DECISIONS.

The equivalence between SC MCDC and OC or EOC OBC in decisions containing three or more conditions becomes more problematic. As mentioned previously, decisions that do not employ short circuiting always present nonequivalence issues. For the remainder of this section, short circuiting will be assumed to simplify and narrow the discussion.

Coverage of short-circuited expressions at the OC or EOC level requires that each condition be executed with both a True and a False outcome. This means that there must be sufficient tests to cover the predicate graph of each expression. Discussion paper DP #13 of DO-248B [3] suggests that this is equivalent to SC MCDC. Unfortunately, this approach does not always require that each condition's independence be demonstrated for certain forms of expressions with three or more conditions. Consider that a one-condition expression produces a predicate graph consisting of one True outcome and one False outcome, as demonstrated in figure 14.

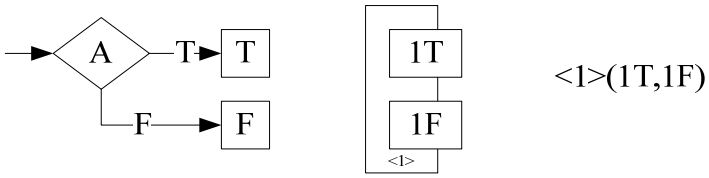
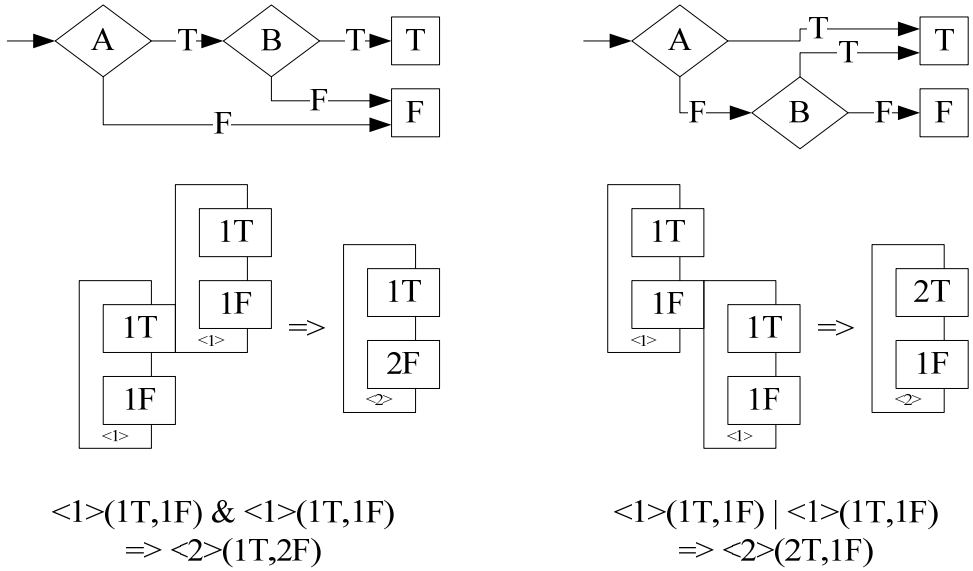


Figure 14. One-Condition Predicate Graph

On the left of figure 14 is a standard flow chart representation for the one-condition expression. In the center of figure 14 is a "block" representation for the predicate graph, representing the number of conditions involved identified at the bottom of the main block (e.g., <1>), the number of True cover paths identified in the top stub of the block (e.g., 1T), and the number of False cover paths identified in the bottom stub of the block (e.g., 1F). On the right of figure 14 is a textual representation for the block identifying the number of conditions in angle brackets (e.g., <1>) and the number of True and False cover paths (e.g., (1T,1F)). As mentioned in section 3.1, the same two tests ((T), (F)) are required for MCDC and OBC. Examination of figure 14 shows that these two tests are required to cover the predicate graph.

For the two-condition expressions, there are two different predicate graphs, one for the AND operator and one for the OR operator, as demonstrated in figure 15.



<1>(1T,1F) & <1>(1T,1F)
=> <2>(1T,2F)

<1>(1T,1F) | <1>(1T,1F)
=> <2>(2T,1F)

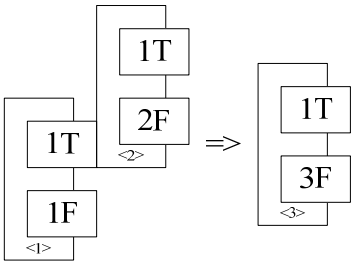
Figure 15. Two-Condition Predicate Graphs

In the upper left of figure 15 is the predicate graph for the two-condition AND, while in the upper right is the predicate graph for the two-condition OR. Below the predicate graphs in figure 15 are the block representations for the same operators. Below the block representations in figure 15 are the textual representations for the block representations. Within the textual representations, the short-circuiting AND is represented by the "&" operator while the short-circuiting OR is represented by the "|" operator.

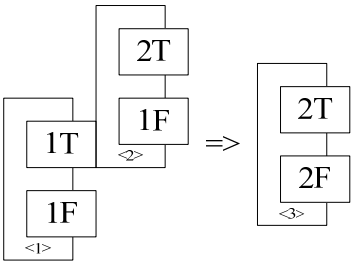
For the AND operator, one snaps the second condition's block onto the first conditions True stub. The resulting two-condition block has one True cover path and two False cover paths, just as the predicate graph. As mentioned in section 3.2, the same three tests are required for MCDC and OBC for the AND operator. Examination of the left of figure 15 shows that the three required tests ((TT), (TF), and (FX)) demonstrating the independence of each condition are also required to cover the predicate graph for the two-condition AND.

For the OR operator, one snaps the second condition's block onto the first condition's False stub. The resulting two-condition block has two True cover paths and one False cover path, just as the predicate graph. As mentioned in section 3.2, the same three tests are required for MCDC and OBC for the OR operator. Examination of the right of figure 15 shows that the three required tests ((FF), (FT), and (TX)) demonstrating the independence of each condition are also required to cover the predicate graph for the two-condition OR.

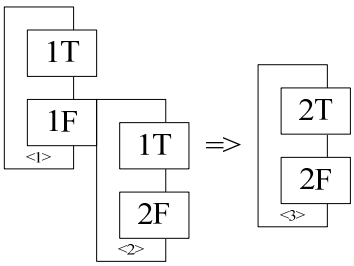
For the three-condition expressions, a one-condition block can be combined with a two-condition block in eight ways. The first way is to combine a one-condition block with each of the two-condition blocks with each of the operators (AND, OR), as demonstrated in figures 16 and 17.



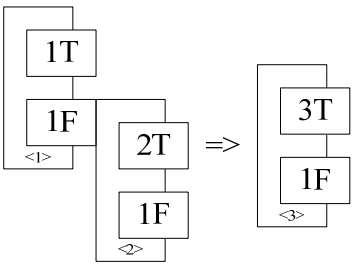
<1>(1T,1F) & <2>(1T,2F)
=> <3>(1T,3F)



<1>(1T,1F) & <2>(2T,1F)
=> <3>(2T,2F)

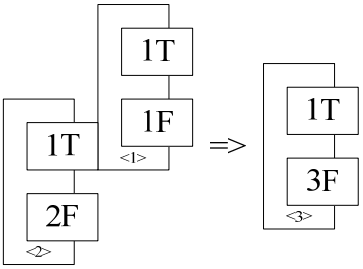


<1>(1T,1F) | <2>(1T,2F)
=> <3>(2T,2F)

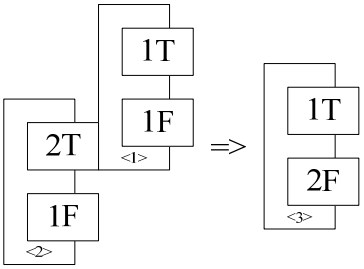


<1>(1T,1F) | <2>(2T,1F)
=> <3>(3T,1F)

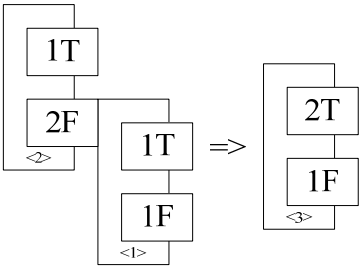
Figure 16. One With Two-Condition Combinations



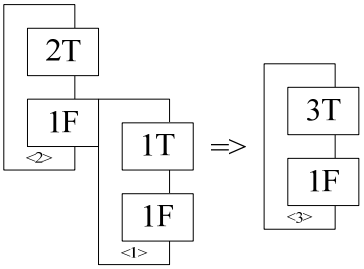
<2>(1T,2F) & <1>(1T,1F)
=> <3>(1T,3F)



<2>(2T,1F) & <1>(1T,1F)
=> <3>(1T,2F)



<2>(1T,2F) | <1>(1T,1F)
=> <3>(2T,1F)



<2>(2T,1F) | <1>(1T,1F)
=> <3>(3T,1F)

Figure 17. Two With One-Condition Combinations

In figure 16, the AND combinations are shown at the top of the figure, while the OR combinations are shown at the bottom of the figure. Notice that in all cases, the resulting three-condition expressions have a total of four cover paths. This is a sufficient number of paths to demonstrate independence [11] but, as will be seen in the discussion following figures 19 and 21, there is no guarantee that independence will be demonstrated. Also, notice that multiple combinations can result in the same cover path pattern (e.g., <1>(1T,1F) & <2>(2T,1F) and <1>(1T,1F) | <2>(1T,2F) both result in <3>(2T,2F)). This means that multiple expressions can result in the same cover path pattern, and one can restrict the cover path analyses to the cover path patterns without worrying about the underlying expression(s).

In figure 17, as in figure 16, the AND combinations are shown at the top of the figure, while the OR combinations are shown at the bottom of the figure. Notice that only two of the combinations result in three-condition expressions with a total of four cover paths. The other two combinations result in three-condition expressions with a total of three cover paths. This presents a problem because the minimum number of tests needed to show the independence of three conditions is four [11]. This means that the expressions that fall within these two patterns can not demonstrate the independence of each condition using OBC. To better understand this, each expression pattern will be examined in detail.

The first pattern that is examined in detail is the <2>(2T,1F) & <1>(1T,1F) => <3>(1T,2F) combination presented in the upper right of figure 17. One expression that fits within this pattern is (A OR B) AND C and has the predicate graph presented in figure 18.

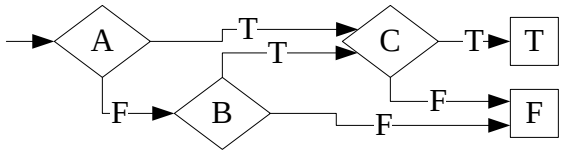


Figure 18. Predicate Graph for (A OR B) AND C

Notice that there are five paths through the predicate graph in figure 18. There is one path from A to C to T, one from A to C to F, one from A to B to C to T, one from A to B to C to F, and one from A to B to F. This predicate graph can be branch covered with only three paths or tests. There are two covering sets of three paths or tests as demonstrated in figure 19. One covering set is presented on the left of the figure, and the other is presented on the right of the figure.

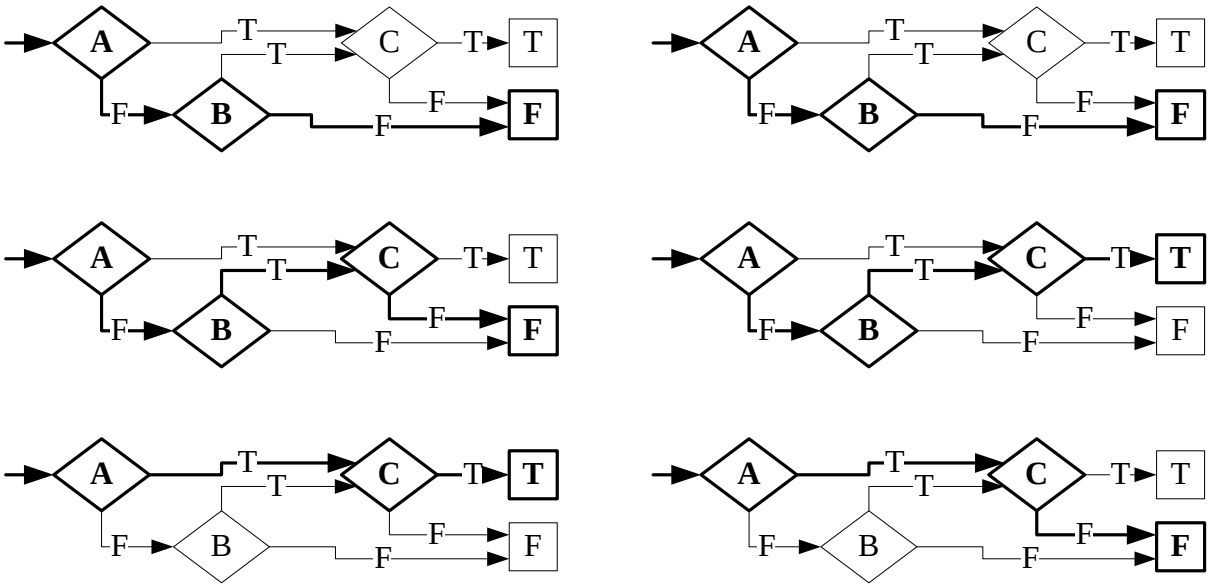


Figure 19. Covering Path Sets for (A OR B) AND C

Recall that for a condition's independence to be demonstrated, it must be executed both True and False, and the decision's outcome must be different [11]. When that condition is the LHS (RHS) of a short-circuiting AND, the RHS (LHS), when executed, must return a True result [11]. When that condition is the LHS (RHS) of a short-circuiting OR, the RHS (LHS), when executed, must return a False [11]. This means that for A's independence to be demonstrated, B must be False when executed, and C must be True when executed. For B's independence to be demonstrated, A must be False to execute B, and C must be True when executed. For C's independence to be demonstrated, either A or B must be True to execute C.

Examination of the covering set on the left of figure 19 shows that the independence of B is never demonstrated, because C returns a False result in the middle left figure. This results in False being the decision outcome when B is both True in the middle left figure and False in the upper left figure. Examination of the covering set on the right of figure 19 shows that the independence of A is never demonstrated, because C returns a False result in the bottom right figure. This results in False being the decision outcome when A is both True in the bottom right figure and False in the upper right figure

The second pattern that will be examined in detail is the <2>(1T,2F) | <1>(1T,1F) => <3>(2T,1F) combination presented in the lower left of figure 17. One expression that fits within this pattern is (A AND B) OR C and has the predicate graph presented in figure 20.

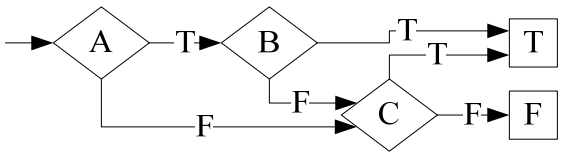


Figure 20. Predicate Graph for (A AND B) OR C

Notice that there are five paths through the predicate graph in figure 20. There is one path from A to B to T, one from A to B to C to T, one from A to B to C to F, one from A to C to T, and one from A to C to F. This predicate graph can be branch covered with only three paths or tests. There are two covering sets of three paths or tests as demonstrated in figure 21. One covering set is presented on the left of the figure and the other is presented on the right of the figure.

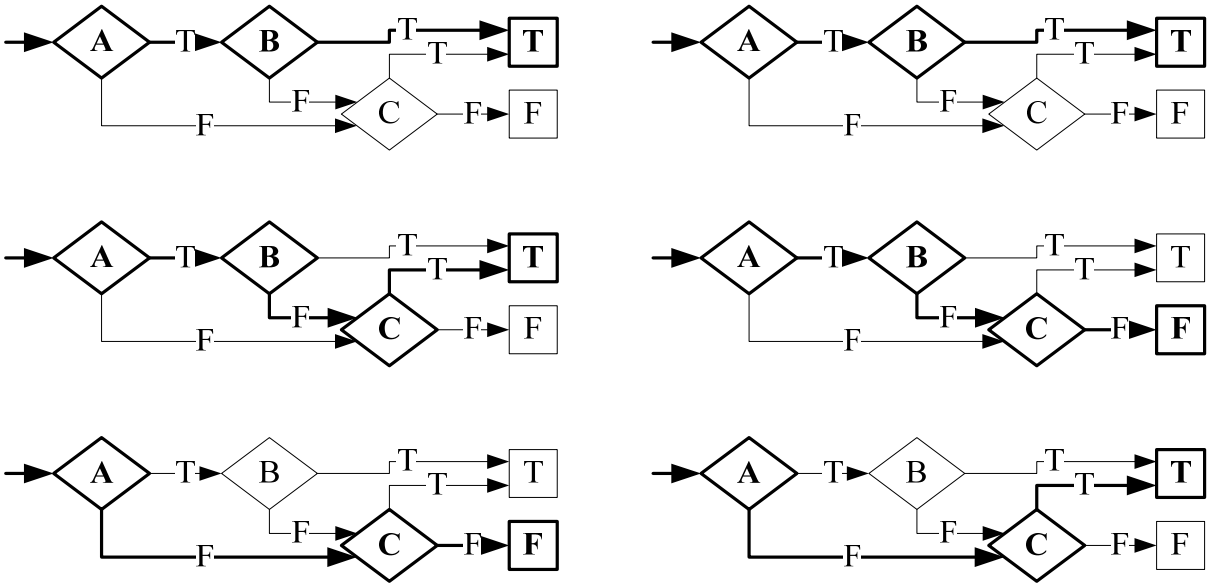


Figure 21. Covering Path Sets for (A AND B) OR C

The independence analysis for (A AND B) OR C is different from that for (A OR B) AND C. For A's independence to be demonstrated, B must be True when executed, and C must be False when executed. For B's independence to be demonstrated, A must be True to execute B, and C must be False when executed. For C's independence to be demonstrated, either A or B must be False to execute C.

Examination of the covering set on the left of figure 21 shows that the independence of B is never demonstrated, because C returns a True result in the middle left figure. This results in True being the decision outcome when B is both True in the top left figure and False in the middle left figure. Examination of the covering set on the right of figure 21 shows that the independence of A is never demonstrated, because C returns a True result in the bottom right figure. This results in True being the decision outcome when A is both True in the upper right figure and False in the lower right figure.

The reduced test set size and the absence of demonstrating a condition's independence has an effect on the error detecting effectiveness of OBC versus MCDC. To demonstrate this, a small fault injection analysis was performed against the expressions that can be constructed of three nonrepeated conditions and the logical operators (AND, OR, and NOT) and the pattern families they fall within. A detailed example of the fault injection and analysis process is provided in appendix A. The first part of the analysis was to conduct the fault injection against the expressions in the two pattern families <3>(1T,2F) and <3>(2T,1F). Examples of expressions in

these two families are: (A OR B) AND C and (A AND B) OR C, respectively. The following faults, based on reference 12, were used:

- Operator reference faults (ORF)—each occurrence of an operator (AND and OR) is replaced with the other operator (e.g., OR for AND, and AND for OR).

- Variable negation faults (VNF)—each of the conditions is replaced with the complement of that condition (e.g., NOT A for A, A for NOT A).

- Expression negation faults (ENF)—each subexpression is replaced with the complement of that subexpression (e.g., NOT (A OR B) for (A OR B), and (A OR B) for NOT (A OR B)). This fault is equivalent to the variable negation fault for an operator instead of a condition.

- Associative shift faults (ASF)—each three-condition subexpression has the parenthesis shifted so that the subordinate operator becomes the dominant operator (e.g., A AND (B OR C) becomes (A AND B) OR C). Note that any of the conditions (A, B, or C) can be subexpressions themselves.

Injecting the above faults into each of the original expressions results in seven faulty expressions for each of the original expressions. How well MCDC and OBC detect these faults establishes a relative ranking for the two criteria. If MCDC and OBC are equivalent, then the scores for detecting the faults should be equivalent.

As mentioned previously, OBC requires three tests, while MCDC requires four tests for the expressions in the two pattern families <3>(1T,2F) and <3>(2T,1F). There is an inherent unfairness involved in comparing criteria that require different numbers of tests. The unfairness results in not knowing whether the criterion requiring more tests performs better because of the additional tests. To avoid this bias, every test set of four nonduplicate tests complying with OBC and every test set of four nonduplicate tests complying with MCDC of the original expressions was generated.

However, though the common practice is to use the same number of tests [12], it was discovered that these additional tests decreased the fault detecting capability of OBC. The results presented in the main body of the Report will follow the common practice to allow comparison with other fault injection studies. An analysis using the minimum sized test sets for OBC is provided in appendix B.

Note that any test set that complies with MCDC will also comply with OBC. This means that for certain expressions, OBC will have more complying test sets than MCDC will. These test sets were then run against the faulty expressions to determine which test sets would detect the fault and which tests would not.

Detection was said to occur if the faulty expression either returned a result different from the original expression (the standard practice for these kinds of experiments [12]), the MCDC test set did not execute an independence pair for one or more conditions (i.e., the executed test

vectors could not form independence pairs), or the OBC test set did not execute each condition both True and False. Inclusion of the coverage failure detection is needed to take into account the coverage analysis required by DO-178B [1]. The results are presented in figure 22.

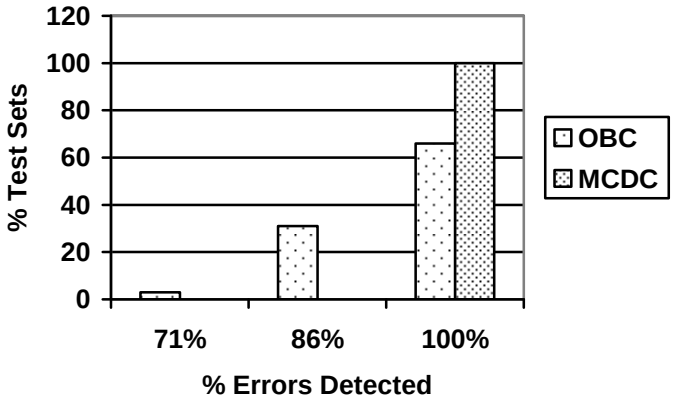


Figure 22. Modified Condition Decision Coverage Versus OBC Fault Detection—2(1T,2F) AND <3>(2T,1F) Families

In figure 22, OBC is shown as having more variability in the effectiveness of its test sets than MCDC. For MCDC, 100% of the test sets are shown as detecting 100% of the faults. For OBC, 66% of the test sets detect 100% of the faults, 31% of the test sets detect only 86% of the faults, and 3% of the test sets detect only 71% of the faults. In total, OBC detected 95% of all the faults.

The previous analyses looked at the expressions from two pattern families where OBC required fewer tests than MCDC and therefore presented a worst-case picture. However, even when the pattern family requires the same number of tests for OBC and MCDC, results show that the fault detection effectiveness of OBC and MCDC are not necessarily equivalent. To demonstrate this point, fault injection analysis was conducted against expressions in the pattern family <3>(2T,2F). Examples of expressions in this family are A OR (B AND C) and A AND (B OR C). The results from this analysis are presented in figure 23.

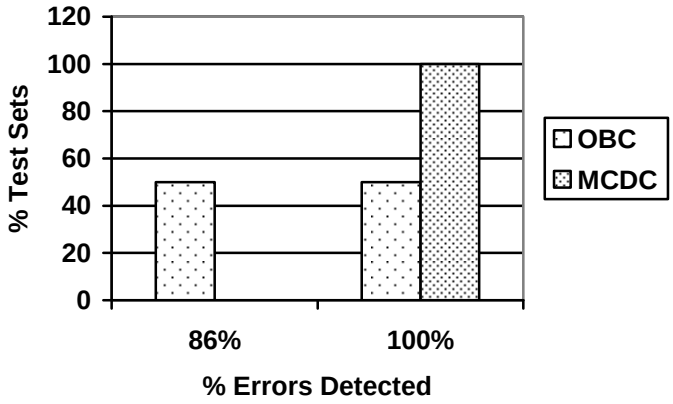


Figure 23. Modified Condition Decision Coverage Versus OBC Fault Detection—
<3>(2T,2F) Family

In figure 23, OBC is shown as having more variability in the effectiveness of its test sets than MCDC, but less variability than demonstrated in figure 22. For MCDC, 100% of the test sets are shown as detecting 100% of the faults. For OBC, 50% of the test sets detect 100% of the faults, while the other 50% of the test sets detect only 86% of the faults. In total, OBC detected 93% of all the faults for the pattern family analyzed in figure 23, less than that demonstrated in figure 22. It is important to note that even though the pattern family analyzed in figure 23 requires the same number of tests for OBC and MCDC, OBC demonstrated less effectiveness for the pattern family analyzed in figure 23 than for the pattern families analyzed in figure 22 where OBC required fewer tests than MCDC. This demonstrates that the fault detection effectiveness of OBC compared to that of MCDC is not necessarily a function of the required test set size.

Table 11 presents an analysis of the five pattern families for three-condition expressions. The first column identifies the pattern family. The second column identifies the percentage of expressions that fit within the family. The third column identifies the relationship of the minimum test set sizes for OBC and MCDC. The fourth column identifies whether OBC and MCDC require the same tests in their test sets. The fifth column identifies the relative fault detection capability between the OBC and MCDC test sets.

Table 11. Comparison of Three-Condition Pattern Families

Family	Percentage of Expressions	No. Tests?	Same Tests?	Fault Detection?
<3>(1T,2F)	12.5	OBC < MCDC	No	OBC < MCDC
<3>(1T,3F)	25	OBC = MCDC	Yes	OBC = MCDC
<3>(2T,1F)	12.5	OBC < MCDC	No	OBC < MCDC
<3>(2T,2F)	25	OBC = MCDC	No	OBC < MCDC
<3>(3T,1F)	25	OBC = MCDC	Yes	OBC = MCDC

The data in table 11 shows that for the majority of three-condition expressions (75%), OBC and MCDC require the same number of tests in a test set. This result is obtained by summing the entries in column two where the entry on the same row in column three shows the number of tests to be equal (rows 3, 5, and 6). However, the data in the table also shows that for half of the expressions, OBC is not equivalent to MCDC in fault detection capability. This result is obtained by summing the entries in column two where the entry on the same row in column five shows the fault detection of OBC to be less than MCDC (rows 2, 4, and 5). The fault injection analysis results for all three-condition expressions are presented in figure 24.

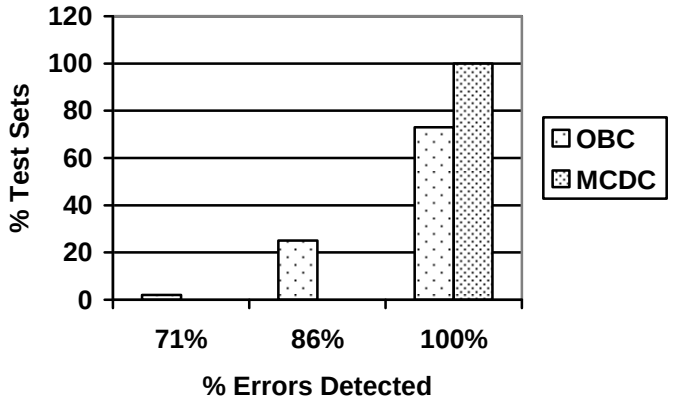


Figure 24. Modified Condition Decision Coverage Versus OBC Fault Detection—All Three-Condition Expressions

In figure 24, as expected, OBC is shown as having more variability in the effectiveness of its test sets than MCDC. For OBC, 73% of the test sets detect 100% of the faults, 25% of the test sets detect only 86% of the faults, and 2% of the test sets detect only 71% of the faults. In total, OBC detected 96% (95.74%) of all the faults.

3.4 BEYOND THREE-CONDITION DECISIONS.

Between the two- and three-condition decisions, the effectiveness of MCDC remained at 100%, while the effectiveness of OBC dropped from 100% to 96%. This raises the question of how OBC and MCDC will fare with higher levels of expression complexity. One way to gain some insight into the trend is to look at the proportion of expressions where MCDC and OBC test set sizes are equivalent as one moves to expressions of larger numbers of conditions. Figure 25 shows the proportion of expressions where MCDC and OBC test set sizes are equivalent, and where they are not, for expressions of one through six conditions.

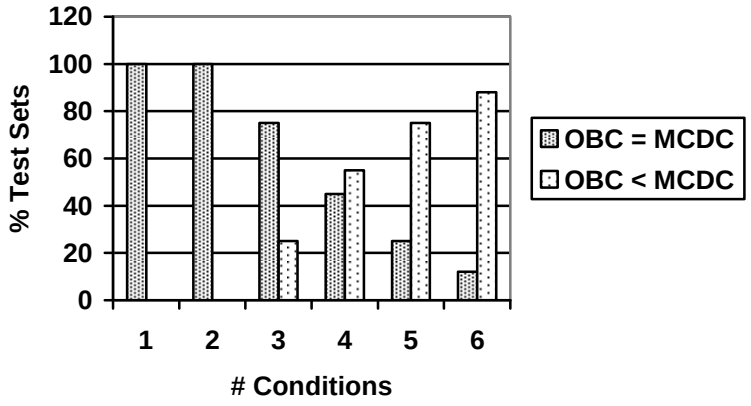


Figure 25. Modified Condition Decision Coverage Versus OBC Test Set Size Equivalence in Expressions

Figure 25 shows that for three-condition expressions, test set size for OBC is equivalent to MCDC for 75% of the expressions; for four-condition expressions, test set size for OBC is equivalent to MCDC for 45% of the expressions; for five-condition expressions, test set size for OBC is equivalent to MCDC for 25% of the expressions; and for six-condition expressions, test set size for OBC is equivalent to MCDC for 12% of the expressions. As the analysis for three conditions showed, only a percentage of the expressions where the test set size for OBC is equal to that for MCDC also gave equivalent fault detection capability. This declining trend of equivalence in test set size means that as expressions get more complex, the difference between OBC and MCDC increases. Figure 26 shows this declining trend of numbers of test sets with equivalent fault detection capability.

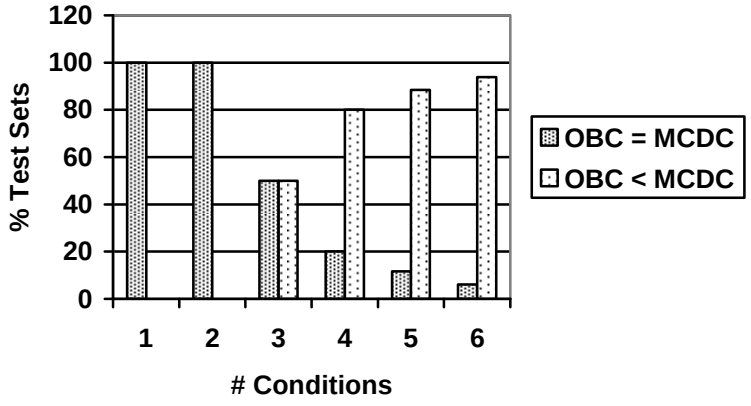


Figure 26. Modified Condition Decision Coverage Versus OBC Test Set Fault Detection Equivalence in Expressions

To confirm the projection that the difference between OBC and MCDC would worsen with more complex expressions, fault injection was conducted against four-condition expressions. The results are presented in figure 27.

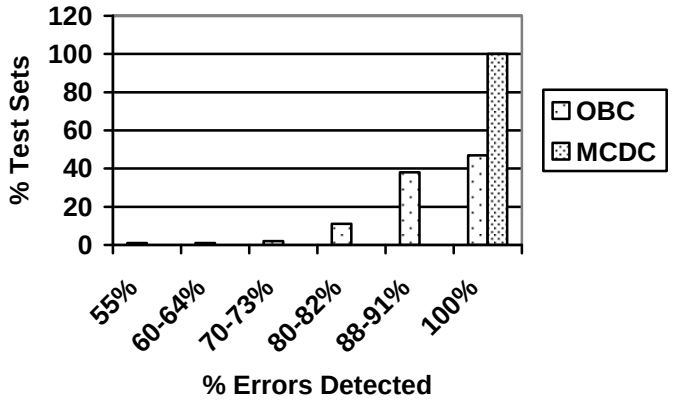


Figure 27. Modified Condition Decision Coverage Versus OBC Fault Detection—All Four-Condition Expressions

The results in figure 27, when compared with those in figure 24, show that the variability in the effectiveness of OBC has increased, and the overall effectiveness of OBC has decreased between three- and four-condition expressions. OBC detects only 93% of the total faults in four-condition expressions compared with 96% for the three-condition expressions.

Part of this reduction is because only 20% of the four-condition expressions require the same tests for both OBC and MCDC, compared with 50% for three-condition expressions. This shows that not only is the percentage of expressions requiring the same number of tests between OBC and MCDC is declining with complexity, but also that the percentage of these tests giving equivalent fault detection capability is also decreasing (44% compared with 60%).

Another part of this reduction is because at the four-condition level, OBC requiring the same tests as MCDC no longer gave equivalent fault detection capability. This was because certain faults could only be detected by the absence of an independence pair for a condition (i.e., a test set that should have provided an independence pair for a condition did not). Since OBC does not check for independence pairs, it did not detect the faults that MCDC did.

3.5 MODIFIED CONDITION DECISION COVERAGE VERSUS OBC CONCLUSIONS.

The analyses presented have shown that OBC and MCDC are not equivalent in the general case for expressions of three or more conditions.

During this study, it was determined that three patterns present in an expression would always cause OBC to not be equivalent to MCDC:

- Nonequivalence results when the subexpression on the LHS of an AND has multiple True paths. The OR operator always has two True paths; therefore, an OR as the LHS operand of an AND will always cause nonequivalence. Note that the patterns presented in figure 16 show that an AND operator can also have multiple True paths under certain conditions and thereby cause nonequivalence. The expression A AND (B OR C) used in this study is one such example.

- Nonequivalence results when the subexpression on the left of an OR has multiple False paths. The AND operator always has two False paths; therefore, an AND as the LHS operand of an OR will always cause nonequivalence. Note that the patterns presented in figure 16 show that an OR operator can also have multiple False paths under certain conditions and thereby cause nonequivalence. The expression A OR (B AND C) used in this study is one such example.

- Nonequivalence results when the expression contains four or more conditions using a left-associative expression tree (e.g., ((A OR B) OR C) OR D, ((A AND B) AND C) AND D) and the same operators (ANDs or ORs). Note that if these expressions used mixed operators (ANDs and ORs), they will exhibit one of the previous patterns. What this study has found is that if all the operators are the same (i.e., only ANDs or only ORs), equivalence between OBC and MCDC is not guaranteed.

If one is using OBC, expressions containing the above patterns will need to be identified and additional verification will need to be conducted to make the OBC analysis equivalent to MCDC. The analysis in this Report shows that the execution and confirmation of independence pairs determined from the SC is required to achieve this equivalence. Because the analysis conducted for this Report was only able to analyze expression of less than five conditions, there may yet be other patterns at higher levels of complexity (i.e., expressions with five or more conditions). However, the execution and confirmation of independence pairs determined from the SC will still provide equivalence between MCDC and OBC. Note that these results are equally applicable to both non-OOT and OOT software.

4. RESULTS.

This Report documents the results of an investigation into issues and acceptance criteria for the use of SCA at the SC versus OC or EOC levels within OOT in commercial aviation as required by Objectives 5-8 of Table A-7 in DO-178B. The intent of the SCA is to provide an objective assessment (measure) of the completeness of the requirements-based tests and supports the demonstration of the absence of unintended function.

This Report recommends the use of both SCC and OC or EOCC for the SCA objectives of DO-178B for OOT software. A number of OOT features were identified where proper coverage could only be obtained from the OC/EOC level as the compiler creates data structures and code for these features in the OC/EOC. These features are:

- Method tables
- Constructors and initializers

- Destructors, finalizers, and finally blocks

Satisfaction of MCDC from the OC or EOC perspective using OBC of short-circuited logic was found not to be equivalent to satisfaction of MCDC from the SC perspective. The analysis in this Report shows that the execution and confirmation of independence pairs determined from the SC is required to achieve this equivalence. Note that this result is equally applicable to both non-OOT and OOT software.

These two findings indicate that a mix of SCC and OCC/EOCC is needed for OOT software, especially for Level A software. This requires further study to determine the proper mix of SCC and OCC/EOCC for all three levels where structural coverage analysis is required by DO-178B (Levels A-C). If a combination of SCC and OCC/EOCC is undesirable, the coverage of the OC/EOC can be used with the addition of SC to OC/EOC traceability. This traceability would need to apply to Levels B and C software where it currently does not under DO-178B.

5. OBJECT-ORIENTED TECHNOLOGY VERIFICATION ISSUES REQUIRING FURTHER WORK.

As part of this task, OOT verification issues requiring further work were to be identified. During Phase 2, four issues were identified, and during Phase 3 (the current phase), one additional issue was identified. The four issues from the Phase 2 work are repeated here for completion of the task. Three of these issues are related to the guidance provided by Objective 8 in Table A-7 of DO-178B. Within this table, test coverage (confirmation) of data coupling and control coupling is required for software Levels A through C. The only difference identified for this objective in the table is that Level C software does not require independence. This is in contrast to the control-flow adequacy criteria of Objectives 5 through 7, where there are software level dependent differences. One of these issues relates to the proper coverage of polymorphism and dynamic dispatch. The final issue is related to the guidance provided by Objectives 5-7 in Table A-7 of DO-178B related to the use of both SC and OCC/EOCC for the SCA.

The first issue, from the Phase 2 Report, concerns the level of dependency coverage required by the two major coupling-based integration testing approaches. Both approaches adapted standard data-flow coverage criteria to apply interprocedurally, thus allowing for different levels of coverage. Both approaches were further adapted to apply to OOT. A follow-on study is recommended to determine if different levels of dependency coverage should be applicable to different software levels, just as different levels of control-flow coverage are applicable to different software levels. More detailed discussion and analysis may be found in the Phase 2 Report.

The second issue, also from the Phase 2 Report, concerns the level of dependency tracing required by the two major coupling-based integration testing approaches. One approach only requires dependencies concerning parameters between direct call-pairs to be covered, and the other more thorough approach requires all interprocedural dependencies to be covered. The recommended approach in the Phase 2 Report conforms to the more thorough analysis requiring all interprocedural dependencies to be covered. A follow-on study is recommended to determine if the alternate approach of requiring only dependencies concerning parameters between direct

call-pairs to be covered should be considered acceptable, and at what level. For example, the less thorough approach could be acceptable for software Levels B and C, where B requires independence, and the more thorough approach with independence could be required only for software Level A. More detailed discussion and analysis may be found in the Phase 2 Report.

The third issue, from both the Phase 2 Report and this Phase 3 Report, is considered major and concerns the level of coverage required for the adequate testing of polymorphism with dynamic binding and dispatch and the corresponding coverage of methods tables. Defining adequate testing of polymorphism with dynamic binding and dispatch is an active research area with no definitive answer yet. As such, the recommendation in the Phase 2 Report may only be considered an interim solution where polymorphism with dynamic binding and dispatch is concerned. Two major approaches emerged during the course of the Phase 2 study: the target-methods criterion (TMC) and the receiver-classes criterion (RCC) [7]. The majority of proposals studied in the Phase 2 Report achieve a subset of RCC. The recommended approach in the Phase 2 Report also achieves a subset of RCC; however, it is closer to TMC in application. A follow-on study is recommended to determine the acceptability of any of these approaches, and at what software level. More detailed discussion and analysis may be found in the Phase 2 Report.

The fourth issue, from the Phase 2 Report, concerns the cost and effectiveness of the coverage of intercomponent dependencies. Though studies have shown the cost and effectiveness of other dependency approaches, the specific approach recommended in the Phase 2 Report has not undergone such an analysis. A follow-on study is recommended to determine the cost and effectiveness of the coverage of intercomponent dependencies as recommended in the Phase 2 Report.

The fifth issue, from this Phase 3 Report, concerns the use of both SC and OCC/EOCC for the SCA objectives of DO-178B. A number of OOT features were identified where proper coverage could only be obtained from the OC/EOC level as the compiler creates data structures and code for these features in the OC or EOC. Satisfaction of MCDC from the OC and EOC perspective using OBC of short-circuited logic was found to be not equivalent to satisfaction of MCDC from the SC perspective. These two findings indicate that either a mix of SC and OCC/EOCC is needed for OOT software, especially for Level A software, or SC to OC/EOC traceability is required for all software levels. This requires further study to determine the proper mix of SC and OCC/EOCC and SC to OC/EOC traceability for all three software levels where SCA is required by DO-178B (Levels A-C).

6. REFERENCES.

Note that links were known to be correct when this Report was published.

1. "Software Considerations in Airborne Systems and Equipment Certification," Document No. RTCA/DO-178B, RTCA Inc., December 1, 1992.

2. Chilenski, J.J., Heck, D., Hunt, R., and Philippon, D., "Object-Oriented Technology (OOT) Verification Phase 1 Report—Survey Results," The Boeing Company.

3. "Final Report for Clarification of DO-178B "Software Considerations in Airborne Systems and Equipment Certification," Document No. RTCA/DO-248B, RTCA Inc., October 12, 2001.

4. Knickerbocker, J., "Object-Oriented Software—Object-Oriented Technology in Aviation (OOTiA) Survey," a presentation to the 2005 FAA Software/CEH Conference, July 2005.

5. Chilenski, J.J., Timberlake, T.C., and Masalskis, J.M., "Issues Concerning the Structural Coverage of Object-Oriented Software," FAA report DOT/FAA/AR-02/113, November 2002.

6. "Handbook for Object Oriented Technology in Aviation (OOTiA)," Revision 0, October 26, 2004, available at: http://faa.gov/aircraft/air_cert/design_approvals/air_software/oot/, last visited July 24, 2006.

7. Chilenski, J.J. and Kurtz, J.L., "Object-Oriented Technology Verification Phase 2 Report—Data Coupling and Control Coupling," FAA report DOT/FAA/AR-07/52, August 2007.

8. F. Gasperoni, "Safety Security and Object Oriented Programming," AdaCore, 8 rue de Milan, 75009 Paris, France., available at: http://www.adacore.com/category/developers-center/reference-library/technical-papers/?scat=10.

9. F. Siebert, **"**The Impact of Realtime Garbage Collection on Realtime Java Programming," ISORC'04—*The 7th IEEE International Symposium on Object-oriented Real-time distributed Computing*, May 12-14, 2004, Vienna, Austria.

10. Certification Authorities Software Team (CAST), Position Paper CAST-17, "Structural Coverage of Object Code," Completed June 2003, (Rev 3), available at: http://faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/.

11. Chilenski, J.J., "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion," FAA report DOT/FAA/AR-01/18, March 2001.

12. K. Kapoor, J.P. Bowen, "Experimental Evaluation of the Tolerance for Control-Flow Test Criteria," *Software Testing, Verification and Reliability*, Vol. 14, No. 3, September 2004, pp. 167-187.

APPENDIX A—FAULT INJECTION EXAMPLE

This appendix contains an example of fault injection analysis on the expression: "(A AND B) OR C)".

As mentioned in section 3.3 of the Report, the faults that are used in the experiment are:

- Operator reference faults (ORF)—each occurrence of an operator ("AND" and "OR") is replaced with the other operator (e.g., OR for AND, and AND for OR).

- Variable negation faults (VNF)—each of the conditions is replaced with the complement of that condition (e.g., NOT A for A, A for NOT A).

- Expression negation faults (ENF)—each subexpression is replaced with the complement of that subexpression (e.g., NOT (A OR B) for (A OR B), (A OR B) for NOT (A OR B)). This fault is equivalent to the variable negation fault for an operator instead of a condition.

- Associative shift faults (ASF)—each three-condition subexpression has the parenthesis shifted so that the subordinate operator becomes the dominant operator (e.g., A AND (B OR C) becomes (A AND B) OR C). Note that any of the conditions (A, B, or C) can be subexpressions themselves.

The expressions that result from the fault injection are given in table A-1. The first column in table A-1 gives a number to the expression. Expression 0 is the original expression. The second column identifies which fault is injected into the original expression. The third column provides the resultant expression.

Table A-1. Fault Injection Expressions

No.	Injected Fault	Resultant Expression
0	none	(A AND B) OR C
1	ORF: AND for OR	(A AND B) AND C
2	ORF: OR for AND	(A OR B) OR C
3	VNF: complement A	(NOT A AND B) OR C
4	VNF: complement B	(A AND NOT B) OR C
5	VNF: complement C	(A AND B) OR NOT C
6	ENF: complement AND	NOT (A AND B) OR C
7	ASF	A AND (B OR C)

Once the expressions that are to be used for the experiment are known, the truth tables for the expressions need to be generated. For three independent conditions (A, B, and C), there are eight (2^3) combinations of those conditions, as given in table A-2.

Table A-2. Truth Combinations for Three Conditions

Combination No.	A	B	C
0	False	False	False
1	False	False	True
2	False	True	False
3	False	True	True
4	True	False	False
5	True	False	True
6	True	True	False
7	True	True	True

In table A-2 the first column gives a number to the combination. This number is defined by interpreting the condition combinations as a binary number where False is 0 and True is 1. The second column gives the value for condition "A", the third column gives the value for condition "B", and the fourth column gives the value for condition "C".

The next step in the analysis is to determine the response given by each of the expressions to the condition combinations from table A-2. The results of this analysis are presented in table A-3.

Table A-3. Truth Combination Responses

Combination No.	Combination	0	1	2	3	4	5	6	7
0	(FFF)F	(FXF)F	(FXX)F	(FFF)F	(FFF)F	(FXF)F	(FXF)T	(FXX)T	(FXX)F
1	(FFT)T	(FXT)T	(FXX)F	(FFT)T	(FFT)T	(FXT)T	(FXT)F	(FXX)T	(FXX)F
2	(FTF)F	(FXF)F	(FXX)F	(FTX)T	(FTX)T	(FXF)F	(FXF)T	(FXX)T	(FXX)F
3	(FTT)T	(FXT)T	(FXX)F	(FTX)T	(FTX)T	(FXT)T	(FXT)F	(FXX)T	(FXX)F
4	(TFF)F	(TFF)F	(TFX)F	(TXX)T	(TXF)F	(TFX)T	(TFF)T	(TFX)T	(TFF)F
5	(TFT)T	(TFT)T	(TFX)F	(TXX)T	(TXT)T	(TFX)T	(TFT)F	(TFX)T	(TFT)T
6	(TTF)T	(TTX)T	(TTF)F	(TXX)T	(TXF)F	(TTF)F	(TTX)T	(TTF)F	(TTX)T
7	(TTT)T	(TTX)T	(TTT)T	(TXX)T	(TXT)T	(TTT)T	(TTX)T	(TTT)T	(TTX)T

In table A-3, the first column identifies the condition combination number from table A-2. The second column identifies the condition combination for the original expression in parenthesis along with the evaluation result for that combination. The third through tenth columns present the evaluation results for the short-circuited expression forms of expressions 0 through 7 from table A-1. Within table A-3, "T" is used for True, "F" is used for False, and "X" is used for "not executed".

The next step in the analysis is to determine the test sets that comply with modified condition decision coverage (MCDC) and object-code branch coverage (OBC). To ensure that there is no bias in the analysis given to a criterion that requires more tests, all test sets will be of the same size. Test set size follows the "N+1" rule for MCDC, where "N" represents the number of conditions in the expression and the test set needs to consist of N+1 tests to show the independence of all N conditions [A-1]. For this example, the test set sizes will consist of four (3+1) nonduplicate tests, where each test is one of the condition combinations from table A-2. There are 70 combinations of 8 things taken 4 at a time. Each combination needs to be examined to determine if it satisfies OBC or MCDC.

For the test set to satisfy OBC, all that is required is that at least one of the condition combinations for the original expression (expression #0) has a T entry for A, one has an F entry for A, one has a T entry for B, one has an F entry for B, one has a T entry for C, and one has an F entry for C. For example, consider the test set consisting of condition combinations (0, 1, 2, and 3) depicted in table A-4. Table A-4 is generated from table A-3 by removing the rows for condition combinations (4, 5, 6, and 7) and the columns for expressions (1, 2, 3, 4, 5, 6, and 7). Examination of the third column of table A-4 shows that the test set (0, 1, 2, and 3) will not satisfy OBC because none of those combinations provide a T for A or a T or F for B because B is never executed.

Table A-4. (0, 1, 2, and 3) Test Set

Combination No.	Combination	0
0	(FFF)F	(FXF)F
1	(FFT)T	(FXT)T
2	(FTF)F	(FXF)F
3	(FTT)T	(FXT)T

The test set consisting of condition combinations (0, 1, 4, and 6) depicted in table A-5, however, does satisfy OBC. Table A-5 is generated from table A-3 in the same manner as table A-4 was generated. Examination of the third column of table A-5 shows that condition combinations (0 and 1) provide an F for A, condition combinations (4 and 6) provide a T for A, condition combination (4) provides an F for B, condition combination (6) provides a T for B, condition combinations (0 and 4) provide an F for C, and condition combinations (1 and 6) provide a T for C.

Table A-5. (0, 1, 4, and 6) Test Set

Combination No.	Combination	0
0	(FFF)F	(FXF)F
1	(FFT)T	(FXT)T
4	(TFF)F	(TFF)F
6	(TTF)T	(TTX)T

Note that the condition combination (0) is an extra test as the test set consisting of condition combinations (1, 4, and 6) is sufficient for OBC. This is one instance where an additional nonduplicate test is present in the test set so that the OBC test sets contain the same number of nonduplicate tests as MCDC does.

For the test set to satisfy MCDC, there must be at least two condition combinations that provide an independence pair for A, two condition combinations that provide an independence pair for B, and two condition combinations that provide an independence pair for C. For example, the test set consisting of condition combinations (0, 1, 2, and 3) depicted in table A-4 will not satisfy MCDC as condition combinations (0 and 2) provide the F half of the independence pair for A,

but there is no T half, and there are no condition combinations for either half of the independence pair for B because B is never executed. The test set consisting of condition combinations (0, 1, 4, and 6) depicted in table A-5 does satisfy MCDC as condition combinations (0 and 6) provide the independence pair for A, condition combinations (4 and 6) provide the independence pair for B, and condition combinations (0 and 1) and (1 and 4) each provide independence pairs for C.

For this example, of the 70 possible test sets of 4 nonduplicate condition combinations, 12 will satisfy MCDC and 32 will satisfy OBC. Note that the test sets that satisfy MCDC will also satisfy OBC so, for this example, OBC can be satisfied with 20 test sets that do not also satisfy MCDC.

The next step in the analysis is to determine whether MCDC or OBC will detect the faults in the expressions with injected faults. Detection of the fault occurs in one of three ways. The first way, common to both MCDC and OBC, is when the expression returns a result that differs from the correct expression. For example, consider the test set consisting of condition combinations (0, 1, 4, and 6) depicted in table A-6. Table A-6 is generated from table A-3 by removing the rows for condition combinations (2, 3, 5, and 7). In addition, where a function response is different from the required response in column 2, the incorrect response is shown in lowercase. Examination of table A-6 shows that all of the faulty expressions have at least one lowercase (incorrect) response; therefore, the test set (0, 1, 4, and 6) detects the faults in all expressions for both MCDC and OBC when using the incorrect response of expressions.

Table A-6. Test Set (0, 1, 4, and 6) Responses

Combination No.	Combination	0	1	2	3	4	5	6	7
0	(FFF)F	(FXF)F	(FXX)F	(FFF)F	(FFF)F	(FXF)F	(FXF)t	(FXX)t	(FXX)F
1	(FFT)T	(FXT)T	(FXX)f	(FFT)T	(FFT)T	(FXT)T	(FXT)f	(FXX)T	(FXX)f
4	(TFF)F	(TFF)F	(TFX)F	(TXX)t	(TXF)F	(TFX)t	(TFF)t	(TFX)t	(TFF)F
6	(TTF)T	(TTX)T	(TTF)f	(TXX)T	(TXF)f	(TTF)f	(TTX)T	(TTF)f	(TTX)T

The second way a fault can be detected is if the OBC satisfying test set fails to provide a T and F execution for each condition in the faulty expression. For example, consider the test set consisting of condition combinations (0, 3, 4, and 7) depicted in table A-7. Table A-7 is generated from table A-3 by removing the rows for condition combinations (1, 2, 5, and 7).

Table A-7. Test Set (0, 3, 4, and 7) Responses

Combination No.	Combination	0	1	2	3	4	5	6	7
0	(FFF)F	(FXF)F	(FXX)F	(FFF)F	(FFF)F	(FXF)F	(FXF)T	(FXX)T	(FXX)F
3	(FTT)T	(FXT)T	(FXX)F	(FTX)T	(FTX)T	(FXT)T	(FXT)F	(FXX)T	(FXX)F
4	(TFF)F	(TFF)F	(TFX)F	(TXX)T	(TXF)F	(TFX)T	(TFF)T	(TFX)T	(TFF)F
7	(TTT)T	(TTX)T	(TTT)T	(TXX)T	(TXT)T	(TTT)T	(TTX)T	(TTT)T	(TTX)T

Examination of table A-7 shows that functions (1 and 6) have no F entry for C, and functions (2 and 7) have no T entry for C. This shows that OBC using the (0, 3, 4, and 7) test set will detect the faults in faulty expressions (1, 2, 6, and 7) when using the nonexecution of conditions.

The third way a fault can be detected is if the MCDC satisfying test set fails to provide the correct vectors for independence pairs for one or more conditions in the faulty expression when executed. For example, consider again the test set consisting of condition combinations (0, 3, 4, and 7) depicted in table A-7. For A's independence to be demonstrated, B when executed must be T and C when executed must be F. This leads to the two vectors (condition combinations plus required result) ((TTX)T, (FXF)F) to demonstrate A's independence. Examination of table A-7 shows that function (4) provides only the (FXF)F vector and functions (5 and 7) provide only the (TTX)T vector. Since none of the faulty functions provide both of the required vectors, this shows that MCDC using the (0, 3, 4, and 7) test set will detect the faults in all faulty expressions when using the nonexecution of independence pairs.

The next step is to combine each of the fault detection steps into a fault detection result for each test set. For example, consider again the test set consisting of condition combinations (0, 3, 4, and 7) depicted in table A-7. Examination of table A-7 shows that expressions (1, 2, 4, 5, 6, and 7) provide incorrect responses. Combining this result with the nonexecution of conditions result of (1, 2, 6, and 7) for OBC results in OBC detecting the faults in expressions (1, 2, 4, 5, 6, and 7). Combining the incorrect response results with the nonexecution of independence pairs result of (1, 2, 3, 4, 5, 6, and 7) for MCDC results in MCDC detecting the faults in expressions (1, 2, 3, 4, 5, 6, and 7).

The final step is to combine the fault detection results for all test sets. For this example, all 12 test sets for MCDC detect all 7 faults. This means that for the 84 faults (7*12) presented to MCDC, 100% were detected. For OBC, 21 of the 32 test sets (66%) detect all 7 faults (100%), 10 of the test sets (31%) detect only 6 of the 7 faults (86%), and 1 test set (3%) detects only 5 of the 7 faults (71%). This means that for the 224 faults (7*32) presented to OBC, 212 (7*21 + 6*10 + 1*5) or 94.64% were detected. Figure A-1 presents the fault detection results.

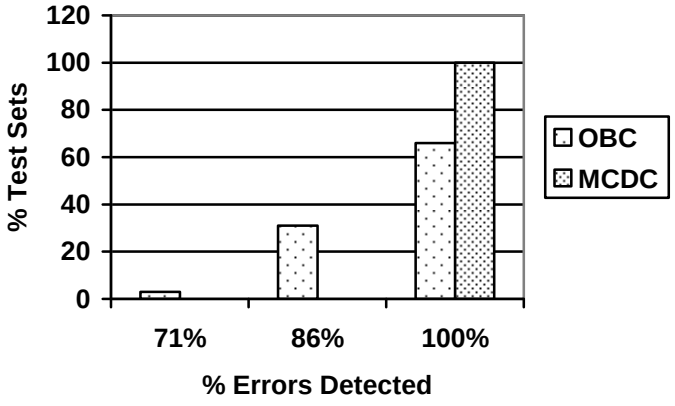


Figure A-1. Modified Condition Decision Coverage Versus OBC Fault Detection—
(A AND B) OR C

For the full fault injection experiment, the above needs to be performed for all expressions. Table A-8 provides summary information for the full experiment.

Table A-8. Fault Injection Experiment Information

No. of conditions	3	4
No. of expressions	128	2560
No. of injected faults/expression	7	11
No. of MCDC test sets/expression	8..12	64..240
No. of OBC test sets/expression	8..32	64..1936

REFERENCE

A-1 Chilenski, J.J., "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion," FAA report DOT/FAA/AR-01/18, March 2001.

APPENDIX B—FAULT INJECTION WITH MINIMUM-SIZED OBJECT-CODE BRANCH COVERAGE TEST SETS

This appendix contains the results when minimally sized test sets for object-code branch coverage (OBC) are used. Figure B-1 is the equivalent of figure 24 for the three-condition analysis, and figure B-2 is the equivalent of figure 27 for the four-condition analysis.

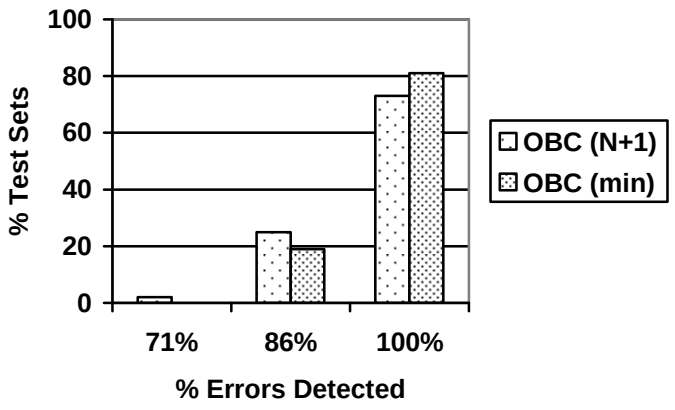


Figure B-1. Object-Code Branch Coverage Fault Detection—All Three-Condition Expressions

In figure B-1, OBC is shown with less variability and more effectiveness when the minimally sized test sets are used than when the "N+1" sized test sets are used. The minimally sized test sets detected 97.12% of all the faults, and the N+1 sized test sets detected 95.74% of all the faults.

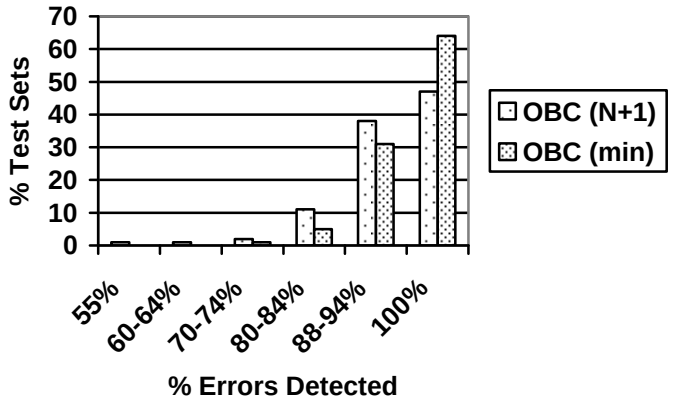


Figure B-2. Object-Code Branch Coverage Fault Detection—All Four-Condition Expressions

In figure B-2, OBC is again shown with less variability and more effectiveness when the minimally sized test sets are used than when the N+1 sized test sets are used. The minimally sized test sets detected 95.85% of all the faults, and the N+1 sized test sets detected 93.22% of all the faults.