DOT/FAA/AR-06/54

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

Software Verification Tools Assessment Study

June 2007

Final Report

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

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-06/54		

4. Title and Subtitle	5. Report Date
SOFTWARE VERIFICATION TOOLS ASSESSMENT STUDY	June 2007
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Viswa Santhanam, John Joseph Chilenski, Raymond Waldrop, Thomas Leavitt, and Kelly J. Hayhurst	

9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
NASA Langley Research Center Hampton, VA 23681-2199	
	11. Contract or Grant No. DTFA03-96-X-90001 NAS1-00106 task order 1008 NAS1-00079

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
U. S. Department of Transportation Federal Aviation Administration Air Traffic Organization Operations Planning Office of Aviation Research and Development Washington DC 20591	Final Report
	14. Sponsoring Agency Code AIR-120

15. Supplementary Notes

The Federal Aviation Administration Airport and Aircraft Safety COTR was Charles Kilgore.

16. Abstract

The Software Verification Tools Assessment Study (SVTAS) was a research effort to investigate criteria for effectively evaluating structural coverage analysis tools for use on projects intended to comply with RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment." The objective for developing these criteria was to improve the accuracy and consistency of the qualification process for these tools. The SVTAS included surveys of commercially available structural coverage analysis tools, literature searches to review policy and guidance relevant to structural coverage and tool qualification, and analysis of structural coverage definitions. Based on these findings, a test suite was proposed as an approach to increase objectivity and uniformity in the application of the tool qualification criteria. Goals for a full test suite were defined, and a prototype test suite was developed. The prototype test suite was run on three different structural coverage analysis tools to evaluate the efficacy of the test suite approach. The prototype test suite identified anomalies in each of the three coverage analysis tools, demonstrating the potential for a test suite to help evaluate a tool's compatibility with the DO-178B requirements.

17. Key Words	18. Distribution Statement
Structural coverage, DO-178B, Tool qualification, Modified condition decision coverage	This document is available to the 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	139	

TABLE OF CONTENTS

LIST OF TABLES

ACRONYMS AND ABBREVIATIONS

BBM	Black-box MCDC
CAST	Certification Authorities Software Team
CCM	Coupled-cause MCDC
DP	Discussion paper
FAA	Federal Aviation Administration
FAQ	Frequently asked question
MCDC	Modified condition decision coverage
MSM	Masking MCDC
NASA	National Aeronautics and Space Administration
OBC	Object-code branch coverage
OCC	Operator coverage criterion
RC/DC	Reinforced condition/decision coverage
SVTAS	Software Verification Tools Assessment Study
UCM	Unique-cause MCDC
USM	Unique-cause + masking MCDC

EXECUTIVE SUMMARY

The RTCA/DO-178B document, "Software Considerations in Airborne Systems and Equipment Certification," is the primary means used by aviation software developers to obtain Federal Aviation Administration (FAA) approval of airborne computer software. DO-178B describes software life cycle activities and design considerations, and enumerates sets of objectives for the software life cycle processes. The objectives for structural coverage analysis are often considered to be onerous, in part, because meeting these objectives is labor-intensive and costly. In recent years, a number of commercial tools have been developed to ease the burden of structural coverage analysis.

The DO-178B requirements for structural coverage analysis have been the subject of considerable debate within the aviation software community. The debate generally concerns the proper definition and implementation of various forms of structural coverage, particularly decision coverage and modified condition decision coverage. These concerns impact the ability to dependably assess structural coverage analysis tools used in a DO-178B context.

Because of increasing dependence on structural coverage analysis tools to meet the DO-178B objectives, the FAA Software and Digital Systems Safety technical community identified a need for improved methods for evaluating these tools. The Software Verification Tools Assessment Study (SVTAS) was initiated to develop specific criteria that could be used to determine whether the performance of a structural coverage analysis tool on a project is acceptable in the context of DO-178B. The objective was to promote consistent and correct assessment of coverage analysis tools.

This report describes the results of the research activities in the SVTAS. The research activities comprised two phases. Phase 1 included tool surveys, literature searches, and analysis of structural coverage definitions. Two surveys of commercial structural coverage tools identified 31 tools in the marketplace that perform some level of structural coverage analysis. A literature search identified policy and guidance from the commercial aviation industry relevant to structural coverage and tool qualification and comparable tool qualification requirements in nonaviation industries. A number of ambiguities in definitions and use were identified and possible resolutions suggested. Based on the findings in Phase 1, a test suite was determined to be the most effective approach to increase objectivity and uniformity in the application of the tool qualification criteria. That is, passing the tests in the test suite would provide a reasonable degree of assurance that a tool assesses structural coverage consistent with the DO-178B objectives.

In Phase 2, goals for a test suite were defined and a prototype test suite was developed. The prototype consists of a subset of the tests defined for the full test suite. The prototype test suite was run on three different structural coverage analysis tools to determine the efficacy of the test suite. The testing identified anomalies in each of the three coverage analysis tools, demonstrating the potential for a test suite to help evaluate whether a tool's performance is consistent with the DO-178B requirements for structural coverage.

1. INTRODUCTION.

The DO-178B document, "Software Considerations in Airborne Systems and Equipment Certification," [1] is the primary means used by aviation software developers to obtain Federal Aviation Administration (FAA) approval of airborne computer software. DO-178B describes software life cycle activities and design considerations, and enumerates sets of objectives for the software life cycle processes. The objectives serve as a focal point for approval of the software.

This report considers one particular set of objectives in DO-178B, namely, the verification objectives for structural coverage analysis. Coverage, in the context of DO-178, refers to the extent to which a given verification activity has satisfied its objectives. Coverage measures can be applied to any verification activity, although they are most frequently applied to testing activities. Structural coverage analysis determines how much of the code structure was executed by the requirements-based tests and establishes traceability between the code structure and the test cases. This report assumes a fundamental understanding of structural coverage and DO-178B.

In recent years, a number of commercial tools have been developed to ease the burden of structural coverage analysis. When automated tools are used for credit in the verification of software in compliance with DO-178B objectives, those tools are subject to qualification criteria. Section 12.2 of DO-178B states that qualification of a tool is needed when processes in DO-178B are eliminated, reduced, or automated by the use of a software tool when its output is not verified. The objective of tool qualification is to ensure that tools provide confidence at least equivalent to that of the processes eliminated, reduced, or automated. Qualification requirements differ depending on whether a tool is a development tool (whose output is part of the software) or a verification tool (that cannot introduce errors, but may fail to detect them). For this study, attention is focused specifically on structural coverage analysis tools, which are verification tools per DO-178B.

A number of concerns are relevant to the qualification of structural coverage analysis tools. The DO-178B objectives for structural coverage analysis are often considered to be onerous, in part, because meeting those objectives is labor-intensive and costly [2]. At least some of the difficulty stems from an on-going debate and misunderstanding about definitions of structural coverage metrics. This debate combined with increased dependence on coverage tools has given rise to concerns about the adequacy of existing tool qualification criteria to ensure consistent and effective evaluation of tools. The following statement from a discussion paper (DP) in DO-248B [3] summarizes the issue: "The task of comparing the tool function definitions against the DO-178B/ED-12B objectives becomes more difficult to achieve if those objectives are not readily understood (for instance, modified condition decision coverage (MCDC)). If the party responsible for selecting the tool misunderstands the relevant DO-178B/ED-12B objectives, they will not be able to determine whether or not they have been implemented correctly within the tool."

Despite various attempts to clarify definitions related to structural coverage analysis [3-10], misunderstandings about fundamental aspects of structural coverage have continued to arise about how to consistently interpret and implement analysis criteria in tools and how to evaluate those tools for qualification purposes. The intent of this study was to identify some of the areas

of misunderstanding and develop evaluation criteria with the goal of facilitating consistent assessment of structural coverage analysis tools.

1.1 APPROACH.

The Software Verification Tools Assessment Study (SVTAS) was initiated to develop specific evaluation criteria that could be used to determine whether the performance of a structural coverage analysis tool is consistent with the structural coverage objectives in DO-178B. This study was conducted in two phases with the following activities:

- Phase 1 activities:
  - Reviewed current tool available and compared their characteristics
  - Documented and summarized research related to structural coverage and tool qualification
  - Developed guidelines or methods (process and criteria) for assessing structural coverage analysis tools within the DO-178B context
- Phase 2 activities:
  - Applied the guidelines from Phase 1 to commercial verification tools on the market
  - Documented lessons learned from the evaluation
  - Modified the guidelines based on the lessons learned for use by the FAA and aviation industry applicants

This report documents the results of both phases. Because much of this report is taken directly from preliminary reports from this study [11-14], references to them are omitted in the text, except in circumstances where the reference would be particularly useful.

1.2 DOCUMENT STRUCTURE.

This report is organized as follows.

Section 2 discusses the results of surveys conducted to identify commercially available structural coverage analysis tools.

Section 3 provides an overview of the literature review results about structural coverage and tool qualification.

Section 4 discusses nuances of structural coverage definitions and details of how structural coverage metrics are implemented that contribute to different concepts of structural coverage. Ten specific issues are discussed and resolutions suggested with respect to the evaluation criteria.

Section 5 gives the requirements for a test suite for structural coverage analysis tools. Categories of tests are described for Levels A, B, and C.

Section 6 discusses a prototype test suite based on the requirements described in section 5. The prototype test suite was evaluated with three structural coverage analysis tools. The results of the evaluation are presented.

Section 7 summarizes the results of the SVTAS.

A few formatting points are worth noting.

- When referring to programming language statements within the normal text of the document, boldface lowercase words such as **if** and **case** are used.

- When referring to an expression or condition within the normal text of the document, boldface text is used. Examples are **A and B** and **not A**.

- When referring to variable names and conditions, boldface text is used and capital letters are used for the first character of the name or condition as well as for the first character after an underscore; all other letters are lowercase. Examples are **A**, **T1**, and **Flight_Mode**. Note that a handful of exceptions are made for variable names in example code that is explicitly coded in the C language.

2. STRUCTURAL COVERAGE ANALYSIS TOOLS.

A number of structural coverage analysis tools can be found in the commercial marketplace today. Some structural coverage analysis tools are offered individually, while others are offered as one of many capabilities in a fully integrated development environment. Some tools are marketed specifically to the aviation industry with DO-178B in mind, while others are marketed to a broader audience of software developers. In any case, the number of structural coverage analysis tools available today has grown significantly over the past decade.

The Boeing Company and SRI International independently conducted surveys using different approaches to collect information on structural coverage analysis tools available in the commercial marketplace. Boeing conducted a mail-out (e-mail) survey of prospective tool vendors to identify structural coverage analysis tools and their basic characteristics. SRI reviewed publicly available information on various structural coverage analysis tools and corresponded with a subset of the tool vendors to gather additional information about tool use and capabilities.

Table 1 shows the combined results of information collected from the Boeing and SRI surveys. This information includes tool name and vendor information (current as of February 2006), type of coverage measured, and level where coverage is measured (source or object code). A blank in the column under Type of Coverage Measured indicates that a particular coverage type is not measured. Information that was not collected or not confirmed is indicated by "?." Additional information collected through the surveys is available in references 11 and 14.

The survey results are not intended to endorse any tool nor to claim that these are the only tools available. The primary purpose of the surveys was to get a general understanding of structural coverage analysis tools that are commercially available.

Table 1. Directory of Structural Coverage Analysis Tools

Tool Name (Vendor, Web site)	Type of Coverage Measured				Where Coverage is Measured: Source or Object Code
	Statement	Decision	MCDC	Other	
AdaTest 95 (IPL Information Processing Ltd., www.ipl.com)	√	√	√	√	Source
Aprobe and RootCause (OC Systems, Inc., www.ocsystems.com)	√		√		Source and Object
ASMCover (Avionyx, Inc., www.avionyx.com)	√	√	√		Source and Object
BEACON AUTT (Applied Dynamics International, www.adi.com)	√	√	√	√	Source
BEACON for Simulink Tester (Applied Dynamics International, www.adi.com)	√	√	√	√	Source
C++Test 2.1 (Parasoft, www.parasoft.com)		√	√	√	Source
Cantata++ (IPL Information Processing Ltd., www.ipl.com)	√	√	√	√	Source
BullseyeCoverage (C-Cover) (Bullseye Testing Technology, www.bullseye.com)		√		√	Source
Clover (Cenqua, www.cenqua.com)	√	√		√	Source
CodeTEST (Metrowerks Corporation, www.metrowerks.com)	√	√	√		Source and Object
CoverageScope (Real-Time Innovations, www.rti.com)		√		√	Source
CTC++ (Testwell Oy, Ltd., www.testwell.fi)	√	√	√	√	Source
DACS Object Coverage (DDC-I, Inc., www.ddci.com)	√	√	√		Object
G-Cover (Green Hills Software, Inc., www.ghs.com)	√	√	√	√	Object
Hindsight (Tester's Edge, Inc., www.testersedge.com)	√	√	√	√	Source
IBM Rational Test RealTime (IBM, www.ibm.com)	√	√	√		Source
J Cover (Codework, Inc., www.codework.com)	√	√			Source

Table 1. Directory of Structural Coverage Analysis Tools (Continued)

Tool Name (Vendor, Web site)	Type of Coverage Measured			Other	Where Coverage is Measured: Source or Object Code
	Statement	Decision	MCDC		
McCabe IQ or McCabe Test (McCabe & Associates, Inc., www.mccabe.com)	√	√	√	√	Source
Adacover from ObjectAda Real-Time RAVEN (Aonix, www.aonix.com)		√	√		Object
Panorama, Panorama-2 OO-Test (Internationall Software Automation, Inc., www.softwareautomation.com)		√		√	Source?
Reactis Tester (Reactive Systems, Inc., www.reactive-systems.com)		√		√	Design
Simulink Performance Tools Model Coverage (The MathWorks, Inc., www.mathworks.com)		√	√	√	Design
Telelogic Tau Logiscope Test-Checker (Telelogic, www.telelogic.com)		√	√	√	Source
TCMON—Test Coverage Monitor for Ada (Testwell Oy, Ltd., www.testwell.fi)	√			√	Source
Rational Test RealTime, Rational PurifyPlus RealTime (IBM, www.ibm.com)	√	√	√	√	Source
TestMate (IBM, www.ibm.com)		√	√	√	Source (for MCDC) and object (block & branch coverage)
TestQuest Pro (TestQuest, Inc., www.testquest.com)				√	?
TestWorks/Coverage TCAT (Software Research, Inc., www.soft.com)	√	√	√	√	Source
Vector*CAST*/Cover (Vector Software, Inc., www.vectorcast.com)	√	√	√		Source
VerOCode (Verocel, Inc., www.verocel.com)	√	√	√		Object

According to the survey, there are a significant number of tools in the commercial marketplace, including many that claim to assess structural coverage per DO-178B. Also, a majority of the tools perform coverage analysis on source code, many by inserting software probes. Based on the survey results, a test suite was determined to be a reasonable means to bring uniformity and objectivity to the application of tool qualification criteria.

3. LITERATURE REVIEW.

A literature review was included in this study to identify (1) guidance and policy in the aviation software industry relevant to the SVTAS goal of improving the qualification process for structural coverage analysis tools, (2) potential areas of ambiguity in the guidance that may affect qualification, and (3) regulatory guidance outside of the aviation industry that may assist in meeting the SVTAS goal. A more detailed listing of the results of the literature review are given in reference 14.

3.1 THE FAA-RELATED GUIDANCE DOCUMENTS.

A number of documents influence assessment of structural coverage. Some of these documents are official guidance documents that are referred to in this report as primary guidance documents. Other documents provide further descriptions of some aspects of structural coverage, but are not recognized by the FAA as regulatory. These documents, referred to as auxiliary documents, include position papers, job aids, tutorials, and other publicly available documents. The next four sections discuss the primary and auxiliary documents for structural coverage and tool qualification from within the aviation industry.

3.1.1 Primary Guidance Documents for Structural Coverage.

DO-178B is the primary source of guidance for applicants seeking certification of airborne systems and equipment with software content. Section 6.4 of the document addresses verification of software through testing. The primary objectives of structural coverage analysis are to verify satisfaction of the software requirements and to determine what software structures were not exercised by the requirements-based test procedures. The latter objective of structural coverage analysis is commonly achieved through automated verification tools. Three forms of structural coverage and other relevant terms are described in the glossary of DO-178B.

- Statement coverage: Every statement in the program has been invoked at least once.
- Decision coverage: Every point of entry and exit in the program has been invoked at least once, and every decision in the program has taken on all possible outcomes at least once.
- Condition: A Boolean expression containing no Boolean operators.
- Decision: A Boolean expression composed of conditions and zero or more Boolean operators. If a condition appears more than once in a decision, each occurrence is a distinct condition.
- Modified condition decision coverage (MCDC): Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently

affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.

The description of MCDC given in DO-178B is often referred to as unique cause MCDC (UCM) because of the requirement to show a condition's independence by varying just that condition while holding fixed all other possible conditions. An alternative form of MCDC, called masking MCDC (MSM), has gained acceptance by regulatory authorities, but is only discussed in auxiliary documents such as reference 8.

Table A-7 in Annex A of DO-178B prescribes the levels of coverage that must be achieved based on the criticality level of the software. Level A software must meet statement coverage, decision coverage, and MCDC criteria; Level B must meet statement and decision coverage criteria; and Level C must meet the statement coverage criterion. No structural coverage criteria are prescribed for Level D software.

In DO-178B, structural coverage analysis criteria are stated at the source code level using code-level elements such as statements, Boolean expressions, and operators. However, DO-178B also calls for additional analysis at the object code level for Level A software if the compiler generates object code that is not directly traceable to source code statements. The exact nature of this analysis is not specified. Source to object code traceability is the subject of a Certification Authorities Software Team (CAST) position paper discussed in the following section.

3.1.2 Auxiliary Guidance Documents for Structural Coverage.

In addition to DO-178B, a number of other documents, including DO-248B and various CAST position papers, serve to provide additional information. This information is not considered official regulatory guidance.

DO-248B [3] was published in October 2001 to address questions raised about DO-178B by both the industry and certification authorities. According to the FAA, DO-248B provides information rather than new guidance, since DO-248B is intended to clarify DO-178B. DO-248B contains information on a wide range of issues, including answers to frequently asked questions (FAQ) and DPs relevant to structural coverage analysis. The FAQs and DPs provide information relevant to a number of issues for structural coverage analysis tools. The following FAQs and DPs are relevant to this study.

- FAQ #42: Can structural coverage be demonstrated by analyzing the object code instead of the source code? Then, can a compiler be used to simplify the analysis?
- FAQ #43: What is the intent of structural coverage analysis?
- FAQ #44: Why is structural testing not a DO-178B/ED-12B requirement?
- DP #3: The differences between DO-178A/ED-12A and DO-178B/ED-12B guidance for meeting the objective of structural coverage

- DP #7: Definition of commonly used verification terms
- DP #8: Structural coverage and safety objectives
- DP #12: Object code to source code traceability issues
- DP #13: Definitions of Statement Coverage, Decision Coverage, and Modified Condition Decision Coverage (MCDC)

In addition to the information from DO-248B, CAST has produced a number of position papers related to structural coverage analysis.

- CAST Position Paper 10, "What is a "Decision" in Application of Modified Condition Decision Coverage (MCDC) and Decision Coverage (DC)?" [7]:

  This position paper outlines the certification authorities' position that all Boolean expressions in a program are subject to decision coverage or MCDC. This position is consistent with a literal interpretation of the term decision as any Boolean expression rather than as a mechanism that causes control flow branching.

- CAST Position Paper 6, "Rationale for Accepting Masking MCDC in Certification Projects" [8]:

  MCM offers an alternative to UCM in the form of coverage that concentrates on the observability of the effect of operators at the output. In MCM, a condition is shown to independently affect a decision's outcome by applying principles of Boolean logic to ensure that no other condition influences the outcome, even though many conditions may change value at once. This position paper calls for the recognition of MCM as an acceptable method of complying with DO-178B.

- CAST Position Paper 12, "Guidelines for Approving Source Code to Object Code Traceability" [15]:

  This paper provides guidelines for assessing an applicant's source code to object code traceability and verification activities and results when a review of Level A software is performed. This information is applicable if the compiler generates object code that is not directly traceable to the source code.

- CAST Position Paper 17, "Structural Coverage of Object Code" [9]:

  This paper presents certification authorities' concerns and position regarding the analysis of structural coverage at the object-code level rather than at the source-code level, particularly for Level A software. This paper addresses verification of assumptions made about compiler behavior and assurance of coding standards.

Finally, three other references are relevant to the discussion of structural coverage analysis.

- "A Practical Tutorial on Modified Condition Decision Coverage" [4]:

  Motivated by the difficulties encountered by the software community in understanding and applying MCDC, the FAA and the National Aeronautics and Space Administration (NASA)-sponsored development of an MCDC tutorial. The tutorial is designed to guide users through a five-step process for constructing test cases to achieve MCDC for a given Boolean expression. The method described provides an alternative form of coverage to UCM and is more widely applicable than the latter. The concepts are elaborated in a subsequent paper by two of the authors [5].

- "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion" [6]:

  This work by Chilenski compares UCM to MSM and a variant that combines the notions of unique-cause and masking versions, called Unique Cause + Masking MCDC (USM). Of these, Chilenski concludes that MSM should be preferred because it is the simplest to achieve in the general case and does not seem to be any less effective in finding coding errors.

- "Applicability of Modified Condition Decision Coverage to Software Testing" [10]:

  This paper is the first published reference on MCDC. The paper introduces and defines MCDC with respect to Boolean expressions, and rationalizes its use in safety-critical applications.

It is also worth noting that there is a web site hosted by Dr. Sergiy Vilkomir at the University of Wollongong, Australia, dedicated to MCDC: www.dsl.uow.edu.au/~sergiy/MCDC.html [16]. The web site contains pointers to more than 50 links with online information on MCDC, including government and industry reports, standards, and journal and conference papers.

3.1.3 Primary Guidance Documents for Tool Qualification.

When a software tool is used to eliminate, reduce, or automate a DO-178B process without its output being verified, tool qualification is the means used to ensure that the tool provides confidence at least equivalent to the process being eliminated, reduced, or automated. Section 12.2 of DO-178B outlines broad requirements for qualifying tools that automate the software development or verification process. Tools that automate the software development process and whose output is part of the airborne software are subject to more rigorous qualification requirements than those that automate verification processes. Subsection 12.2.1 of DO-178B discusses the qualification criteria for development tools.

For tools that assist with verification and whose output is not part of the airborne software, the qualification criteria stated in section 12.2.2 of DO-178B are rather terse: "The qualification criteria for software verification tools should be achieved by demonstration that the tool complies with its Tool Operational Requirements under normal operational conditions." The concise nature of this guidance prompted the specification of more measurable criteria for

qualifying software tools. Such additional guidance is provided in the form of the FAA Notice N8110.83 issued in April 1999, and reissued as Notice N8110.91 [17] in January 2001. This latter Notice expired in 2002.

The information from the above notices was subsumed in Chapter 9 of FAA Order 8110.49, which was approved in 2004 [18]. In particular, section 9-6 of FAA Order 8110.49 provides guidance in several topics, including (a) determining when a tool needs to be qualified, (b) the nature of qualification data needed for development and verification tools, (c) what are acceptable tool operational requirements, and (d) how to verify that the tool satisfies its operational requirements.

3.1.4 Auxiliary Guidance Documents for Tool Qualification.

DO-248B contains some information related to tool qualification in the following FAQs and DPs.

- FAQ 61: What constitutes a development tool and when should it be qualified?

- FAQ 65: In DO-178B/ED-12B sections 12.3.3.4 (Tool Qualification for Multiple-Version Dissimilar software) and 12.3.3.5 (Multiple Simulators and Verification) what is meant by "equivalent software verification process activities"?

- DP 1: Verification Tool Selection Considerations

- DP 11: Qualification of a Tool Using Service History

- DP 1 contains the following statement that summarizes much of the motivation for this report:

  > "The task of comparing the tool function definitions against the DO-178B/ED-12B objectives becomes more difficult to achieve if those objectives are not readily understood (for instance, Modified Condition Decision Coverage – MCDC). If the party responsible for selecting the tool misunderstands the relevant DO-178B/ED-12B objectives, they will not be able to determine whether or not they have been implemented correctly within the tool."

This statement confirms some of the difficulties associated with structural coverage analysis tools. FAQs 61 and 65 and DP 11 discuss various aspects of tool qualification, but are not directly relevant to the development of evaluation criteria for structural coverage analysis tools. Other references relevant to tool qualification include the following.

- Khanna, V., "Software Tool Qualification," *Proceedings of the FAA National Software Conference,* Dallas, Texas, May 2002 [19].

- Khanna, V., "Software Tool Qualification," *Proceedings of the FAA National Software Conference,* Reno, Nevada, September 2003 [20].

- Hayhurst, K. and Rierson, L., "Verification Tools," *Proceedings of the FAA National Software Conference*, Dallas, Texas, May 2002 [21].

- Kornecki, Andrew J. and Zalewski, Janusz, "The Qualification of Software Development Tools From the DO-178B Certification Perspective," *CrossTalk, The Journal of Defense Software Engineering*, April 2006 [22].

3.2 REVIEW OF INDUSTRY GUIDANCE.

As part of the literature review, regulatory guidance and standards outside of the aviation industry were reviewed. Guidance documents, related to either software verification or tools, from the following organizations were surveyed.

- Institute of Electrical and Electronics Engineers
- Institution of Electrical Engineers
- International Electrotechnical Commission
- International Standards Organization
- Motor Industry Software Reliability Association
- Directorate of Standardization of the United Kingdom Ministry of Defence
- U.S. Nuclear Regulatory Commission

Many standards surveyed from other industries do not specifically address tool qualification or assessment, and those that do often cite DO-178B. Some standards require structural coverage, but MCDC seems to be particular to the aviation industry. Consequently, information from standards outside of the aviation community provides little assistance in resolving the problem of consistent and adequate evaluation of structural coverage analysis tools. A list of the guidance documents reviewed for this study is included in reference 14.

4. ISSUES FOR AUTOMATING STRUCTURAL COVERAGE ANALYSIS.

DO-248B, CAST position papers, and auxiliary research reports concerning various aspects of structural coverage have attempted to clarify the requirements for structural coverage in DO-178B. However, some questions still remain about nuances of definitions and about implementation of structural coverage metrics. These nuances can result in significant variation in the qualification and performance of structural coverage analysis tools. Based on the review of the primary and auxiliary documents discussed in section 3, a number of issues were identified. Those issues are discussed below, followed by recommended resolutions with respect to evaluation criteria.

The scope of this study does not include resolution of all misunderstanding about structural coverage. The following issues, however, were determined to be important to the development of a test suite.

4.1 STATEMENT COVERAGE.

DO-178B defines statement coverage as "Every statement in the program has been invoked at least once."

The term statement itself is not defined in DO-178B, but it is relatively well understood within the software community. In programs written in an assembly language, each machine instruction is a statement. In programs written in a high-level procedural language (as opposed to functional programming languages such as Prolog), statements are understood to be units of computation, a sequential ordering of which comprises the program. In procedural languages often used in airborne applications, such as C, C++, and Ada, a simple delimiter, such as a semicolon, is used to set simple statements apart. When flow of control must be selected based on current data values, compound statements, such as **if**, **case**, and **loop** statements, are used. Coverage for compound statements is understood to be coverage for the portion of the statement that evaluates the flow-controlling expression(s) and coverage for all other statements (applied recursively) contained within the compound statement. DO-248B, DP #13 "Definitions of Statement Coverage, Decision Coverage, and Modified Condition Decision Coverage (MCDC)" [3], provides an informal definition along these lines.

4.1.1 Coverage of Implicit Statements (Issue 1).

In some source code languages, notably Ada, a return from a subprogram call need not be coded explicitly as a **return** statement. If only explicitly written statements are subject to coverage analysis, it is possible to demonstrate 100% coverage for programs that do not exercise all the implicit **return** statements, as the following example illustrates.

```
if A > 0 then
        B := A;
        return;
end if;
return;          -- implicit return, not written at source code level
```

A single invocation of the **if** statement with **A** = 1, for example, will seem to provide statement coverage for the subprogram if the implicit return following the **if** statement is not considered in statement coverage analysis. In a language that requires all returns to be explicitly coded, it would require at least two executions of the subprogram to achieve statement coverage.

Suggested resolution for Issue 1: For evaluating structural coverage analysis tools, statement coverage criterion should not apply to implicit returns and other forms of implicit statements (e.g., an implicit **default** case in C's **switch** statement). The rationale for this is that coverage of such statements has been deferred to the decision coverage criterion in other circumstances. For example, an **if** statement without an **else** clause could be considered as one with an implicit **else null** clause, but statement coverage is widely accepted as not requiring coverage of such implied **else** clauses.

4.1.2 Coverage of Declarative Statements (Issue 2).

Most programming languages make a distinction between declarative statements and executable statements. Ordinarily, the term statement is taken to mean executable statements, rather than declarative statements, because the declarative statements do not generate machine instructions that execute at run time in most cases. Exceptions to this rule occur when declarative statements do more than declare types and allocate space for variables. For example, an object declaration in Ada or C that includes initialization will likely generate executable instructions to accomplish the initialization at run time. Run time executable code can arise from more subtle cases. For example, an object declaration in Ada can generate implicit initialization code if the type of the object implies such initializations (e.g., access types are automatically initialized to **null** in Ada, as are record types with initialized components). The issue here is whether declarative statements should be subject to statement coverage.

Suggested resolution for Issue 2: For evaluating structural coverage analysis tools, declarative statements should be subject to statement coverage. The rationale is that, in general, declarative statements could produce executable code. Not subjecting declarative statements to statement coverage could exempt them from all other forms of coverage required for higher levels of criticality. For example, if declarative statements are not subject to statement coverage, an applicant could conclude that Boolean expressions occurring in declarative statements are not subject to MCDC, which could encourage coding practices that exploit this loophole to reduce testing effort. Coverage could be demonstrated through analysis for declarative statements that do not generate executable code.

4.2 DECISION COVERAGE.

DO-178B defines decision coverage as "Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken all possible outcomes at least once." A decision is defined as "A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition." Five issues were identified relevant to decision coverage.

4.2.1 Defining Boolean Expressions (Issue 3).

To effectively evaluate decision coverage, a tool must be able to identify Boolean expressions. There are no glossary terms in DO-178B for a Boolean expression or Boolean operator. In languages with direct support for the Boolean type, such as Ada, decisions can be recognized as expressions of that type without regard to the context in which the expression occurs. In C, the notion of a Boolean type is intertwined with ordinary integer types, and contextual information is required to distinguish a Boolean expression from other integer expressions. For example, bool-expr in each of the following contexts is readily recognized as a Boolean expression:

if (bool-expr) …
else if (bool-expr) …
bool-expr ? *expression-t* : *expression-f*
bool-expr && *bool-expr*

bool-expr || *bool-expr*
!*bool-expr*

Outside of these contexts, however, an integer expression cannot be easily distinguished as a Boolean expression. In particular, subprogram parameters and the right-hand sides of assignments are two contexts where the distinction may not be evident without some analysis of the surrounding contexts.

Suggested resolution for Issue 3: For evaluating structural coverage analysis tools, a Boolean expression should be defined as any expression with two classes of values, commonly designated as False and True, and which (1) is determined to be Boolean using the rules of the programming language, (2) is used to control program flow based on its value, or (3) derives its value, directly or indirectly, from other binary-valued expressions and external sources. Using this definition, expressions can be determined to be Boolean if they are used in certain ways, even if they are not declared explicitly as Boolean.

Because Ada defines Boolean as a primitive data type, declarations and type analysis are normally sufficient to identify all Boolean expressions. However, if non-Boolean data types are used to represent Boolean entities, the contextual rules can be used to identify the intent. Consider the following example.

```
type SWITCH_TYPE is (OFF, ON);
INT_ENABLED : SWITCH_TYPE;
:
if INT_ENABLED = ON then
        :
else
        :
end if;
```

Because INT_ENABLED is of a two-valued discrete type and its values are used to control program flow, it should be treated like a Boolean variable.

ANSI C does not define Boolean as a primitive data type. Use of a macro such as BOOLEAN is optional, but cannot be relied on for finding all Boolean expressions. However, the following indicators can be used to identify all Boolean expressions. The first indication that an expression is Boolean is when it is used as an operand of a **&&**, ||, or **!** operator, or if it is used as an actual parameter whose formal parameter has been recursively identified to be Boolean. The second indication that an expression is Boolean is when it is used to determine flow control as part of **if** statements, **while** loops, or other conditional expressions. Finally, if the right-hand side expression is determined to be Boolean, then the left-hand side variable in an assignment statement is also determined to be Boolean.

4.2.2 Boolean Constants (Issue 4).

It is not clear whether Boolean expressions encompass variables as well as constants such as False and True and 0 and 1. In most cases, the test procedures will not be able to change the

values of Boolean constants during the test procedure in a meaningful way. Therefore, it is necessary to clarify the impact of such constants in an expression that is subject to decision coverage or MCDC.

Suggested resolution for Issue 4: For evaluating structural coverage analysis tools, the phrase Boolean expression should be interpreted as applying to variable Boolean expressions and not to Boolean constants.

4.2.3 Branch Coverage Versus Decision Coverage (Issue 5).

Despite clarification in CAST Position Paper 10 [7] on the definition of decision, debate about decision coverage continues, in part because the CAST position differs from the common definition of decision coverage. The CAST paper recommends that all decisions, regardless of whether they directly impact control flow or not, be subject to applicable coverage criteria. This is based partly on the fact that a Boolean expression evaluated outside of a flow control construct may indeed be saved and used to control program flow later. Subjecting all decisions to the applicable coverage criteria will ensure that the intermediate computation technique is not used to avoid subjecting complex Boolean expressions to required coverage criteria.

In the software engineering community, decision coverage is often synonymous with branch coverage, where branch coverage requires that every branch in the program be taken at least once (that is, a decision occurs at a location that affects control flow). Branch coverage is less stringent than the DO-178B description of decision coverage. Some structural coverage analysis tools may not make a distinction between branch coverage and decision coverage.

Suggested resolution for Issue 5: For evaluating structural coverage tools, it may be helpful to make a distinction between decisions that directly impact program flow and those that do not. This would allow the evaluation to separately determine whether branch coverage (defined as every point of entry and exit in the program has been invoked at least once and every branch within the program has been taken at least once) is evaluated accurately and whether decision coverage (as per CAST Position Paper 10) is evaluated accurately. This may be beneficial if changes in structural coverage requirements are made in revisions of DO-178B.

4.2.4 Exception Handling (Issue 6).

It is not clear how coverage criteria should be applied to exception handling. Some languages, notably Ada, provide for programmed exception handling. That is, the programmer can designate a sequence of statements, called a handler, to execute in the event of an exceptional condition arising from another set of statements, called the scope of the handler. The occurrence of an exception within the scope of the handler causes a branch in program flow, transferring control from the point of exception to the top of the handler. For example, consider the following code segment.

```
begin
        statement-1
        statement-2
        :
```

```
        statement-n
exception
when Constraint_Error =>
        handler-CE
when others =>
        handler-other
end;
statement-m
```

Here, handler-CE and handler-other are exception handlers, handling constraint errors and other exceptions, respectively. Statement-1 through statement-n are in the scope of these handlers. That is, a constraint error arising from any of these statements will cause control to be transferred to handler-CE; any other exception arising from them will cause a branch to handler-other. When an exception handler finishes, control is transferred to statement-m, skipping over any other exception handlers below it. In the absence of any exception arising from the statements in the scope of the handlers, control will branch around all exception handlers, from statement-n to statement-m.

If any of the statements in the scope of the handlers calls a subprogram and an exception is raised after the declarative part of that subprogram has been elaborated, the action taken depends on whether or not another handler was defined in the called subprogram for that exception. If a handler for that exception was defined, that handler will be entered rather than one of the handlers shown above. If no handler for that exception was defined or if the exception is raised during the elaboration of the called subprogram's declarative part, the appropriate handler in the calling subprogram (shown above) will be entered.

Suggested resolution for Issue 6: For evaluating structural coverage tools, the requirement to invoke every point of entry and exit in the program should also include exception handlers. It is debatable whether one entry per exception handler is a sufficient level of testing. A single test to exercise the entry point of each exception handler as a minimum is commensurate with the requirement for the normal entry and exit points of the program. The actual number of test cases necessary to properly evaluate the exception handling function depends on the nature and complexity of the exception handler.

4.2.5 Expressions With Short-Circuit Operators (Issue 7).

It is not clear how coverage should be applied to short-circuit operators. A Boolean operator is said to be a short-circuit operator if an expression involving such an operator is evaluated one operand at a time and the evaluation of any additional operands is stopped as soon as the outcome of the expression is determined. Consider the expression **A and B**. If viewed as a function **and (A, B)**, both operands **A** and **B** may need to be evaluated prior to determining the outcome[1]. However, if **A** evaluates to False, the outcome of the expression will be False, regardless of the value of **B**. In the short-circuit evaluation model, the expression is evaluated to

[1] In Ada, the Boolean arguments must be evaluated prior to the call. In other languages, the evaluation may be deferred until the function body warrants it.

False without attempting to evaluate **B**. Although this model could be applied to a wide range of Boolean operators, only the **and** and **or** operators are commonly evaluated using this model.

In ANSI C, the evaluation model applicable to operators **&&** and || is always the short-circuit evaluation model. In Ada, both the short-circuit evaluation model and the traditional evaluation model are supported, but with distinct syntaxes. The traditional model is represented by the operators **and** and **or**, whereas the short-circuit evaluation model is represented by the constructs **and then** and **or else**. In fact, the Ada language refers to short-circuit evaluation constructs as short-circuit control forms, reflecting the fact that these constructs represent control flow within the expression.

The conditional expression construct in C also represents control flow within an expression. When all three operands are Boolean, the conditional expression construct can be considered a Boolean operator. The conditional expression construct takes the form (condition **?** if-true-expression **:** if-false-expression) and always results in the evaluation of its first operand and exactly one of the other two operands. With this conditional evaluation of its second and third operands, the conditional expression raises many of the same issues as the short-circuit operators described above.

To understand the implications of these operators on decision coverage, consider the following equivalents for the short-circuit operators and the C conditional expression.

`if A and then B then …`	`if A then`
	`  Temp := B;`
	`else`
	`  Temp := False;`
	`end if;`
	`if Temp then …`

`if A or else B then …`	`if A then`
	`  Temp := True;`
	`else`
	`  Temp := B;`
	`end if;`
	`if Temp then …`

`if (A ? B : C) {…}`	`if (A) {`
	`  Temp := B;`
	`} else {`
	`  Temp := C;`
	`}`
	`if (Temp) {…}`

These equivalents show that a decision that includes a short-circuit operator or a C conditional expression can be considered to be split into multiple separate decisions by the operator in question.

Suggested resolution for Issue 7: For evaluating structural coverage analysis tools for decision coverage, the short-circuit operators and the C conditional expression should be considered as splitting a decision in which they occur into separate decisions, each of which must be covered separately.

4.3 MODIFIED CONDITION DECISION COVERAGE.

DO-178B defines MCDC as, "Every point of entry and exit in the program has been invoked at least once, every condition in a decision has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions."

This definition levies four distinct requirements:

a. Every point of entry and exit in the program has been invoked at least once.

b. Every decision in the program has taken all possible outcomes at least once.

c. Every condition in a decision has taken all possible outcomes at least once.

d. Each condition in a decision has been shown to independently affect that decision's outcome.

In general, MCDC is a relatively complicated metric. Assessing MCDC with a tool adds further complications.

4.3.1 Defining a Boolean Operator (Issue 8).

The term Boolean operator is used in DO-178B without a glossary entry to define the term. One question that arises is whether user-defined functions that take Boolean arguments to produce a Boolean result should be treated as operators for MCDC purposes. Much of the work related to structural coverage of Boolean expressions deals with standard binary operators, such as **and** (**&&**), **or** (||), and **xor**, which operate on two Boolean operands to produce a Boolean result. The unary operator **not** (**!**) is rarely mentioned because it operates on a single Boolean operand and produces a Boolean result. Because the result is simply the opposite of the operand value, it is possible to deal with the negated entity as just another variable.

Some languages allow standard operators to be invoked using a unique syntax. For example, Ada permits the expression **A and B** to be written as **and (A, B)**, a syntax similar to a user-defined function call. The regulatory guidance should make it clear that the syntax of how an operator is invoked does not change the way the operator is treated for MCDC.

Suggested resolution for Issue 8: For evaluating structural coverage tools, the term operator should apply only to built-in or predefined operators (i.e., not user-defined functions) whose operands are Boolean and whose result is Boolean. In Ada, for example, in addition to **and**, **or**, and **xor**, the relational operators =, /= (same as **xor**), <, <=, >, and >= all qualify as operators for MCDC purposes.

This suggested resolution could raise a concern that applicants might attempt to hide predefined operators in user-defined functions to avoid being subject to MCDC. The user-defined function

that incorporates the predefined operation, however, would be subject to MCDC. Further, implementing complex requirements as compositions of smaller, tractable functions is a sound software engineering practice that could make achieving MCDC easier.

The above position also helps answer another common question from an Ada viewpoint. If a predefined operator is overloaded[2] with a user-defined function, is the operator still an operator for MCDC purposes? The answer is no, because it is not the syntax that determines whether an operator is an operator for MCDC purposes, but it is whether or not the operator represents a predefined Boolean function.

4.3.2 Expressions With Short-Circuit Operators (Issue 9).

In Issue 7, short-circuit operators were discussed with respect to decision coverage. For MCDC, short-circuit operators impact the requirement to show the independent effect of each condition.

DO-248B DP #13 suggests that decisions consisting of short-circuited[3] conditions be treated as if each short-circuited condition were a separate decision. Using the example from DO-248B, the decision in

if a < 0 and then c = 1 then..

would be treated as if it were two decisions of the form:

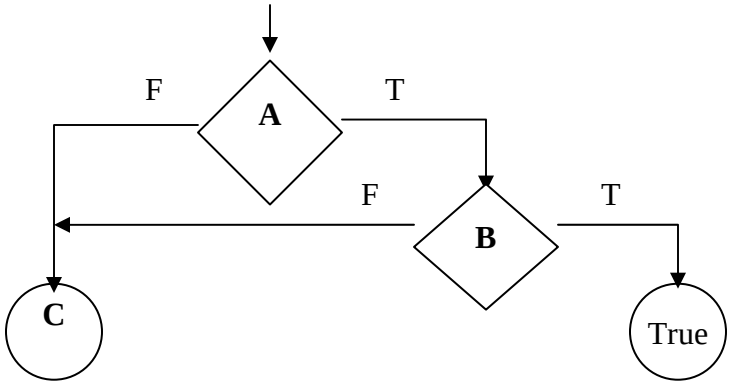
if a < 0 then
if c = 1 then..

Coverage now applies to each decision within each **if** statement. The net effect is that the number of test cases and their general form are the same as if the short-circuit operator were replaced with **and** instead of **and then**.

Unfortunately, the equivalence of the two models breaks down, with respect to number of test cases, when combinations of short-circuit operators are nested in more complex expressions. Consider the decision consisting of three conditions, **A**, **B**, and **C**: (**A and then B**), **or else C**. The flow-chart below illustrates the control flow between the terms of this expression.

[2] An operator is said to be overloaded in Ada if there is more than one meaning for that operator, depending on the types of its operands and the expected type of the result. While most operators come overloaded, as they do in many other programming languages, Ada operators can be overloaded by user-defined functions, and may have a totally new meaning given by the user.

[3] The term short-circuited operand is used to refer to the operand **A** and operand **B** of a short-circuit expression of the form **A and then B** or **A or else B**.



A minimum of four test cases are needed to achieve MCDC for this expression. However, when each condition is treated as a separate decision, MCDC, including branch coverage between short-circuited conditions, can be achieved with just three test cases: (**A**=F, **B**=?, **C**=F), (**A**=T, **B**=F, **C**=T), and (**A**=T, **B**=T, **C**=?). Thus, the use of short-circuit operators leads to a simplification of the task of achieving MCDC. At least one study suggests that this simplification may come at the cost of readability and maintainability [23].

In reference 10, a different approach is suggested in which the short-circuit operators are treated like their traditional counterparts (**and then** like **and**, and **or else** like **or**), where each term must be demonstrated as having independent effect on the outcome. This requires test vector pairs in which the condition of interest and the outcome are the only entities to toggle in value, while the other variables are fixed at the same value in both vectors, or one of the values is irrelevant. A Boolean variable whose value does not influence the outcome is commonly referred to as a don't-care and specified with a ? in test vectors. For example, the independent effect of **B** on the outcome can be demonstrated by pairing up (**A**=T, **B**=F, **C**=F/Outcome=F) with (**A**=T, **B**=T, **C**=?/Outcome=False); the fact that **C** is a don't-care in the second test vector allows it to be paired with any other vector that meets all other conditions.

To see that the approach from reference 10 differs from the one suggested by DO-248B, observe that the test set consisting of the three test cases (**A**=F, **B**=?, **C**=F), (**A**=T, **B**=F, **C**=T), and (**A**=T, **B**=T, **C**=?) does not demonstrate the independence of variable **B** nor that of **C**. Also, observe that this test set does not detect the variable negation fault in the expression (**A and then not B**) **or else C**. Consequently, the approach in reference 10 may be considered a stronger form of coverage for short-circuit operators than the DO-248B approach.

Suggested resolution for Issue 9: The first recommendation is that continued assessment is needed to resolve issues with coverage of expressions with short-circuit operators. This topic should be considered in RTCA/SC-205, Software Considerations in Aeronautical Systems. The second recommendation is that the DO-248B approach should be an acceptable approach for coverage of decisions with short-circuited operators[4], keeping in mind the problems noted with complex expressions. For simple expressions, the DO-248B approach is comparable to the

[4] It is important to note that this recommendation was not a consensus recommendation among the authors of this report.

approach in reference 10. Further, if the source code is modified from the short-circuit expression to a set of nested **if** statements, the MCDC criteria for the new structure will match that of the DO-248B suggested approach.

This recommendation may favor the use of short-circuit operators since using these operators might reduce the number of test cases needed to demonstrate MCDC compared with equivalent expressions that do not use short-circuited operators. This is somewhat ironic, because the use of traditional operators is often favored over short-circuit operators because the absence of branches in the object code leads to improved performance on the modern processors based on pipelined instruction stream architecture.

Another point worth noting is that while some Boolean binary operators like /=[5] and = cannot be short-circuited (i.e., one operand is never sufficient to determine the value of the expression), there are other binary operators that could be short-circuited similar to the way **and** and **or** are commonly short-circuited. For example, each of the relational operators <, <=, >, and >=, when applied to Boolean operands, can be short-circuited. When using such operators, should the applicant be allowed to use the approach suggested in DO-248B? The suggestion is that the weaker criterion should be accepted only if it can be established that the compiler actually generates a short-circuited evaluation sequence for such operators.

With respect to the C conditional expression, it should be noted that because of the semantics of this construct, the three operands should be treated as separate decisions, and coverage of the construct is obtained if coverage of the three decisions is individually obtained. Further, for the conditional expression **(A ? B : C)**, if operand **A** is a condition rather than a decision, coverage of **A** is subsumed by coverage of **B** and **C**, because **A** must take on both True and False values to obtain that coverage.

4.3.3 Multiple Occurrences of a Condition (Issue 10).

Dealing with multiple occurrences of a condition within a decision is a primary source of confusion about MCDC. The DO-178B description of a condition states that each occurrence must be treated as a separate condition. That requirement conflicts with the statement in the description of MCDC that a condition's independence is established by varying just that condition while holding fixed all other possible conditions and showing that the decision's outcome is affected. If each occurrence of a condition is a separate condition, varying one occurrence while holding fixed the other occurrences of the same condition is not possible.

There are two generally accepted methods of complying with the independence requirement—unique cause and masking. As described in reference 8, unique cause is the original approach to showing the independent effect of a condition mentioned in the description of MCDC in the DO-178B glossary. In the unique-cause approach, only the values of the condition of interest and the decision's outcome can change between the two test cases in an independence pair—everything else must remain the same. Holding the value of every other condition fixed ensures that the one condition that changed value is the only condition that influences the value of the decision's

[5] This "not-equal-to" operator is the same as exclusive-**or**, denoted in Ada as **xor**. There is no logical exclusive-**or** operator defined in C.

outcome. The logic of the decision does not need to be examined to determine that the condition of interest is solely responsible for the change in the value of the decision's outcome.

In contrast, masking allows more than one input to change in an independence pair, as long as the condition of interest is shown to be the only condition that affects the value of the decision outcome. However, analysis of the internal logic of the decision is needed to show that the condition of interest is the only condition causing the value of the decision's outcome to change. MSM can be applied to decisions with coupled conditions, where unique-cause cannot be applied. However, the requirement with masking for analysis of the internal logic of a decision presents significant challenges to building a tool to do this.

In addition to the unique cause and masking approaches, there are a number of variations on MCDC—some that have been approved on projects and some that are more theoretical in nature. A brief description of these variations is given in appendix A. As part of this study, Boeing compared and contrasted the variants of MCDC described in appendix A. That comparison is available in reference 12.

Suggested resolution for Issue 10: No single approach to treating multiple occurrences of a condition should be designated as the only acceptable approach. Different approaches may be acceptable, depending on the context. In any case, the approach should be explicitly reported in planning documents that address structural coverage activities. Understanding how a tool treats multiple occurrences of a condition will be important to establishing whether the tool evaluates MCDC properly.

Because masking has been approved as an acceptable approach to showing independent effect, other approaches may also be considered acceptable. For example, consider Coupled Cause MCDC (CCM). This criterion requires that each condition in the decision be shown to affect the outcome of the decision while all other uncoupled conditions are held fixed. This variant essentially interprets all other possible conditions to mean all other uncoupled conditions, because coupled conditions cannot be held (or guaranteed to be held) fixed while varying a condition to which they are coupled.

For example, consider the decision **A or (not A and B)**. The following test vectors provide coverage:

a. (F, T, F/F)
b. (F, T, T/T)
c. (T, F, F/T)

Test vector pair (a, b) demonstrates that condition **B** can independently impact the decision, and the pair (a, c) serves to demonstrate the independent impact of the condition **A** as well as **not A**. Because the conditions **A** and **not A** are coupled, the term independent impact must be interpreted as independent relative to other uncoupled conditions.

CCM overcomes a weakness of the unique cause approach, namely, the inability to handle coupled conditions. It may also be easier for a tool to check the coupled-cause approach than masking.

4.4 STRUCTURAL COVERAGE CRITERIA FOR ASSEMBLY LANGUAGE PROGRAMS.

Virtually all discussions regarding structural coverage in DO-178B are based on source code constructs that are typical of high-order languages, such as statements, Boolean expressions, conditions, and decisions. Few discussions address the application of structural coverage criteria when the source code language is an assembly language. A primary problem with assembly level programs is that they deal with very primitive data types, such as integers, floating-point numbers, pointers, and blocks of contiguous storage locations, rather than the more abstract data types such as Boolean, fixed-point numbers, records, and arrays. DO-178B coverage criteria such as MCDC assume that the more abstract data types are readily identifiable in the source code.

A second problem arises because of the limited range of operations the hardware might support. For example, on most contemporary machines, there is no distinction between a Boolean **and** and a bit-wise **and**. Furthermore, an **and** operation may have nothing to do with a logical operation at all; it could be used to extract a packed field from a record object, or as a modulo arithmetic operation (e.g., **mod** 256). This makes it impractical to perform coverage analysis on the source code of an assembly program for coverage metrics that depend on the ability to distinguish Boolean entities and Boolean operations. However, statement and branch coverage criteria can be mapped to assembly code coverage without the need for such distinction. Table 2 summarizes a recommended approach to map assembly code coverage to DO-178B coverage criteria.

Table 2. Approach to Assembly Code Coverage

DO-178B Coverage Criterion	Assembly or Machine Code Coverage Level
Statement coverage	Instruction coverage
Decision coverage	Instruction coverage + entry point coverage, including interrupt and exception handlers + branch coverage of all branching instructions
MCDC	Branch coverage + MCDC as it applies to an abstraction (e.g., pseudocode) of assembly code

At least one commercial coverage analysis tool uses object-code branch coverage (OBC) as the basis for satisfying Level A structural coverage requirements. Questions that arise from this include: Is OBC always an acceptable alternative to (or equivalent to) MCDC? If not, under what circumstances are the two equivalent? Examples exist where OBC may not imply decision

coverage—a necessary condition for MCDC. The simplest of those examples is the code generated for a simple assignment statement of the form: **A** := **B**.

Most compilers will not generate a branching construct for this assignment; therefore, OBC can be achieved with just one test case. If the expression **B** is Boolean, two test cases are needed to achieve decision coverage or MCDC. Similar shortcomings can be shown to exist with OBC when applied to Boolean expressions containing nested short-circuit operators, leading to the conclusion that OBC cannot be considered equivalent to MCDC in the general case.

The cases where MCDC and OBC are equivalent are limited in form. If each Boolean expression involves only one kind of short-circuit operator (either **and then** or **or else**) and the compiler always generates object code to match the flow graph representation of the Boolean expression, then it can be shown that OBC corresponds to MCDC. Thus, for OBC to be considered equivalent to MCDC, at least the following conditions must be met:

a. The compiler must be shown to produce object code reflecting the flow graph model of short-circuit operations under all applicable optimization levels.

b. Each nonsimple Boolean expression is composed of only one kind of short-circuit operator, i.e., multiple operator kinds are not mixed in a single expression.

c. Decision coverage of simple Boolean expressions (i.e., consisting of a single condition) is guaranteed by other techniques (e.g., range testing).

Because these conditions are not always easy to assert, this study does not recommend using OBC as an alternative to MCDC.

4.5 GENERIC UNITS AND INSTANTIATIONS.

Languages, such as Ada, support generic units, which are software units that the compiler tailors to specific parameters, such as data types. A binary search procedure that operates on arbitrary tables of data is an example of a generic subprogram. The process of making a specific instance of a generic unit is called instantiation. Roughly speaking, generic units can be thought of as models or templates from which compilable code must be derived for each instantiation. To that extent, a generic unit represents a (possibly infinite) family of units, and an instantiation selects a member of that family. A primary objective of generic units is to describe data structures and algorithms in a more general way and let the compiler derive specific instances as needed. In languages like C with no direct support for generics, macros can be used to achieve the same goals. The objective of this section is to define guidelines on how generics and macros should be addressed relative to structural coverage.

Although compilers differ in how they handle instantiations, the Ada semantic model is that of a template in which the generic declaration and body are reproduced with generic formal parameter names substituted with the actual parameter entities. The substitution is not text-for-name substitution, as it is in macro expansions, but rather an entity-for-name substitution. The most straightforward and acceptable implementation is for the compiler to use the entity-for-name substitution for code generation. An alternative implementation is what is known as

shared-code implementation, where multiple instantiations share all or parts of the same common body of code generated directly from the generic body. Segments of code that depend on instantiation parameters are called parametric code segments.

Following the Ada model, an approach to structural coverage that addresses each instantiation individually, regardless of the underlying implementation, should be acceptable from a DO-178B point of view. Determining the appropriate circumstances where testing a smaller number of representative instantiations should be acceptable is not trivial. The following guidelines are proposed:

If not all instantiations are to be analyzed for structural coverage, an analysis should be performed to determine the extent to which generated code is shared.

If the analysis shows that the implementation uses 100% shared code (i.e., every instantiation uses the same body of object code that contains no parametric code segments), it should be acceptable to perform structural coverage analysis using a single instantiation and structural coverage criteria applicable to a nongeneric implementation equivalent to the instantiation. This is not to say that the coverage obtained from multiple instantiations that use the same 100% shared code cannot be combined to produce an overall picture of the coverage obtained; rather, it is a statement that it is sufficient to obtain coverage for a single instantiation of a generic implemented using 100% shared code.

If the analysis shows that there are parametric code segments present, then the generated code should be modeled by pseudocode to represent the variants of parametric code segments, and a sufficient number of instantiations should be tested to attain coverage criteria applicable to the pseudocode model.

If the analysis shows that no code is shared between generic instantiations, coverage must be obtained separately for each instantiation.

The following procedures provide examples.

Consider the generic procedure to exchange the values of two variables of an arbitrary enumeration type:

```
generic
   type Item_Type is (<>);
procedure Exchange (A, B : in out Item_Type);
procedure Exchange (A, B : in out Item_Type) is
   T : constant Item_Type := A;
begin
   A := B;
   B := T;
end Exchange;
```

This generic procedure will lend itself to 100% shared-code implementation if the compiler limits all enumeration types to an eight-bit representation. In this case, it would be sufficient to test and analyze any one instantiation, such as

```
procedure Exch_Bool is new Exchange (Boolean);
```

Now consider a more generic version of Exchange:

```
generic
   type Item_Type is range <>;
procedure Exchange (A, B : in out Item_Type);
```

This procedure will exchange the contents of two variables of an arbitrary integer type. If the compiler implements more than one integer base type of differing lengths (say 8, 16, or 32 bits), then any shared-code implementation will need to distinguish between types of different lengths.

If there are only a few instantiations, testing each instantiation would be reasonable. Otherwise, the generated code could be analyzed for parametric code variants and the shared-code implementation could be modeled in pseudocode, as shown in this example.

```
if Item_type'Size = 8 then
procedure Exchange8 (A8, B8 : in out Item_Type8) is
   T8 : constant Item_Type8 := A8;
begin
   A8 := B8;
   B8 := T8;
end Exchange8;
elsif Itemp_Type'Size = 16 then
procedure Exchange16 (A16, B16 : in out Item_Type16) is
   T16 : constant Item_Type16 := A16;
begin
   A16 := B16;
   B16 := T16;
end Exchange16;
else
procedure Exchange32 (A32, B32 : in out Item_Type32) is
   T32 : constant Item_Type32 := A32;
begin
   A32 := B32;
   B32 := T32;
end Exchange32;
end if;
```

It should be sufficient to test three instantiations, one of each variant of generated code to demonstrate structural coverage. Note that this abbreviated form of testing should be acceptable even if the compiler generates separate code for each instantiation, provided an analysis of the generated code shows that these are the only three variants generated.

The following example addresses a more subtle form of possible parametric code variants.

```
Generic
   type Fixed_1 is delta <>;
   type Fixed_2 is delta <>;
function Compare(F1 : Fixed_1;
                 F2 : Fixed_2) return Boolean;
-- function body
function Compare(F1 : Fixed_1;
                 F2 : Fixed_2) return Boolean is
begin
   return Fixed_2 (F1) = F2;
end;
```

Any shared code generated for this function must be sensitive to the precisions of the fixed point types used to instantiate the function. The type conversion in the **return** statement would require a scaling (perhaps in the form of a left or right shift) that is a function of the precision of Fixed_1 and Fixed_2. If analysis determines that the generated code has the following pseudocode representation, it would be necessary to have at least two tests to achieve minimum structural coverage for the generic unit.

```
function Compare(F1 : Fixed_1;
                 F2 : Fixed_2) return Boolean is
begin
   if Fixed_1'Small < Fixed_2'Small then
   return Right_Shift (F1, Shift_Count) = F2;
   else
   return Left_Shift (F1, Shift_Count) = F2;
   end if;
end;
```

Furthermore, it may be necessary to test the code with shift counts from all equivalence classes (e.g., 0, <32. 32, >32) in each branch. This example also brings out the possibility that shared code implementations could involve parameters, such as shift count, that are not present in the source code.

4.6 TASK TYPES AND PROTECTED TYPES.

The Ada language includes support for concurrent execution of multiple threads of control through the use of the task construct. Ada also provides protected units for controlled access to data shared by multiple tasks. Ada tasks and protected units are objects, and as such, are all instances of some type. It is expected that all instances of the same type will share the same object code. Therefore, when considering the structural coverage of multiple instances of a task type or a protected type, the same logic that was used for generic instances with 100% shared code would also apply.

If there is evidence that the compiler does not implement task types or protected types in this manner, the issue of structural coverage for instances of task types or protected types is the same as that of instances of generic units that do not share 100% of their source code.

4.7 OBJECT-ORIENTED PROGRAMMING.

The FAA's "Handbook for Object Oriented Technology in Aviation" [24] identifies a number of issues relevant to structural coverage analysis. Because object orientation is also a topic of RTCA/SC-205, no specific issues in this area were covered in this study.

4.8 TRACEABILITY TO TEST CASES.

Providing evidence that coverage was achieved for the right reasons is a concern related to traceability to test cases. Evidence might include a direct link between requirements-based test cases and the level of coverage they provide. Controversy arises because there is no specific directive in DO-178B for this evidence.

In general, this issue is beyond the scope of this task, except in the sense that a developer may want a structural coverage analysis tool to provide the desired traceability data. Information about whether a tool generates traceability data may be part of a checklist of questions that could be used in conjunction with the evaluation criteria.

4.9 COVERAGE AT THE OBJECT CODE LEVEL.

Where to measure coverage, at either the source or object code level, is an additional source of debate. Structural coverage achieved at the source code level can differ from that achieved at the object code level. Achieving MCDC at the source code level does not guarantee MCDC at the object code level, and vice versa. CAST position paper 17 discusses concerns regarding the analysis of structural coverage at the object code level rather than at the source code level. Many of the concerns are about the verification of assumptions made about compiler behavior and assurance of coding standards. Some of these are discussed in DO-248B DP #12.

In general, this issue is beyond the scope of this task, because the wide variation in compiler behavior may considerably complicate the development of evaluation criteria for structural coverage tools that measure coverage at the object code level.

4.10 QUALIFICATION REQUIREMENTS.

In general, most of the uncertainty regarding qualification of structural coverage analysis tools comes from ambiguities about structural coverage itself, and not qualification. The major source of uncertainly that is associated with qualification concerns the tool's operational requirements. Clearly defining acceptable operational requirements and depth of verification of those requirements are subject to interpretation. Section 9-6 of FAA Order 8110.49 provides some general guidance about evaluating acceptability of operational requirement data, but the guidance provided is very broad.

5. TEST SUITE FOR STRUCTURAL COVERAGE ANALYSIS.

The focus of this study was on developing criteria to evaluate the acceptability of structural coverage analysis tools in a DO-178B context. Given the discussion in section 4, a test suite was determined to be the most effective approach to increase objectivity and uniformity in the

application of the tool qualification criteria. This approach was selected primarily because execution of a coverage analysis tool was considered to be the most practical way to determine which variation of particular structural coverage metrics is implemented and how those metrics are implemented.

Because development and implementation of a full test suite was beyond the scope of the project, a full set of test requirements was identified for each of the three levels of criticality, and a subset of those requirements was implemented in a prototype test suite. The requirements build from foundational test cases for statement coverage for Level C, with additional test cases added as appropriate for Levels B and A. That is, the test requirements for Level B do not repeat the requirements for statement coverage; and likewise, the test requirements for Level A do not include the requirements for statement and decision coverage already covered in the Level C and B requirements. The complete description of the test suite requirements is extensive, and is deferred to appendix B. The following sections provide a brief outline of the requirements for each criticality level.

5.1 OVERVIEW OF TEST SUITE REQUIREMENTS FOR LEVEL C.

DO-178B requires statement coverage for Level C software. Because statement coverage is the simplest of the coverage metrics, statement coverage is not expected to pose a significant problem for commercial tools. For statement coverage, the expectation is that a coverage analysis tool should be able to correctly monitor a core subset of constructs, including

- sequences of simple statements, i.e., a sequence of statements including no branching constructs or compound statements.
- compound statements, i.e., loops, **if** statements, **case** statements, **switch** statements.
- other branching constructs such as **return** statements that exit subprograms, statements that exit a loop entirely (e.g., Ada's **exit when**), or **goto** statements.

There are, however, a few possible areas of differentiation between tools for statement coverage.

- The range of constructs for which the tool can perform coverage analysis. Some examples of constructs that coverage analysis tools may not fully support are
  - Ada exception mechanisms, including both exception handlers and statements that raise exceptions either explicitly (**raise**) or implicitly (e.g., an attempt to access an element outside the range of an array)
  - declarations that generate executable code, e.g., variables with initial values
  - Ada tasking constructs

- The tool's approach to coverage analysis for constructs where a number of instances are created from the same source code. Two examples of this are inlined subprograms and the Ada generics feature.

In addition, it is expected that a tool should be able to correctly indicate the coverage obtained on the source code in a manner that is consistent with its semantics. A coverage analysis tool should be able to correctly handle situations such as the following:

- Variations in the textual presentation of the constructs in the source code file, including a single statement spread over multiple lines of source code and multiple statements on a single line of source code
- Overloaded names for entities
- Separate compilation of subunits
- Nesting of constructions within other constructs

A coverage analysis tool should also be able to correctly analyze coverage in the presence of compiler optimizations. However, structural coverage tools could legitimately vary in their interpretation of structural coverage in situations where a single segment of source code results in an object code in more than one place. One example is inlined subprograms, another is represented by Ada's generic constructs. In these situations, a tool could choose to monitor coverage in a cumulative manner, with the coverage obtained by all the instances being combined. Alternatively, a tool could monitor the coverage obtained separately for each instance.

With the above discussion in mind, the test suite requirements for Level C were defined as follows:

- Verify the range of constructs handled by the tool. Tests should address constructs according to the following progression:
  - Basic blocks (straight-line code, no branching)
  - Compound statements (loops, conditional statements)
  - Ada exception mechanisms (exception handlers, statements raising exceptions)
  - Various exit points (**exit**, **break**, **return**)
  - Initialization statements that generate executable code (variable declarations that specify an initial value)

  The tests dealing with Ada's exception mechanisms were split out because they are specific to the Ada language. While the constructs providing exit points may vary between languages, there is sufficient overlap in concept to group these together.

- Verify whether partial evaluation of expressions results in coverage of the entire expression. This is a special case of the above requirement to test the range of constructs covered by the tool.

- Verify whether a tool's coverage analysis is consistent with the semantics of the language. These tests should address whether a tool can correctly deal with such things as variations in the textual presentation of constructs in source code files, overloaded names, separate compilation, and nested constructs.

- Verify coverage of Ada's generic constructs. These tests are split out because they deal with the Ada-specific generic constructs. These tests should provide insight into the manner in which a tool deals with the issue of having multiple instances of object code created from one instance of source code.

- Verify coverage analysis in the presence of compiler optimizations. These tests address a tool's ability to correctly handle situations in which a compiler performs optimizations that result in code movement or omission. This includes verifying that the following actions do not impact coverage:

  - killed assignment elimination
  - common subexpression optimization
  - loop invariant optimization
  - dead code elimination optimization
  - subprogram inlining optimization

5.2 OVERVIEW OF TEST SUITE REQUIREMENTS FOR LEVEL B.

A structural coverage analysis tool used on a Level B project must satisfy the test requirements for Level C in addition to the test requirements specific to Level B. Here, the requirements specific to decision coverage are discussed. The tests for decision coverage fall into three groups that reflect the definition of the criteria itself.

- Tests that address the handling of Boolean expressions
- Tests that address the handling of entry and exit points
- Tests that address the Ada-specific features of type derivation and operator overloading

Tests for Boolean expressions can be broken down into tests for Boolean expressions that occur in branching constructs and those that do not. Tests for Boolean expressions that occur in branching constructs can be broken down further into two groups.

a. Group 1. Verify that Boolean expressions in branching constructs are handled correctly.

(1) Verify that simple Boolean expressions are handled correctly, including

(a) variable references.
(b) calls to functions that return Boolean results.
(c) relational and range test expressions.
(d) Boolean attribute functions in Ada.
(e) constant Boolean values.

(2) Verify that complex Boolean expressions are handled correctly, including

(a) Boolean expressions containing the unary **not** operator.

(b) Boolean expressions containing the binary operators **and** and **or.**

(c) Boolean expressions containing relational operators such as greater than, less than, or equal.

(d) Boolean expressions containing higher-order operators, that is, operators with more than two operands.

(e) Boolean expressions containing multiple operators.

Within each category, the requirement should be tested in a number of contexts, such as **if** statements, **elsif** statements, and **case** statements.

Tests for Boolean expressions in nonbranching contexts examine how Boolean expressions are handled when they appear in different contexts.

b. Group 2. Verify that Boolean expressions in nonbranching constructs are handled correctly.

(1) Verify that Boolean expressions are correctly handled in

(a) assignment statements.

(b) **for** loops where the index variable is of type Boolean.

(c) actual parameters to subprogram calls.

(d) default expressions in various contexts in Ada programs.

(e) various other nonbranching contexts, many of which are found in Ada but not in C, including

1) array subscripts,
2) components of aggregates,
3) actual parameters of generic instantiations, and
4) guard expressions in Ada selective accept statements.

(2) Verify that entry and exit points are handled correctly, including

(a) subprogram entry and exit points.

(b) other entry and exit points as described by DO-248B for such syntactic constructs as **case** or **switch** statements.

(3) Verify that Boolean operators are handled correctly with type derivation and operator overloading. These tests are Ada-specific, but should be included for the evaluation of coverage analysis tools that support the Ada language.

5.3 OVERVIEW OF TEST SUITE REQUIREMENTS FOR LEVEL A.

Following the principle established at Level B, Level A tests should concentrate on structural coverage requirements beyond those for Level B, namely, the requirement to show that each condition in a decision independently affects a decision's outcome. This requirement coupled with statement and decision coverage constitute MCDC.

The Level A test suite requirements are structured similar to those for Level B, namely:

a. Verify that Boolean expressions in branching constructs are handled correctly.

MCDC is applicable to all Boolean expressions regardless of context. However, in some languages an otherwise non-Boolean expression may take on a Boolean variant in only certain contexts, such as **if** statements. C is an example of such a language, where a numeric expression takes on the Boolean meaning if it occurs in the condition of an **if** statement, **while** loop, or any other condition that controls program flow. Consequently, coverage analysis tools developed specifically for such languages may not track MCDC quite the same way for expressions occurring in all contexts.

(1) Verify simple Boolean expressions including variables, function calls, relational expressions, and attribute references.

(2) Verify Boolean expressions containing Boolean operators.

(a) Expressions containing uncoupled Boolean terms

1) Expressions containing **and**, **or**, and **not** operators
2) Expressions containing other binary Boolean operators
3) Expressions containing short-circuit operators

(b) Expressions containing coupled Boolean terms

1) Expressions containing **and**, **or**, and **not** operators
2) Expressions containing other binary Boolean operators
3) Expressions containing short-circuit operators

b. Verify that Boolean expressions in nonbranching contexts are handled correctly.

These objectives could be divided into subclasses similar to those under Boolean expressions in branching contexts. However, such a subdivision is not very useful because the real issue between branching and nonbranching contexts is whether or not the tool recognizes Boolean expressions as such. Experience with the construction of coverage analysis tools suggests that, once recognized, the coverage treatment of the Boolean expressions is not likely to be dependent on the differences in the context in which they appear. Therefore, the tests needed in this category could be limited to a sampling of tests from all the subcategories under Boolean expressions in branching contexts.

As noted in Issue 10 in section 4.3.3, several interpretations of MCDC exist. A number of interpretations of MCDC could be considered acceptable for a given project, depending on the details of the project. Distinguishing among those interpretations, especially as they may be implemented in a tool, is a challenge. For the test suite, a number of special test cases were identified that could be executed to distinguish between various interpretations of MCDC. The rationale for the test cases, along with a description of those test cases, is given in appendix B.

6. THE PROTOTYPE TEST SUITE.

As one might imagine, building a complete test suite would be a large undertaking. Not only would a test suite need to account for the nuances of the structural coverage definitions to confirm compliance with DO-178B, but a test suite would also need to address differences in programming language, level of measurement (whether at the source or object code level), and other implementation details specific to different types of tools. Developing a full test suite was beyond the scope of the SVTAS.

For the SVTAS, a prototype test suite was developed to explore the feasibility of a test suite as described in section 5. To help narrow the scope of the prototype test suite, test cases were written at the source code level only and were written for the Ada language. To select a reasonable set of test requirements to implement, the following requirements were considered.

- For Level C, the prototype tests should
  - discriminate between tools that provide a separate coverage analysis for each instantiation of a generic unit and those that provide a single, cumulative coverage analysis of a generic unit using coverage accumulated from all instantiations of that unit.
  - discriminate between tools that provide coverage analysis for declarative statements and those tools that do not.

- For Level B, the prototype tests should

  – discriminate between tools that provide coverage analysis for decisions in nonbranching as well as branching contexts and those that provide coverage analysis only for decisions in branching contexts.

  – discriminate between tools that regard the operands of short-circuit operators as separate, top-level decisions as suggested by DO-248B, and those that regard them as conditions of a single, higher-level decision as suggested in reference 10.

- For Level A, the prototype tests should

  – discriminate between tools that implement various interpretations of MCDC.

Further, the prototype test suite does not assess all available language constructs. In particular, the prototype test suite does not support tasking or object-oriented programming. The rationale for this choice is two-fold: (a) the tasking features of Ada are rarely used in embedded, safety-critical applications due to the difficulties of guaranteeing deterministic cyclic scheduling that is key to such applications, and (b) coverage analysis of object-oriented programs is still the subject of research. Upon completion, the prototype test suite consisted of a total of 165 test cases with 27 tests specific to Level C, 23 tests specific to Level B, and 115 tests specific to Level A.

To determine the efficacy of the test suite approach to improving the qualification process, three structural coverage analysis tools were selected to evaluate the prototype test suite. The primary basis for selecting tools was the cost of the tools. Other factors included the current use of the tool (is the tool commonly used in aviation applications?), tool capabilities (does it measure all three types of structural coverage?), and language considerations (does it support Ada?). Two of the tools are commercially available tools currently used within the aviation software industry. The third tool is a tool developed by Boeing for its own use. Here, the identity of these tools are masked and are referred to only as Tool X, Tool Y, and Tool Z.

6.1 ADAPTING THE PROTOTYPE TEST SUITE.

Structural coverage analysis tools can differ in several ways, including the placement of source code during the test, the means of automating tests, and the tool-specific artifacts to be generated for each test. Three specific steps were needed to adapt the prototype test suite for each tool.

- Arranging the test source code into a form acceptable to the tool. This step included grouping multiple compilation units into a single source code file and locating the source code with respect to test folders. This step proved highly amenable to automation.

- Generating supporting artifacts needed for executing the tests. Two of the structural coverage analysis tools relied on Microsoft® Windows® batch files for executing groups of tests. The third tool included support for a batch operation mode. Batch files were generated for the tools that relied on them, and tool-specific artifacts were generated for the batch operation mode. One of the test tools required the generation of an Ada

procedure that would handle various initialization and finalization tasks associated with the tool's coverage analysis implementation. This procedure was easily generated by the script adapting the test suite for that tool. This step also proved highly amenable to automation.

- Interpretation of the coverage analysis reports. The highly individual nature of the coverage analysis reports produced by the different tools made this step least amenable to automation. In fact, the coverage analysis reports had to be manually reviewed to determine the results of each test.

The process of adapting the prototype test suite for the various tools was automated using Tcl/Tk[6] scripts. These scripts were created specifically for each tool. With the exception of the analysis of the pass/fail status of each test, the prototype test suite proved quite adaptable for use with different coverage analysis tools.

6.1.1 Tool X.

Tool X is a software verification tool with the ability to generate test harnesses and perform coverage analysis. Tool X expects the user to provide both the source code for the software under test and a test description file describing the test harness to be generated for that test. Each test description file must be placed in a separate folder. The source code for the software under test may be placed in multiple folders; the only restriction is that it may not be placed in the same folder as a test description file.

Tool X is able to perform a group of tests in a batch execution mode. In this mode, the user only needs to provide certain general information about how the tests are to be performed (e.g., where the tests are located and what compiler flags, if any, are required) before starting the batch execution process. Once the batch execution process begins, Tool X enters each test folder, performs that test, and moves on to the next folder without further user interaction.

The script that adapts the test suite for use with Tool X must perform the following actions for each test.

- Create a folder in the modified test suite hierarchy.

- Create a test description file for use by Tool X. This test description file is very basic, providing only instructions for Tool X's coverage analysis facility and specifying a driver that will call the library-level subprogram that is part of the test suite.

- Copy the source code to a common source code folder. Tool X's design is such that the source code for the test cannot remain in the same folder as the test description file. The naming convention used for the source code included with the test suite tests is such that

[6] Tcl, Tool Command Language, is an interpreted language with programming features, available across platforms running Unix®, Windows, and the Apple® Macintosh® operating system. Tk is a graphical user interface toolkit that enables quick creation of graphical user interfaces.

there is no problem with duplicate filenames between different tests. This allows all source code from the test suite to be placed in a single folder.

Once the test suite has been modified for use by Tool X, the tool's batch execution mode is used to perform all the tests.

6.1.2 Tool Y.

Like Tool X, Tool Y is able to both construct a test harness and perform coverage analysis on the software being tested. As with Tool X, the concern is not for testing the tool's ability to generate a test harness; the testing is concerned only with the coverage analysis facilities provided by the tool. Tool Y depends on DOS batch files for automating groups of tests. In this mode, Tool Y is called with a series of parameters that describe the actions to be performed by the tool.

It is important to note that an evaluation copy of Tool Y was used for this study. The evaluation copy did not provide full access to the features of Tool Y. The evaluation copy of the tool was limited to performing coverage analysis on a single source code file for any given test, whereas the full version is able to perform coverage analysis on multiple files. Though at first this seemed to be a serious limitation of the tool with respect to this study, a work-around was developed. This work-around was to simply concatenate all source code files for which coverage analysis was desired for a particular test into a single source code file. Because the source instrumentation introduced to the source code file by Tool Y is indifferent to the means used to compile and run the user's software, it was possible to instrument the combined source code file, split the single instrumented source code file into the multiple source code files required by the compiler used in this study (GNAT), compile the test software, execute the test, and perform the required coverage analysis using Tool Y.

The script that adapts the test suite for use with Tool Y performs the following actions for each test.

- Create a folder in the modified test suite hierarchy.
- Copy the source code for the test into the new folder. All source code that must be instrumented will be placed into a single source code file.
- Create a batch file that will
  - use Tool Y to instrument the source code.
  - split the instrumented source code file into separate source files for each compilation unit.
  - compile the test.
  - run the test executable.
  - use Tool Y to perform the coverage analysis.
- Add instructions to the per-level batch file to execute the test's batch file.

The modified test hierarchy allows the test suite tests to be executed in three ways:

- Individually, using the batch file in a single test folder
- By level of criticality (i.e., C, B, or A), using a batch file created to run all tests for that level of criticality
- In full, by using a batch file that calls each of the batch files that correspond to the different levels of criticality

6.1.3 Tool Z.

Tool Z also is able to both construct a test harness and perform coverage analysis on the software being tested. Tool Z depends on DOS batch files for automating groups of tests. In this mode, Tool Z's instrumenter is called with a series of parameters indicating the source code to be instrumented and the type of instrumentation to be performed. In addition, Tool Z requires an Ada procedure that will perform various initialization and finalization activities and serve as the Ada partition main subprogram. After this procedure has been compiled along with the user's instrumented source code, the resulting executable program is run and the coverage analysis reports are produced.

The script that adapts the test suite for use with Tool Z performs the following actions for each test:

- Create a folder in the modified test suite hierarchy.
- Copy the source code for the test into the new folder. All source code that must be instrumented will be placed into a single source code file. While Tool Z does not require this as did Tool Y, because the script for adapting the test suite for Tool Z was based on the script created for Tool Y, it was easier to copy the source code in this manner.
- Create the Ada subprogram required by Tool Z for executing the test.
- Create a batch file that will
  - use Tool Z to instrument the source code.
  - split the instrumented source code file into separate source files for each compilation unit.
  - compile the test.
  - run the test executable.
  - use Tool Z to perform the coverage analysis.
- Add instructions to the per-level batch file to execute the test's batch file.

The modified test hierarchy allows the test suite tests to be executed in three ways:

- Individually, using the batch file in a single test folder.
- By level of criticality (i.e., C, B, or A), using a batch file created to run all tests for that level of criticality.
- In full, by using a batch file that calls each of the batch files that correspond to the different levels of criticality.

6.2 RESULTS OF PROTOTYPE EVALUATION.

The prototype test suite was run with each of the evaluation tools. The tools' performance on each test was assigned to one of the following categories:

- Pass: the tool's coverage analysis was correct on this test.
- False-Negative: the tool incorrectly reported that coverage was not attained when coverage was in fact attained.
- False-Positive: the tool incorrectly reported that coverage was attained when coverage was in fact not attained.
- False-Negative Tool Limitation: the tool incorrectly reported that coverage was not attained when coverage was in fact attained. However, the failure occurred because of a documented tool limitation.
- False-Positive Tool Limitation: the tool incorrectly reported that coverage was attained when coverage was in fact not attained. However, the failure occurred because of a documented tool limitation.
- Other: a catch-all category for test results that were not classified by one of the other categories. This category includes such situations as the following:
  - The tool crashed during the test and produced no coverage report at all.
  - The tool had difficulty during a test, which addressed multiple items, and did not report on all of the items.
  - A test was determined to be incorrect or inadequate at too late a point in the study to allow rerunning of the test.
  - Some tests were constructed to distinguish between two different variations of MCDC. In these cases, there was not necessarily a right or wrong answer, but simply an answer that indicated that a particular variation of MCDC was implemented by the tool. The appropriateness of that variation could be

dependent on other aspects of the project (for example, the use of particular coding standards).

It should also be noted that there are some differences in the total number of tests run among the three tools. This difference occurred because some tests were revised or added late in the project and, due to resource constraints, could not be rerun on all tools.

6.2.1 Results for Tool X.

Table 3 summarizes the test results for Tool X. This tool fared fairly well during the testing, although it did have some problems. This tool had 13 false-positive results, although four of those were in areas where the tool has documented limitations. Seven Level A tests were not run for Tool X due to the resource constraints mentioned previously.

Table 3. Test Results for Tool X

Test Level	Pass	False-Negative	False-Positive	False-Negative Tool Limitation	False-Positive Tool Limitation	Other	Total
C	22		4		1		27
B	14	1	5		3		23
A	84	10				14	108

6.2.2 Results for Tool Y.

Table 4 summarizes the results for Tool Y. This tool had 15 false-positive results, only one of which was due to a documented tool limitation. In addition, this tool was judged to have a false-negative result in 26 tests. It also obtained the most test results in the Other category. Of the 30 test results in that column, 18 were due to tool crashes. Nineteen Level A tests were not run for Tool Y.

Table 4. Test Results for Tool Y

Test Level	Pass	False-Negative	False-Positive	False-Negative Tool Limitation	False-Positive Tool Limitation	Other	Total
C	22	2	3				27
B	3	14	5		1		23
A	50	10	6			30	96

6.2.3 Results for Tool Z.

Table 5 summarizes the results for Tool Z. The results for this tool illustrate the need for testing all three criteria. Tool Z performed well for Levels C and B, but had mixed success for Level A. On Level A, it had nearly as many pass results as Tool X; however, Tool Z also had 19 false-positive results. Nine Level A tests were not run for Tool Z.

Table 5. Test Results for Tool Z

Test Level	Pass	False-Negative	False-Positive	False-Negative Tool Limitation	False-Positive Tool Limitation	Other	Total
C	22					5	27
B	18		4			1	23
A	82	3	19		1	1	106

6.3 OBSERVATIONS AND LESSONS LEARNED FROM THE PROTOTYPE.

Based on the results from the three coverage analysis tools, the following observations can be made about the prototype test suite.

- A test suite can provide information regarding the accuracy of the tool's coverage analysis. More specifically, given that the prototype test suite was not a complete implementation of the test suite, the number of errors indicated was somewhat surprising. However, it should also be noted that the choice of tests to implement was not random, but rather was based on experience and judgment regarding situations where errors might exist.

- A test suite can be constructed in such a way that it is relatively simple to apply its tests to multiple coverage tools. The prototype test suite was adapted for use with three different coverage analysis tools. This process was simple and straightforward.

- The tests in a test suite can be constructed in a way that allows easy subsetting of the tests based on certain information about the test. It is simple to provide a quick categorization of tests according to the coverage criteria to which they apply because the tests are both grouped by criteria and named in a manner such that the first character of the test name indicates the coverage criterion it is designed to test. Because of the test naming convention, it is also possible to easily create test subsets based on various aspects of those coverage criteria. It is somewhat more difficult to provide a simple means for categorizing tests according to other factors such as the constructs used in the test.

- It may be possible to produce a test suite that is applicable to multiple programming languages. This would require careful planning on the part of those constructing the test suite, and would be aided by an approach that concentrates more on the lowest common

denominator rather than features specific to a particular language. Such a test suite could be expected to have a significant body of tests that were applicable across languages, as well as tests that were specific to features found in a particular language.

7. SUMMARY.

The TVAS was initiated to develop specific evaluation criteria that could be used to determine whether the performance of a structural coverage analysis tool on a project is acceptable in the context of DO-178B. The primary objective was to promote consistent and correct assessment of structural coverage analysis tools. During the first phase of the study, commercial structural coverage tools and related policy and guidance were surveyed, issues that might contribute to inconsistencies in the automation of structural coverage analysis were identified, and resolutions for each issue were suggested. The results of these activities led to the development of a prototype test suite for structural coverage analysis tools.

A prototype test suite was developed to explore whether a test suite would be an effective approach to increase objectivity and uniformity in the application of the qualification criteria to coverage tools. Test suite requirements to meet Level A, B, and C structural coverage objectives were defined, and a prototype test suite implementing a subset of those requirements was developed. The prototype test suite was evaluated using three different structural coverage analysis tools to determine the efficacy of the test suite. The prototype test suite identified anomalies in each of the three tools.

The results of this study demonstrate the potential for a test suite to help evaluate whether a structural coverage analysis tool is compatible with DO-178B coverage requirements. The lessons learned from the project, including the issues identified from the literature and tool surveys and from the evaluation of the test suite, provide insight into the difficulties in accurately specifying structural coverage requirements and into issues associated with automated structural coverage analysis.

8. REFERENCES.

1. RTCA/DO-178B, “Software Considerations in Airborne Systems and Equipment Certification,” RTCA, Washington, DC, 1992.

2. Hayhurst, Kelly J., Dorsey, Cheryl A., Knight, John C., Leveson, Nancy G., and McCormick, G. Frank, “Streamlining Software Aspects of Certification: Report on the SSAC Survey,” NASA/TM-1999-209519, August 1999.

3. RTCA/DO-248B, “Final Report for Clarification of DO-178B ‘Software Considerations in Airborne Systems and Equipment Certification’,” RTCA, Washington, DC, 2001.

4. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., and Rierson, L.K., “A Practical Tutorial on Modified Condition Decision Coverage,” NASA/TM-2001-210876, May 2001.

5. Hayhurst, K.J. and Veerhusen, D.S., "A Practical Approach to Modified Condition Decision Coverage," *Proceedings of the 20th Digital Avionics Systems Conference*, Daytona Beach, FL, October 14-18, 2001.

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

7. FAA CAST Position Paper 10, "What is a "Decision" in Application of Modified Condition Decision Coverage (MCDC) and Decision Coverage (DC)?" June 2002.

8. FAA CAST Position Paper 6, "Rationale for Accepting Masking MCDC in Certification Projects," August 2002.

9. FAA CAST Position Paper 17, "Structural Coverage of Object Code," December 2003.

10. Chilenski, John Joseph and Miller, Steven P., "Applicability of Modified Condition Decision Coverage to Software Testing," *Software Engineering Journal*, Vol. 7, No. 5, September 1994, pp. 193-200.

11. The Boeing Company, "Study of Qualification Criteria for Software Verification Tools, Amended Interim Report (Phase 1 Report)," a contract deliverable to NASA under contract NAS1-00106, Task Order 1008, May 19, 2003.

12. The Boeing Company, "Study of Qualification Criteria for Software Verification Tools, Phase 2 Position Paper," a contract deliverable to NASA under contract NAS1-00106, Task Order 1008, September 25, 2003.

13. The Boeing Company, "Final Report, Study of Qualification Criteria for Software Verification Tools, Revision 1," a contract deliverable to NASA under contract NAS1-00106, Task Order 1008, January 18, 2005.

14. Crow, Judith, "Software Verification Tools Assessment Study, Task 7, Phase 1 Final Report," a contract deliverable to NASA under contract NAS1-00079, June 25, 2003.

15. FAA CAST Position Paper 12, "Guidelines for Approving Source Code to Object Code Traceability," December 2002.

16. Modified Condition Decision Coverage Software Testing Criterion, available at http://www.dsl.uow.edu.au/~sergiy/MCDC.html, last visited October 16, 2006.

17. FAA Notice N8110.91, "Guidelines for the Qualification of Software Tools Using RTCA/DO-178B," January 16, 2001 (cancelled January 16, 2002).

18. FAA Order 8110.49, "Software Approval Guidelines," November 2002.

19. Khanna, V., "Software Tool Qualification," *Proceedings of the FAA National Software Conference,* Dallas, Texas, May 2002.

20. Khanna, V., "Software Tool Qualification," *Proceedings of the FAA National Software Conference,* Reno, Nevada, September 2003.

21. Hayhurst, K. and Rierson, L., "Verification Tools," *Proceedings of the FAA National Software Conference*, Dallas, Texas, May 2002.

22. Kornecki, Andrew J. and Zalewski, Janusz, "The Qualification of Software Development Tools From the DO-178B Certification Perspective," *CrossTalk, The Journal of Defense Software Engineering,* April 2006.

23. Dupuy, A. and Leveson, N., "An Empirical Evaluation of the MCDC Coverage Criterion on the HETE-2 Satellite Software," *Proceedings of the 19th Digital Avionics Systems Conference*, October 2000.

24. Federal Aviation Administration, "Handbook for Object Oriented Technology in Aviation (OOTiA)," October 26, 2004.

APPENDIX A—VARIATIONS OF MODIFIED CONDITION DECISION COVERAGE

As described in the main body of this report, a variety of alternate forms of Modified Condition Decision Coverage (MCDC) exist. Here, a number of terms associated with structural coverage analysis are described that are not found in the glossary of DO-178B. The descriptions below are not intended to be exact, but are intended to help explain terms that are not commonly used in conventional software engineering discussions.

Black-Box MCDC (BBM). In this criterion, a decision is treated as a black-box function with *n* inputs and one output. Each input is an independent (uncoupled) variable, with any coupled variables rewritten in terms of one or more of the independent variables. In the general case, the input variables need not be Boolean variables, but the output is always Boolean. The criterion calls for a test set that demonstrates that each of the *n* inputs can independently impact the output.

Branch Coverage. In this criterion, every branch in the program must be taken at least once (that is, a decision occurs at a location that affects control flow). Within the software engineering community, branch coverage is often referred to as decision coverage. Branch coverage, however, is less stringent than the DO-178B description of decision coverage.

Coupled conditions. Conditions that cannot be varied independently of each other are said to be coupled.

- Strongly coupled: Two conditions are said to be strongly coupled if changing one condition will always change the other.

- Weakly coupled*:* Two conditions are said to be weakly coupled if changing one condition may change the other.

- If **X** and **Y** are strongly coupled, then either **X = Y**, or **X = not Y**. Weak coupling can be more subtle; for example, **Y < 1** and **Y < 10** are weakly coupled.

- Uncoupled*:* Conditions that can be varied independently of each other.

Coupled-Cause MCDC (CCM). This criterion requires that each condition in a decision be shown to affect the outcome of the decision while all other uncoupled conditions are held fixed. This variant essentially interprets all other possible conditions (in the DO-178B glossary description of MCDC) to mean all other uncoupled conditions, because coupled conditions cannot be held (or guaranteed to be held) fixed while varying a condition to which they are coupled.

Masking MCDC (MSM). This variant of MCDC corresponds to an interpretation of the criterion stated in DO-178B under which, instead of holding fixed all other possible conditions, one is required only to set all other conditions to such values that permit the effect of changing the one condition whose independence one is trying to establish to be observed at the output.

Object-code Branch Coverage (OBC). This criterion replaces the source level MCDC criterion with a Branch Coverage criterion at the object code level, subject to the restriction that all Boolean expressions be written using the short-circuit forms of the **and** and **or** operators.

Operator Coverage Criterion (OCC). Each operand of each operator of the Boolean expression is demonstrated to have an impact on the outcome of that operator, independent of all other operands of that operator. The independent impact of operand i of operator O is asserted by a pair of operand vectors $X = (x_1, x_2,\ldots x_n/O_x)$, $Y = (y_1, y_2,\ldots y_n/O_y)$, where $O_x /= O_y$, $x_i /= y_i$, and x_i and y_i are both observable (as defined in MSM) at the output of operator O.

Reinforced Condition/Decision Coverage (RC/DC). This criterion extends MCDC to address situations where a decision should retain its value even when the value of a condition within that decision changes. A condition in a decision is said to keep the decision's value if there exists some test vector pair such that the value of the condition changes between the pairs, the value of the decision remains constant, and the values of the other conditions remain constant, if possible.

Unique-Cause MCDC (UCM). This refers to a literal interpretation of MCDC as stated in DO-178B, were each condition must be varied while holding fixed all other conditions, coupled or not, to demonstrate that the varying condition can impact the decision's outcome.

Unique-Cause + Masking MCDC (USM). This criterion is a variant of MSM, with the additional constraint that, while varying the value of a condition to demonstrate its independent impact on the output, the values of all other noncoupled conditions be held fixed. In other words, this criterion places additional restrictions on the test vectors that could be paired to demonstrate the MSM criteria for each condition.

APPENDIX B—TEST SUITE REQUIREMENTS

This appendix contains a description of requirements envisioned for a full test suite (with respect to Ada) to evaluate whether a tool provides structural coverage analysis consistent with the structural coverage metrics in DO-178B. A reasonable objective of a test suite is to determine if a given coverage tool will correctly identify coverage or deficiency thereof in any valid program being tested. The phrase "full test suite" here refers to the fact that each structural coverage criterion in DO-178B was applied to all applicable features of Ada and indirectly to the equivalent features of C. Those aspects of Ada and C that are commonly used in aviation application were included in the test suite description, but some constructs were omitted due to their complexity and rarity of use. Those omissions should have little, if any, impact on the ability of the described test suite to meet its objective. However, it is worthwhile to note that no review, outside of the Software Verification Tools Assembly Study (SVTAS), was conducted to verify the completeness of the test suite description. Such a review may be worthwhile if the test suite concept is developed further.

The test suite requirements are given here for each criticality level, starting with the requirements for Level C.

B.1 REQUIREMENTS FOR LEVEL C TESTS.

Below is a description of tests to evaluate whether a tool correctly assesses statement coverage for Level C. For each test requirement, a test requirement number is given in parentheses followed by a description of that test requirement.

(C.1) Verify that statement coverage of basic blocks is indicated correctly.

The purpose of these tests is to verify that if execution begins at the top of a basic block (a sequence of statements with one entry point and one exit point), all statements in the block are identified as covered. Conversely, if a basic block is not entered, then none of the statements are marked as covered. Note that this definition excludes compound statements such as **if**, **case,** and **loop** statements from inclusion in such a block. However, this definition of basic block does allow the inclusion of block statements.

(C.1.1) Verify that basic blocks are correctly detected.

At this point, it should be noted that detection of basic blocks inside compound statements will be addressed when the detection of those compound statements is addressed. While it would be possible to address the inclusion of basic blocks inside various compound statements as part of this objective, that treatment would have significant overlap with the following treatment.

(C.1.1.1) Verify that basic blocks in subprograms are correctly detected.

Example:

```
procedure Foo is
begin
  A := A + 1;
end Foo;
```

(C.1.1.2) Verify that basic blocks forming a handled-sequence-of-statements in a package body are correctly detected.

Example:

```
package body Foo is
...
begin
  A := B + 10;
  Some_Proc;
end Foo;
```

(C.1.2) Verify that compound statements are correctly handled. Note that correct handling of a compound statement implies correct handling of any statements contained in that compound statement.

(C.1.2.1) Verify that loops are correctly handled.

Example:

```
while X > 10 loop
  X := X - 1;
  Some_Proc;
end loop;

for I in 1 .. 10 loop
  Some_Proc;
end loop;

loop
  Some_Proc;
  exit when X > 10;
end loop;
for I in Support.Identity (1) .. Support.Identity (0) loop
  Some_Proc;  -- not reached
end loop;

while Support.Identity (1) < Support.Identity (0) loop
  Some_Proc;  -- not reached
end loop;
```

(C.1.2.2) Verify that conditional statements are correctly handled.

(C.1.2.2.1) Verify that **if** statements are correctly handled, including variants such as **if-else** and **if-elsif-else**.

Example:

```
if X > 10 then
  Proc1;
elsif Y < 5 then
  Proc2;
else
  Proc3;
end if;
```

(C.1.2.2.2) Verify that **case** statements are correctly handled.

Example:

```
case Light_Color is
  when Red => Proc1;
  when Yellow => Proc2;
  when Green => Proc3;
end case;
```

(C.1.3) Verify that coverage is correctly reported in the presence of exception mechanisms in Ada.

(C.1.3.1) Verify that coverage is correctly reported in the presence of exception handlers in Ada.

Individual exception handlers represent disjoint sequences of statements with implicit transfer of control into and out of them. The objective of these tests would be to verify that statement coverage is correctly reported both when at least one handler is entered or none are entered.

Example:

```
procedure Foo (C : out Integer) is
  Excp1, Excp2 : exception;
  B : Boolean := Support.Identity (True);
begin
  if B then
    raise Excp1;
  else
    raise Excp2;
  end if;
  C := 1;
exception
when Excp1 =>
```

```
  C := 0;
when Excp2 =>
  C := 2;
when others =>
  C := 3;
end Foo;
```

If Foo is called, it should return 0 in C, and only the statements shown in bold face should be reported as covered.

(C.1.3.2) Verify that (apparent) basic blocks that do not complete execution because of exceptions or "long jumps" are handled correctly.

In Ada and in C, it is possible to abandon the execution of an apparent basic block by raising exceptions or by long jumps. The coverage reported in these cases should reflect only those statements that were actually reached.

Example:

```
procedure Foo (B : out Boolean) is
  MyExcp : exception;
  procedure Nested_Foo (Excp : Boolean) is
  begin
    if Excp then
      raise MyExcp;
    end if;
  end Nested_Foo;
  A : Integer := Support.Identity (5);
begin
  A := A + 1;
  Nested_Foo (A = Support.Identity (6));
  B := True;
exception
when MyExcp =>
  B := False;
end Foo;
```

In this case, if Foo is called, the statement B := True; should not be reached, and therefore, should not be identified as covered in the statement coverage report. The statements in bold face represent expected coverage.

(C.1.4) Verify that (apparent) basic blocks that do not complete execution to the end are handled correctly.

(C.1.4.1) Verify that (apparent) basic blocks that do not complete execution because of exit or break statements are handled correctly.

Ada and C both allow the programmer to indicate that a loop execution should be completed and any subsequent statements in the body of the loop should be skipped. Sometimes this skipping of the remaining statements can be for a single

iteration of the loop. In the example below, while the call to Some_Other_Func will not be executed in the event that the value of X is 9, this call is guaranteed to execute at least once unless Some_Func raises an exception.

Example:

```
while X > 0 {
  Some_Func;
  if X = 9 continue;
  Some_Other_Func;
}
```

At other times, the programmer may cause a loop to immediately terminate, skipping any subsequent statements in the body of the loop. Statements that are used to cause this termination are the Ada **exit when** statement and the C **break** statement. In the next example, the call to Some_Other_Proc will not be executed if the value of X is greater than 10 upon the return from Some_Proc.

Example:

```
loop
  Some_Proc;
  exit when X > 10;
  Some_Other_Proc;
end loop;
```

Additional test cases with exits from the top of the loop body and the bottom of the loop body will help verify this objective. A more general case, consisting of multiple exits, could be represented by a test case such as the one below:

```
procedure Foo (I : in out Integer) is
  J : Integer := 0;
begin
  loop
    exit when I = Support.Identity (0);
    J := 1;
    exit when I = Support.Identity (1);
    J := 2;
    exit when I = Support.Identity (2);
    J := 3;
    exit when I = Support.Identity (3);
    J := 4;
    exit when I /= Support.Identity (3);  -- sure exit
  end loop;
  I := -J;
end Foo;
```

Calling procedure Foo with various value of I in the range 0 .. 4 will help verify the objective in the presence of multiple exits within a loop. For example, calling

Foo with I = 2 should return a value of -2 in I and show the following statement coverage:

```
procedure Foo (I : in out Integer) is
  J : Integer := 0;
begin
  loop
    exit when I = Support.Identity (0);
    J := 1;
    exit when I = Support.Identity (1);
    J := 2;
    exit when I = Support.Identity (2);
    J := 3;
    exit when I = Support.Identity (3);
    J := 4;
    exit when I /= Support.Identity (3);  -- sure exit
  end loop;
  I := -J;
end Foo;
```

(C.1.4.2) Verify that (apparent) basic blocks that do not complete execution because of **goto** statements are handled correctly.

(C.1.4.3) Verify that (apparent) basic blocks that do not complete execution because of return statements are handled correctly.

(C.1.5) Verify that declarations that generate executable code are included in statement coverage report.

The objective is to verify that declarations are treated like statements, especially if they generate executable code. In Ada and C, it is possible to initialize objects with expressions that are computed at runtime. Such declarations essentially subsume an assignment statement within them. As such, the declaration should be included in statement coverage report.

Example:

```
procedure Foo (J : out Integer) is
  I : Integer := Support.Identity (0);
begin
  J := I;
end Foo;
```

A call to Foo should identify both the object declaration and the assignment statement as covered.

(C.2) Verify that partial evaluation does not impact statement coverage.

In many languages (Ada and C among them), portions of a statement may not be evaluated due to the semantics of the construct. A statement coverage report should not be impacted by whether or not portions of the statement are evaluated.

(C.2.1) Verify that partial evaluation of short-circuited expressions does not impact statement coverage.

Short-circuit operators, such as **and then** (**&&** in C) and **or else** (|| in C), lead to partial evaluation of statements containing them. The objective of this test category is to verify that statement coverage reported is not impacted by partial evaluation.

Example:

```
A := B or else C;
```

Executing this statement should result in full coverage of the assignment statement, regardless of whether the value of B is true or false.

(C.2.2) [C Only] Verify that partial evaluation of conditional expressions does not impact statement coverage.

In the C language, conditional expressions represent another case where portions of a statement may not be fully evaluated. In fact, a single evaluation of a conditional expression always leaves one subexpression unevaluated. The objective of this test category is to verify that statement coverage is correctly reported for the entire statement even if a portion of a conditional expression contained therein is not evaluated.

Example:

```
A = (A ? B : 0);  /* equivalent to A =&& B;
```

(C.3) Verify that coverage reported is consistent with the semantics of the code.

Some coverage analysis tools may not use full semantics of the language when instrumenting the source code. The tests in this category are aimed at ensuring that the coverage reported is correct with respect to the semantics of the language and any shortcuts taken by the tool do not affect the coverage report.

(C.3.1) Verify that various textual forms of basic blocks are handled correctly.

(C.3.1.1) Verify that a basic block spanning multiple lines is handled correctly.

The objective is to verify that there are no dependencies on line breaks in the way coverage is reported.

Example:

```
A := A + 1;
Call_A_Procedure (A);
B := A > 1 or
     else
     A < 1;
```

If the first assignment statement is reached, then all lines should be reached. Likewise, if the first assignment is not reached, none of the lines should be reached.

(C.3.1.2) Verify that a basic block spanning a single line is handled correctly.

This is the flip side of requirement (C.3.1.1) to verify that there are no dependencies on the presence of line breaks.

Example:

```
A := A + 1; Call_A_Procedure (A); B := A > 1 or else A < 1;
```

If the first assignment statement is reached, then all lines should be reached. Likewise, if the first assignment is not reached, none of the lines should be reached.

(C.3.1.3) Verify that multiple basic blocks on one line are handled correctly.

The objective is to verify that covered and uncovered regions can start and end of the same line.

Example:

```
if A then Foo; else No_Foo; end if;
Depending on the value of A, we should have either:
if A then Foo; else No_Foo; end if;
or
if A then Foo; else No_Foo; end if;
```

(C.3.1.4) Verify that the coverage reported is not affected by source code indentation.

Examples:

```
  while Support.Identity (I) > I loop
I := I - 1;       -- not executed
  end loop;

        if Support.Identity (I) > I
      then
    I := I - 1;   -- not executed
  end
if;
```

(C.3.1.5) Verify that keywords are not used out of context in reporting coverage.

In Ada, a number of keywords (reserved words) are used in multiple contexts. For example, the keyword **if** is used both at the beginning of an **if** statement and its end. Tools that look for keywords at the beginning of a line to identify statements could be tripped up by the presence of an **if** at the beginning of a line.

Examples:

```
if A > B then
  null;
end
if;               -- not the beginning of an IF statement

Foo:
loop
  exit
when Quit; end    -- not the begging of a case clause
loop              -- not the beginning of an LOOP statement
  Foo;            -- not a procedure call

A :=
Foo (2);          -- not a procedure call
B := B or
else              -- not the ELSE of an IF statement
 Stop;

function Foo
return Some_Type; -- not a return statement
```

(C.3.2) Verify that overloading of names does not impact coverage reported.

In the presence of overloading, identical names could have different meanings. For example, if two procedures are named Foo and only one is called, the report should correctly identify the one that was called.

Example:

```
procedure Foo (I : out Integer) is
begin
  I := Support.Identity (1);
end Foo;
procedure Foo (B : in out Boolean) is
begin
  B := not B;
end Foo;
procedure Foo_2 is
  B : Boolean := Support.Identity (0) = 0;
  I : Integer := Support.Identity (99);
begin
  if B then
```

```
    Foo (B);
  else
    Foo (I);
  end if;
end Foo;
```

If Foo_2 is called, the coverage attained should indicate that the second procedure Foo was called and not the first Foo:

```
procedure Foo (I : out Integer) is
begin
  I := Support.Identity (1);
end Foo;
procedure Foo (B : in out Boolean) is
begin
  B := not B;
end Foo;
procedure Foo_2 is
  B : Boolean := Support.Identity (0) = 0;
  I : Integer := Support.Identity (99);
begin
  if B then
    Foo (B);
  else
    Foo (I);
  end if;
end Foo_2;
```

(C.3.3) Verify that separately compiled bodies do not impact coverage.

In Ada, bodies of subprograms and packages that are nested within other units may be compiled separately from the containing unit. The statement coverage report should not be impacted by whether such bodies are written in-line or in a separate compilation unit.

Example:

```
procedure Foo is
  I : Integer := Support.Identity (0);

  procedure Nested_Foo_1 is separate;

  procedure Nested_Foo_2 is separate;
begin
  if I = 1 then
    Nested_Foo_1;
  else
    Nested_Foo_2;
  end if;
end Foo;
```

```
separate (Foo)
procedure Nested_Foo_1 is
begin
  I := Support.Identity (1);
end Nested_Foo_1;

separate (Foo)
procedure Nested_Foo_2 is
begin
  I := Support.Identity (2);
end Nested_Foo_2;
```

If Foo is called, the coverage attained should indicate that Nested_Foo_2 was called and Nested_Foo_1 was not:

```
procedure Foo is
  I : Integer := Support.Identity (0);

  procedure Nested_Foo_1 is separate;

  procedure Nested_Foo_2 is separate;
begin
  if I = 1 then
    Nested_Foo_1;
  else
    Nested_Foo_2;
  end if;
end Foo;

separate (Foo)
procedure Nested_Foo_1 is
begin
  I := Support.Identity (1);
end Nested_Foo_1;

separate (Foo)
procedure Nested_Foo_2 is
begin
  I := Support.Identity (2);
end Nested_Foo_2;
```

(C.3.4) Verify that nesting of program units within each other is handled correctly.

In Ada, subprograms, packages, and declare blocks may all be nested within each other in numerous ways. The objective is to verify that the tool correctly handles such forms of nesting.

Example:

```
procedure Foo (I : Integer) is
  function Foo (I : Integer) return Integer is
  begin
```

```
    return Support.Identity (I);
  end Foo;
  J : Integer := 1;
begin
  Foo1:
  declare
    package Foo is
      function Foo return Integer;
    end Foo;
    package body Foo is
      function Foo return Integer is
      begin
        return Support.Identity (2);
      end Foo;
    end Foo;
  begin
    if I = 1 then
      J := Foo1.Foo.Foo;
    else
      J := Standard.Foo.Foo (3);
    end if;
  end Foo1;
end Foo;
```

If procedure Foo is called with I = 1, the coverage attained should indicate that function Foo nested within package Foo is called; otherwise the function Foo declared within procedure Foo is called.

Several more test cases should be used to assure that other combinations of nesting are also handled properly.

(C.4) [Ada Only] Verify that generics are handled correctly.

Existing guidance is not sufficient to determine how coverage should be tracked for generic units and their instantiations. Test requirements outlined here represent one approach to handling generics.

The method used to indicate coverage resulting from executions of instantiated units may vary from tool to tool. In this document, we use the generic body of the unit to show coverage, as if the generic body itself was executed. Cumulative coverage from executions of multiple instantiations may be indicated in a single copy of the generic body if the implementation uses the shared code approach, or on separate copies for each instantiation if the implementation uses no shared code between instantiations. In either case, the text from generic unit body is used to show coverage by highlighting statements, as we have done in nongeneric cases.

(C.4.1) Verify that statement coverage is reported correctly for a single instantiation.

In this subcategory, the test requirements focus on a single instantiation. Tests should include single execution resulting in partial coverage, single execution resulting in full coverage, multiple executions resulting in partial coverage and multiple executions resulting in full coverage. These cases are illustrated in the following example of an Ada generic package.

Example:

```
generic
  type Item_Type is private;
package Data_Mover is
  type Action_Type is
    (No_Action, Copy_A, Copy_B, Swap);
  procedure Move (A, B : in out Item_Type;
                  Action : Action_Type);
end Data_Mover;

package body Data_Mover is
  procedure Move (A, B : in out Item_Type;
                  Action : Action_Type) is
  begin
    case Action is
    when Copy_A =>
      B := A;
    when Copy_B =>
      A := B;
    when Swap =>
      declare
        T : Item_Type := A;
      begin
        Move (A, B, Copy_B);
        Move (T, A, Copy_A);
      end;
    when No_Action =>
      null;
    end case;
  end Move;
end Data_Mover;

-- Instantiation of Data_Mover for Integer
package Int_Mover is new Data_Mover (Integer);

-- Testing instantiation for statement coverage
with Int_Mover;
procedure Foo_1 is
  package Progress renames Progress_Report;
  I : Integer := 11;
  J : Integer := 22;
begin
  Int_Mover.Move (I, J, Int_Mover.Copy_A);
End Foo_1;
```

```
-- Testing instantiation for statement coverage
with Int_Mover;
procedure Foo_2 is
  package Progress renames Progress_Report;
  I : Integer := 11;
  J : Integer := 22;
begin
  Int_Mover.Move (I, J, Int_Mover.Swap);
end Foo_2;

-- Testing instantiation for statement coverage
with Int_Mover;
procedure Foo_3 is
  I : Integer := 11;
  J : Integer := 22;
begin
  Int_Mover.Move (I, J, Int_Mover.Swap);
  Int_Mover.Move (I, J, Int_Mover.No_Action);
end Foo_3;
```

Calling Foo_1 should yield partial coverage of statements of procedure Move in the instantiation Int_Mover, indicated by:

```
package body Data_Mover is
  procedure Move (A, B : in out Item_Type;
                  Action : Action_Type) is
  begin
    case Action is
    when Copy_A =>
      B := A;
    when Copy_B =>
      A := B;
    when Swap =>
      declare
        T : Item_Type := A;
      begin
        Move (A, B, Copy_B);
        Move (T, A, Copy_A);
      end;
    when No_Action =>
      null;
    end case;
  end Move;
end Data_Mover;
```

Calling Foo_2 also yields partial coverage, indicated by:

```
package body Data_Mover is
  procedure Move (A, B : in out Item_Type;
                  Action : Action_Type) is
  begin
    case Action is
    when Copy_A =>
```

```
      B := A;
    when Copy_B =>
      A := B;
    when Swap =>
      declare
        T : Item_Type := A;
      begin
        Move (A, B, Copy_B);
        Move (T, A, Copy_A);
      end;
    when No_Action =>
      null;
    end case;
  end Move;
end Data_Mover;
```

Finally, calling Foo_3 yields full coverage of procedure Move:

```
package body Data_Mover is
  procedure Move (A, B : in out Item_Type;
                  Action : Action_Type) is
  begin
    case Action is
    when Copy_A =>
      B := A;
    when Copy_B =>
      A := B;
    when Swap =>
      declare
        T : Item_Type := A;
      begin
        Move (A, B, Copy_B);
        Move (T, A, Copy_A);
      end;
    when No_Action =>
      null;
    end case;
  end Move;
end Data_Mover;
```

(C.4.2) Verify that statement coverage is reported correctly for multiple instantiations.

If more than one instantiation exists for a generic unit, coverage from calls to distinct instantiation could be accumulated into a single coverage report, or into distinct coverage reports, one for each instantiation. The following example will distinguish between the two approaches by achieving partial coverage on each instantiation, the aggregation of which will yield full coverage for the generic unit.

Example: The same generic Data_Mover package from the preceding subsection is used, with an additional instantiation:

```
package Bool_Mover is new Data_Mover (Boolean);
--
with Int_Mover;
with Bool_Mover;
procedure Foo_1 is
  package Progress renames Progress_Report;
  I : Integer := 11;
  J : Integer := 22;
  A : Boolean := I = J;
  B : Boolean := not A;
begin
  Int_Mover.Move (I, J, Int_Mover.Copy_B);
  Bool_Mover.Move (A, B, Bool_Mover.Swap);
  Int_Mover.Move (I, J, Int_Mover.No_Action);
end Foo_1;
```

If coverage from multiple instantiation is cumulative, the procedure Move should achieve full statement coverage from the calls to the two instantiations. If coverage is tracked on a per instantiation basis, the following coverage reports would be expected:

```
package body Int_Mover is
  procedure Move (A, B : in out Item_Type;
                  Action : Action_Type) is
  begin
    case Action is
    when Copy_A =>
      B := A;
    when Copy_B =>
      A := B;
    when Swap =>
      declare
        T : Item_Type := A;
      begin
        Move (A, B, Copy_B);
        Move (T, A, Copy_A);
      end;
    when No_Action =>
      null;
    end case;
  end Move;
end Int_Mover;

package body Bool_Mover is
  procedure Move (A, B : in out Item_Type;
                  Action : Action_Type) is
  begin
    case Action is
    when Copy_A =>
      B := A;
    when Copy_B =>
```

```
      A := B;
    when Swap =>
      declare
        T : Item_Type := A;
      begin
        Move (A, B, Copy_B);
        Move (T, A, Copy_A);
      end;
    when No_Action =>
      null;
    end case;
  end Move;
end Bool_Mover;
```

Note: To distinguish the coverage reports, the generic unit name has been replaced with the instantiation name in italics.

(C.5) Verify that compiler optimizations do not impact coverage reported.

The objective is to verify that no matter how much or how little the compiler optimizes the object code, the coverage reported at the source level remains the same.

(C.5.1) Verify killed assignment elimination does not impact coverage.

If an assignment to a variable is killed by another assignment without an intervening reference to that variable, compilers may eliminate the killed assignment in the object code, but the coverage should nevertheless show that the killed assignment is (or is not) covered.

Example:

```
A := 10;
A := 12;
```

Both assignments should be marked as reached if, and only if, the first assignment is reached.

(C.5.2) Verify that common subexpression optimization does not impact coverage.

Another form of optimization is avoiding repeated evaluations of an expression when the same expression appears multiple places without the possibility of any changes to the values of its terms. This is known as common subsexpression optimization and it should not impact the statement coverage report.

Example:

```
A := I + 1;
for J in Integer range 1 .. Support.Identity (0) loop
  A := I + 1;
end loop;
```

There are two occurrences of the expression I + 1. If execution reaches the first assignment statement, the first occurrence will need to be evaluated. However, the second occurrence is never evaluated due to the fact that the loop body is never entered. Thus, the second assignment should not be marked as covered.

Note: The function Identity, defined in Support package, returns the same value as its input argument. Use of this function will keep most compilers from evaluating static expressions as such.

(C.5.3) Verify that loop invariant optimization does not impact coverage.

When a statement in the body of a loop is recognized by the compiler as having no dependency on any terms that could change during the course of execution of the loop, the compiler may arrange to have that statement executed outside the loop. Such code movements should not impact the statement coverage report.

Example:

```
procedure Foo (J : in out Integer) is
  K : Integer := Support.Identity (0);
begin
  for I in Integer range 1 .. J loop
    K := 1;
  end loop;
  J := K;
end Foo;
```

If Foo is called with J > 0, it should return a value of 1, and the statement coverage report should reflect the following coverage:

```
procedure Foo (J : in out Integer) is
  K : Integer := Support.Identity (0);
begin
  for I in Integer range 1 .. J loop
    K := 1;
  end loop;
  J := K;
end Foo;
```

If Foo is called with J <= 1, it should return a value of 0, with the statement coverage reflecting this situation:

```
procedure Foo (J : in out Integer) is
  K : Integer := Support.Identity (0);
begin
  for I in Integer range 1 .. J loop
    K := 1;
  end loop;
  J := K;
end Foo;
```

(C.5.4) Verify that dead code elimination optimization does not impact coverage.

In most languages, it is possible to write programs which contain unreachable statements, and in some cases, the compiler will be able to eliminate them in the object code. However, in a statement coverage report, such statements should be identified as uncovered.

Example:

```
if True then
  A := 0;
else
  A := 1;
end if;
```

An execution of this **if** statement should result in coverage equivalent to:

```
if True then
  A := 0;
else
  A := 1;
end if;
```

(C.5.5) Verify that subprogram inlining optimization does not impact coverage.

The objective is to verify that if a subprogram is inlined, either because the programmer requested it or because the compiler chose it, the coverage resulting from an execution of the inlined subprogram reflects the same coverage as would be expected if it were not inlined.

Example:

```
procedure Foo (K : out Integer) is
  function Max (I, J : Integer) return Integer is
  begin
    if (I > J) then
      return I;
    else
      return J;
    end if;
  end Max;
  pragma Inline (Max);
begin
  K := Max (Support.Identity (0),
            Support.Identity (-9));
end Foo;
```

When called, Foo should return a value of 0 and the corresponding statement coverage should reflect:

```
procedure Foo (K : out Integer) is
  function Max (I, J : Integer) return Integer is
  begin
    if (I > J) then
      return I;
    else
      return J;
    end if;
  end Max;
  pragma Inline (Max);
begin
  K := Max (Support.Identity (0),
            Support.Identity (-9));
end Foo;
```

(C.5.6) Verify that the statement coverage report is correct in the presence of exceptions.

In most real-time applications, exceptions are suppressed or ignored; in others, exceptions are allowed to be raised and handled by the software. In either case, the coverage analysis tool should be able to correctly report coverage when an exception is imminent.

(C.5.7) Verify that coverage reported is consistent with the handling of exceptions.

If the software is built to suppress an exception or ignore it, the statement coverage report should reflect that execution profile. Likewise, if the exception is raised and trapped, any premature termination of operations due to an exception should be reflected in partial coverage reported.

Example:

```
procedure Foo (I : out Integer) is
  J : Integer;
begin
  I := Support.Identity (1);
  J := I / Support.Identity (0); -- raises divide-by-zero
                                 -- exception
  if I = Support.Identity (I) then
    I := Support.Identity (2);
  else
    I := J;
  end if;
end Foo;
```

If Foo is called, the coverage attained should be a function of whether the divide-by-zero exception is allowed to occur, if allowed, whether or not it is ignored. If suppressed or ignored, the procedure will execute to completion and produce a result of 2 in I. If the exception is allowed to occur and control flow is terminated at the point of exception (and handled elsewhere in the calling environment), the value returned in I should be 1. In either case, the statement

coverage report should be consistent with the value returned in I. If I = 1, then coverage should reflect:

```
procedure Foo (I : out Integer) is
  J : Integer;
begin
  I := Support.Identity (1);
  J := I / Support.Identity (0); -- raises divide-by-zero
                                 -- exception
  if I = Support.Identity (I) then
    I := Support.Identity (2);
  else
    I := J;
  end if;
end Foo;
```

The assignment to J where the exception is raised may or may not be considered covered by the coverage analysis tool. Either choice is probably an acceptable interpretation of DO-178B, which is why the statement is indicated in italics. If I = 2, then the coverage should be equivalent to:

```
procedure Foo (I : out Integer) is
  J : Integer;
begin
  I := Support.Identity (1);
  J := I / Support.Identity (0); -- raises divide-by-zero
                                 -- exception
  if I = Support.Identity (I) then
    I := Support.Identity (2);
  else
    I := J;
  end if;
end Foo;
```

B.2 DISCRIMINATING TESTS FOR LEVEL C.

Statement coverage is relatively well understood in the software community, so much so that variations in the interpretation of statement coverage required by DO-178B for Level C software should be rare and confined to relatively obscure cases. Nevertheless, a few situations, such as the following, are worth noting and are included in the test suite.

a. Coverage of declarative statements, especially if they generate code.

b. Coverage of generic instantiations. Are they reported on a per instantiation basis or on a cumulative basis over all instantiations?

c. Coverage of inlined subprograms. When the compiler honors the pragma Inline, is coverage handled separately for each instance of the inlined code or on a cumulative basis as if there were only one copy of the object code?

Test cases for a. were outlined in (C.1.5), case b. is addressed with in (C.4), and case c. is addressed in (C.5.5).

B.3 REQUIREMENTS FOR LEVEL B TESTS.

Level B structural coverage requires that all of the following must be true:

i. All statements have been executed.
ii. All entry and exit points have been covered.
iii. All Boolean expressions have taken on both True and False values.

The first of these items is the statement coverage criterion used for Level C, and thus for a tool to be judged as adequate for Level B structural coverage, that tool must also be judged adequate for Level C structural coverage. With this in mind, the Level C requirements for the test suite will not be repeated here, but should be met to satisfy the Level B test requirements.

With regard to item ii, the Level C test requirements can be seen to satisfy this requirement partially. The Level C test requirements will assure that a tool under evaluation adequately monitors the coverage of explicit entry and exit points; it remains for the level B test requirements to test whether a tool under evaluation adequately monitors implicit entry and exit points. In most high-level languages, including Ada and C, the first statement or the first declaration that generates executable object code is the only (explicit) entry point for a subprogram. An example of multiple explicit entry points could be found in some dialects of FORTRAN, where additional points in a subroutine could be marked as entry points. Explicit exit points are often represented by a **return** statement or a subprogram **end** directive. Examples of implicit entry points can be found in Ada exception handlers. Such exception handlers are commonly entered through an exception raised during the execution of other statements. For example, consider the following code segment:

```
procedure Foo (K : Integer) is
      I : Integer;
begin
      I := K ** 2;
      Bar (I);
exception
when Constraint_Error =>
      Bar (K);
end Foo;
```

The statement Bar (K); is the exception handler for Constraint Error and it is an implicit entry point. It could be entered in many different ways, including from an overflow resulting from the evaluation of the expression K ** 2, from some constraint violation arising in Bar, or from an explicit **raise** statement in Bar. Likewise, if the occurrence of an exception causes a subprogram to be exited, such an exit point would be considered an implicit exit point. Additionally, the discussion in DO-248B indicates that some constructs, such as Ada's **case** statement or C's **switch** statement, can themselves contain entry points. With respect to these two constructs, the entry points would be the various alternatives they present for execution. In the case of Ada, if statement coverage is achieved for each alternative, then all entry points for the construct have

been covered as well[1]. In the case of the C **switch** statement, however, mere structural coverage would be insufficient to indicate coverage of the entry points if any of the alternatives are allowed to flow down into the following alternative (i.e., if any of the alternatives other than the last does not end with a **break** statement.) In the following example, the **switch** statement has two entry points, but statement coverage could be achieved with only one test case (i=42), leaving the second entry point uncovered.

```
switch (i) {
  case 42 : {
    Special_Processing (i);
  }
  default : {
    Normal_Processing (i);
  }
};
```

With regard to item iii, it should be noted that some software verification tools may recognize and handle Boolean expressions that control flow in a different manner than those that occur in other contexts. For this reason, the requirements for Level B tests include separate top-level categories for dealing with Boolean expressions in branching contexts and for dealing with Boolean expressions in nonbranching contexts.

In succeeding lower levels, possible variations in the construction of the decision itself are considered, such as whether the expression is a simple Boolean variable, whether it is a call to a Boolean function, whether it contains Boolean operators, and so on. Ideally, the test suite should address all possible combinations of these attributes. But the combinatorial explosion that will result from the Cartesian product of all attributes will make that approach impractical. Because real-life compilers and other tools tend to handle most of these attributes independently of each other (e.g., if function call is correctly handled in an assignment context, it will be correctly handled in an actual parameter context, and vice versa), it should be sufficient to sprinkle the variants of the less significant attributes across combinations the most significant attributes in order to keep the test suite tractable.

B.3.1 NOTATION FOR COVERAGE INDICATION.

The following sections describe the objectives of the Level B tests in detail, and in many instances the discussion is accompanied by examples of source code annotated to indicate the expected coverage from a given set of test cases. In those examples, the following notation is used to indicate expected coverage:

- A Boolean expression that takes on only the value False is single-boxed.
- A Boolean expression that takes on only the value True is double-boxed.
- A Boolean expression that takes on both True and False values is boxed in bold stripes.

[1] Based on the if-elsif-else model of implementation for the **case** statement. Tests based on a branch table model will incorrectly fail tools that are based on the if-elsif-else implementation model.

- A Boolean expression that does not get evaluated at all is in **boldface text**.

The portion of text that is boxed is determined as follows:

- For a simple Boolean variable or constant, the name is boxed, e.g., Bool.RA.
- For an expression involving an operator that delivers a Boolean result, the operator is boxed, e.g., (A > 10) or (B <= 0).
- For a function call, the function name is boxed, e.g., Is_Positive (I).

B.3.2 Actual Test Requirements for Level B.

Below is a description of tests to evaluate whether a tool correctly assesses decision coverage for Level B.

(B.1) Verify that Boolean expressions in branching constructs are handled correctly.

The purpose of these tests is to verify that Boolean expressions that control flow are handled correctly. In other words, these tests confirm proper handling of the traditional branch coverage criterion. The following contexts should be tested:

The **if** condition of an **if** – **then** statement.
The **elsif** condition of an **if** – **then** – **elsif** statement.
The loop condition of a **while** loop.
The condition for an **exit when** statement.
case with a Boolean selector expression.
Boolean expressions with short-circuit operators.

(B.1.1) Verify that simple Boolean expressions are handled correctly.

The phrase "simple Boolean expressions" means Boolean expressions that contain a single Boolean term. A Boolean term could be a Boolean variable reference, a relational expression, a range test, a Boolean function call, an attribute reference, and so on. Boolean expressions containing unary, binary, or other higher-order operators are not considered in this objective.

(B.1.1.1) Verify that Boolean variable references are handled correctly.

A Boolean variable reference could take the form of a local variable, global variable, a parameter passed in, a component of an array or a record object, or a combination of these. The example under (B.1.1.5) covers many of these cases.

(B.1.1.2) Verify that calls to functions that return Boolean results are handled correctly.

The objective of these tests is to confirm that coverage is reported correctly when a simple Boolean expression consists of a function call. Examples of tests that meet this objective can be found under (B.1.1.5).

(B.1.1.3) Verify that relational and range test expressions are handled correctly.

A relational expression is one that compares the values of two objects of compatible types for an ordering relation, such as less than, less than or equal, or equal. Some languages, including Ada, support a special form of relational expression that tests whether the value of a variable is in a given range, which amounts to testing the variable's value to be greater than or equal to the lower bound of the range and less than or equal to the upper bound without using the Boolean conjunctive operator **and**. The objective of these tests is to verify that decision coverage is correctly reported for such expressions. Tests that meet this objective can be found in the example under (B.1.1.5).

(B.1.1.4) Verify that Boolean attribute references are handled correctly.

In Ada, certain attributes, such as Boolean'Val and *object*'Valid, return Boolean results. The objective of tests in this category is to verify that coverage is reported correctly for Boolean expressions that constitute such attribute references. Tests that meet this objective are included in the example under (B.1.1.5).

(B.1.1.5) Verify that constant Boolean values are handled correctly.

A literal reading of DO-178B would call for all Boolean constants to be marked as not sufficiently covered, as they will never take on all possible Boolean values. Several tools will mark such expressions as uncovered and the applicant often has to write a deviation justifying their use. Use of constant Boolean expressions in branching contexts is less common and could even be indicative of a programming error, but their use in other contexts is far more common, even inevitable. A verification tool that discriminates between these contexts in reporting coverage anomalies may be a desirable. For this reason, the test suite should include test cases representative of both classes of uses of constant Boolean expressions.

Example: The following example illustrates tests for various forms of simple Boolean expressions discussed in (B1.1).

```
package Bool is
   type BoolArr is array (Character) of Boolean;
   type Rec is
      record
         A : Integer;
         B : Boolean;
         C : Character;
         D : BoolArr;
```

```
      end record;
   type RecArr is array (Natural range <>) of Rec;

   B1 : Boolean := False;
   CB : BoolArr := ('A' .. 'Z' => True, others => False);
   Rc : constant Rec :=
     (10, False, 'J', ('a' .. 'z' => False, others => True));
   R1 : Rec := Rc;
   RA : RecArr := (1 .. 100 => Rc);
   I1 : Integer := 0;

   function Is_Positive (I : Integer) return Boolean;
   --  return True if I > 0, false otherwise; in addition,
   --  the global variable I1 is incremented on each call.
end Bool;

with Bool;
with Support;
with Progress_Report;
procedure Foo is
   package Progress renames Progress_Report;
   Always_True : constant Boolean := True;
   K_99 : constant Integer := 99;
   B : Boolean := Support.Identity (1) > 1;
begin
   --  local variable references
   if B then
      Progress.Unexpected (1);
   end if;
   B := Support.Identity (9) < 10;
   if B then
      Progress.Checkpoint (2);
   else
      Progress.Unexpected (3);
   end if;

   --  global variable references
   Bool.B1 := Support.Identity (1) = 1;
   if Bool.B1 then
      Progress.Checkpoint (4);
   else
      Progress.Unexpected (5);
   end if;

   --  function call (ensure function is called just once)
   Bool.I1 := Support.Identity (0);
   if Bool.Is_Positive (Support.Identity (-1)) then
      Progress.Unexpected (6);
   else
      Progress.Checkpoint (7);
   end if;
   if Bool.I1 /= Support.Identity (1) then
```

```
   Progress.Unexpected (8);
end if;

--  Boolean attribute reference
Bool.R1.A := Boolean'Pos (Support.Identity (0) = 1);
Bool.RA (2).A := Boolean'Pos (Support.Identity (1) = 1);
if Boolean'Val (Bool.R1.A) then
   Progress.Unexpected (9);
elsif Boolean'Val (Bool.RA (Support.Identity (2)).A) then
   Progress.Checkpoint (10);
end if;

--  relational and range test expressions
if Support.Identity (1) = Support.Identity (2) then
   Progress.Unexpected (11);
end if;
--  The following relational expression involves Boolean
--  primaries themselves. In this objective, we are only
--  looking for the coverage of the top-level relational
--  expression; not that of the Boolean primaries.
if Bool.Is_Positive (Support.Identity (10)) = True then
   Progress.Checkpoint (12);
end if;

--  expressions exercised both ways and never exercised
for I in Support.Identity (0) .. 1 loop
   if I in 1 .. Support.Identity (0) then
      Progress.Unexpected (13);
   elsif Bool.Is_Positive (I) then
      Progress.Checkpoint (14);
   end if;
   Bool.I1 := Support.Identity (I - 1);
   if abs (I) < Support.Identity (0) then
      Progress.Unexpected (15);
   elsif Bool.Is_Positive (I + 1) then
      Progress.Checkpoint (16);
      if Bool.I1 /= I then
         Progress.Unexpected (17);
      end if;
   end if;
   if I not in Bool.RA'Range then
      Progress.Checkpoint (18);
   end if;
   if I >= Support.Identity (0) then
      Progress.Checkpoint (19);
   elsif I = Bool.RA (I + 1).A then
      Progress.Unexpected (20);
   end if;
end loop;

--  Boolean case selector
case Support.Identity (1) < 0 is
```

```
   when False =>
      Progress.Checkpoint (21);
   when True =>
      Progress.Unexpected (22);
end case;

--  Short-circuit operators represent branching contexts
--  similar to IF-ELSIF construct. They are tested here.
B := Support.Identity (1) > 1;
if B or else Support.Identity (1) = 1 then
   Progress.Checkpoint (23);
else
   Progress.Unexpected (24);
end if;
if B and then Support.Identity (1) < 1 then
   Progress.Unexpected (25);
else
   Progress.Checkpoint (26);
end if;
B := not B;
if Support.Identity (1) < 1 or else B then
   Progress.Checkpoint (27);
else
   Progress.Unexpected (28);
end if;
if B or else Support.Identity (1) = 1 then
   Progress.Checkpoint (29);
else
   Progress.Unexpected (30);
end if;

--  Multiple short-circuit operators
declare
   --  False
   A : constant Boolean := Support.Identity (1) > 1;
   --  True
   B : constant Boolean := Support.Identity (1) = 1;
   --  True
   C : constant Boolean := Support.Identity (1) >= 1;
begin
   if (A and then B)
     or else (B and then C)
     or else (C and then A) then
      Progress.Checkpoint (31);
   else
      Progress.Unexpected (32);
   end if;
end;
--  Constant Boolean expressions
if Always_True then
   Progress.Checkpoint (33);
```

```
   elsif not Always_True then
      Progress.Unexpected (34);
   else
      Progress.Unexpected (35);
   end if;
   if K_99 > 99 then
      Progress.Unexpected (36);
   elsif K_99 < 99 then
      Progress.Unexpected (37);
   else
      Progress.Checkpoint (38);
   end if;
end Foo;
```

(B.1.2) Verify that complex Boolean expressions are handled correctly.

The objective for the tests in this class is to verify that Boolean expressions containing Boolean operators are handled correctly. It should be noted that Level B coverage criteria are not concerned with the coverage of individual terms that constitute a complex Boolean expression, but only with the correct coverage of the overall decision without regards to which term(s) contribute to the outcome of that decision.

(B.1.2.1) Verify that Boolean expressions containing the unary **not** operator are handled correctly.

Example:

```
with Bool;  --  From example in section (B.1.1)
with Support;
with Progress_Report;
procedure Foo_2 is
   package Progress renames Progress_Report;
   B : Boolean := Support.Identity (1) > 1;
begin
   --  Simple variable reference negated
   if not B then
      Progress.Checkpoint (1);
   else
      Progress.Unexpected (2);
   end if;

   --  Negated relational, range test, attribute reference
   if not (Support.Identity (0) = 0) then
      Progress.Unexpected (3);
   elsif not (Support.Identity (7) in Bool.RA'Range) then
      Progress.Unexpected (4);
   elsif not (Boolean'Succ (Support.Identity (0) = 1)) then
      Progress.Unexpected (5);
   else
```

```
         if (not B) and (Support.Identity (1) = 1) then
            Progress.Checkpoint (6);
         else
            Progress.Unexpected (7);
         end if;
         if (Support.Identity (0) = 0) and not B then
            Progress.Checkpoint (8);
         else
            Progress.Unexpected (9);
         end if;
         if (Support.Identity (99) = 1) or (not (not B)) then
            Progress.Unexpected (10);
         else
            Progress.Checkpoint (11);
         end if;
      end if;

      --  Negated function call
      Bool.I1 := Support.Identity (41);
      if not Bool.Is_Positive (Support.Identity (0)) then
         Progress.Checkpoint (12);
      else
         Progress.Unexpected (13);
      end if;
      --  Check that the function was evaluated just once
      if not (Bool.I1 = Support.Identity (1)) then
         Progress.Checkpoint (14);
      end if;
   end Foo_2;
```

(B.1.2.2) Verify that Boolean expressions containing the binary operators **and** and **or** are handled correctly.

Tests involving these two familiar operators can be found in the example in section (B.1.2.3).

(B.1.2.3) Verify that Boolean expressions containing relational operators are handled correctly.

There are a total of 16 distinct binary operators, operating on Boolean terms yielding Boolean result, as shown in the table below.

A	B	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_a	F_b	F_c	F_d	F_e	F_f
F	F	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T
F	T	F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T
T	F	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T
T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
Common Operator Symbols			AND	>		<		XOR	OR	NOR	=		>=		<=	NAND	

The operators **and** (F_1) and **or** (F_7) are perhaps the most commonly used in programming languages. Their negative counterparts, **nand** (F_e) and **nor** (F_8) are common in logic circuits and often realized as **not** (a **and** b) and **not** (a **or** b) in programming languages. Of the twelve, F_0 and F_f are pathological cases equivalent to the Boolean constants False and True respectively. Four others, F_3, F_5, F_a, and F_c are also of little practical value, as they simply copy one of their operands (or its negation) at the output.

The remaining six operators can be realized in Ada through the six relational operators: = (F_9), /= (F_6), < (F_4), <= (F_d), > (F_2), and >= (F_b). Function F_6 can also be represented by the more familiar **xor** operator.

The objective of the tests in this category is to verify that the six relational operators are handled correctly by the coverage analysis tool. Because these tests address coverage analysis at Level B, the primary concern is that the tool correctly recognizes that the two subexpressions that form the operands for these relational operators are still part of the same decision. Tests in this category will not attempt to exercise the tool under all possible values for all operands in a complex Boolean expression. Rather, they will merely determine whether the tool in question recognizes the entire Boolean expression as a single expression and correctly records whether the expression has taken on both values True and False without insisting that each operand take on both values.

Example: The following example combines all useful binary operators into one test.

```
with Progress_Report;
procedure Foo_3 (A, B : in Boolean) is
   package Progress renames Progress_Report;
begin
   if A and B then
      Progress.Checkpoint (1);
   else
      Progress.Checkpoint (2);
   end if;
   if A > B then
      Progress.Checkpoint (3);
```

```
   else
      Progress.Checkpoint (4);
   end if;
   if A < B then
      Progress.Checkpoint (5);
   else
      Progress.Checkpoint (6);
   end if;
   if A /= B then
      Progress.Checkpoint (7);
   else
      Progress.Checkpoint (8);
   end if;
   if A xor B then
      Progress.Checkpoint (9);
   else
      Progress.Checkpoint (10);
   end if;
   if A or B then
      Progress.Checkpoint (11);
   else
      Progress.Checkpoint (12);
   end if;
   if not (A or B) then
      Progress.Checkpoint (13);
   else
      Progress.Checkpoint (14);
   end if;
   if A = B then
      Progress.Checkpoint (15);
   else
      Progress.Checkpoint (16);
   end if;
   if A >= B then
      Progress.Checkpoint (17);
   else
      Progress.Checkpoint (18);
   end if;
   if A <= B then
      Progress.Checkpoint (19);
   else
      Progress.Checkpoint (20);
   end if;
   if not (A and B) then
      Progress.Checkpoint (21);
   else
      Progress.Checkpoint (22);
   end if;
end Foo_3;
```

The expected checkpoints sequences for the four distinct calls possible are:

```
Foo_3 (False, False)    2,4,6,8,10,12,13,15,17,19,21;
```

```
Foo_3 (False, True)          2,4,5,7, 9,11,14,16,18,19,21;
Foo_3 (True, False)          2,3,6,7, 9,11,14,16,17,20,21;
Foo_3 (True, True)           1,4,6,8,10,11,14,15,17,19,22.
```

None of the four calls will yield full decision coverage by itself for any of the ten decisions in Foo_3. Therefore, it is necessary to study coverage following two or more calls. The following table shows a few combinations of calls and coverage that can be expected.

Calls	Full Decision Coverage Expected?									
Foo_3 (x, y)	A and B	A < B	A > B	A xor B	A or B	A nor B	A = B	A >= B	A <= B	A nand B
(F, F)	N	N	N	N	N	N	N	N	N	N
(F, F); (F, T)	N	Y	N	Y	Y	Y	Y	Y	N	N
(F, F); (T, T)	Y	N	N	N	Y	Y	N	N	N	Y
(F, T); (T, F)	N	Y	Y	N	N	N	N	Y	Y	N
(F, T); (T, F); (T, T)	Y	Y	Y	Y	N	N	Y	Y	Y	Y
(F, F); (F, T); (T, F); (T, T)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

(B.1.2.4) Verify that Boolean expressions containing higher-order operators are handled correctly.

Ternary and higher-order operator symbols are not common in programming languages. However, a couple of exceptions can be found in Ada and C. In Ada, a range test involving all Boolean operands represents a ternary operator:

A	B	C	A in B .. C
F	F	F	T
F	F	T	T
F	T	F	F
F	T	T	F
T	F	F	F
T	F	T	T
T	T	F	F
T	T	T	T

In C, the conditional expression A ? B : C (if A then B else C) represents a form of ternary Boolean operator if operands B and C are Boolean:

A	B	C	A ? B : C
F	F	F	F
F	F	T	T
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	F
T	T	F	T
T	T	T	T

However, both cases are rare in practice. The Ada range test A in B .. C boils down to the relatively obscure expression (**not A and not B**) or (**A and C**) in the more familiar notation. In the case of C's conditional expression, A ? B : C is rarely used with Boolean operands B, C. In the unlikely case that B and C are Boolean values or the conditional expression is used in a branching construct, then the expression is equivalent to (**A and B**) **or** (**not A and C**).

The following example illustrates how Ada's range test construct could be tested.

Example:

```
with Progress_Report;
procedure Foo_4 is
   package Progress renames Progress_Report;
   I : Integer := 1;
begin
   for A in Boolean'Range loop
      for B in Boolean'Range loop
         for C in Boolean'Range loop
            if A in B .. C then
               Progress.Checkpoint (I);
            else
               Progress.Checkpoint (I + 1);
            end if;
            I := I + 2;
         end loop;
      end loop;
   end loop;
end Foo_4;
```

The expected checkpoint sequence from this test is: 1, 3, 6, 8, 10, 11, 14, 15.

(B.1.2.5) Verify that Boolean expressions containing multiple operators are handled correctly.

This objective is present to verify that a tool correctly monitors the entire Boolean expression rather than incorrectly monitoring only a portion of the expression. The proper behavior is for the tool to monitor the behavior of the entire expression. While it is permissible for a tool to monitor subexpressions, this is not required.

(B.2) Verify that Boolean expressions in nonbranching constructs are handled correctly.

CAST Position Paper 10 states that all Boolean expressions, regardless of the context in which they appear, should be subject to decision coverage for Level B software. The objective of this section is to establish the coverage analysis tool's compliance with that interpretation. The key to correct coverage analysis is recognizing a Boolean expression as such in various other contexts than the branching constructs dealt with in the Level A tests. To that extent, the tests in this section will concentrate on forms of Boolean expressions that are likely to present challenges in their recognition rather than their complexity.

In the subsections below, four nonbranching contexts are called out explicitly both because they are quite common in applications that deal with a large number of Boolean entities and because some earlier versions of coverage analysis tools did not deal with them correctly. These four contexts are (1) the assignment context, (2) the **for** loop index context, (3) the actual parameter context, and (4) the default expression context. A number of other less commonly occurring contexts are pooled together in a fourth subsection.

(B.2.1) Verify that Boolean expressions in assignment statements are handled correctly.

The objective is to verify that Boolean expressions that appear in the right-hand side expression of an assignment statement are recognized as such and subject to decision coverage.

Example:

```
package Bool_2 is
   subtype On_Off_Type is Boolean;
   On : constant On_Off_Type := True;
   Off : constant On_Off_Type := False;
   Switch : On_Off_Type;

   type CharArr is array (Boolean) of Character;
   type Rec is
      record
         A : Integer;
         B : Boolean;
         C : Character;
         D : CharArr;
      end record;
```

```
   type RecArr is array (Natural range <>) of Rec;

   B1 : Boolean := False;
   CB : CharArr := (True => 'A', others => 'Z ');
   Rc : constant Rec :=
     (10, False, 'J', (False => 'f', True => 't'));
   R1 : Rec := Rc;
   RA : RecArr := (101 .. 200 => Rc);
   I1 : Integer := 0;

   function Is_Positive (I : Integer) return On_Off_Type;
   --  Return On if I > 0, Off otherwise.  In addition,
   --  the global variable I1 is incremented on each call.

end Bool_2;

--  Conditions that should be reported as evaluating to
--  True only are underlined, those evaluating to False
--  only are italicized, those evaluating to both True and
--  False are underlined and italicized, and those that are
--  never evaluated are shown in bold face.
with Bool_2;
with Support;
with Progress_Report;
procedure Foo_5 is
   package Progress renames Progress_Report;
   Always_True : constant Boolean := True;
   A : Boolean;
   B : Boolean := Support.Identity (1) > 1;
begin
   --  Local variable references
   A := B;
   B := Support.Identity (9) < 10;
   A := A or B;

   --  Global variable references
   Bool_2.Switch := Bool_2.Rc.D (A) = Support.Ident_C ('t');

   --  Function call
   for I in Integer range -1 .. 0 loop
      A := Bool_2.Is_Positive (I);
   end loop;

   --  Boolean attribute reference
   Bool_2.R1.B := Bool_2.CharArr'First (Support.Identity (1));
   Bool_2.RA (2).B := Support.Identity (1) = 1;

   --  Relational and range test expressions
   B := Bool_2.RA (Bool_2.Rc.A).A in Bool_2.RA'Range;
   B := B or (Bool_2.RA (Support.Identity (Bool_2.RA'First)).A
              < 100);
```

```
   --  Expressions exercised both ways or never exercised
   for I in Support.Identity (0) .. 1 loop
      Bool_2.R1.B := Bool_2.Is_Positive (I);
      Bool_2.RA (I).B := abs (I) < Support.Identity (0);
      Bool_2.RA (abs I).B := Bool_2.Is_Positive (I + 1);
      Bool_2.RA (I mod 2).B := I not in Bool_2.RA'Range;
      Bool_2.RA ((I + 1)/2).B := I >= Support.Identity (0)
                 or else
                 I = Bool_2.RA (I + 1).A;
   end loop;

   A := Always_True;
   B := not Always_True;
end Foo_5;
```

(B.2.2) Verify that Boolean **for** loop index variables are handled correctly.

Tests in this section will verify that decision coverage is correctly tracked and reported for a **for** loop index if its type is Boolean.

Example:

```
with Progress_Report;
procedure Foo_6 is
   package Progress renames Progress_Report;
   subtype On_Off_Type is Boolean;
   subtype Always_On is Boolean range True .. True;
   subtype Never_On  is Boolean range False .. False;
   subtype Null_Bool is Boolean range True .. False;
begin
   for B in On_Off_Type loop
      Progress.Checkpoint (1);
   end loop;
   for B in Always_On loop
      Progress.Checkpoint (2);
   end loop;
   for B in Never_On loop
      Progress.Checkpoint (3);
   end loop;
   for B in Null_Bool loop
      Progress.Unexpected (4);
   end loop;
end Foo_6;
```

(B.2.3) Verify that Boolean actual parameters are handled correctly.

Tests in this section are written to verify that decision coverage is correctly tracked and reported for Boolean expressions when they occur in the context of actual parameters of functions and procedures.

It should be noted that only the IN mode parameters (and the input sides of IN OUT mode parameters) are of concern for coverage. It is assumed that coverage of OUT mode parameters (and the output side of IN OUT parameters) will be addressed in conjunction with coverage for the subprogram in question.

Example:

```
package Bool_3 is
   subtype On_Off_Type is Boolean;
   On : constant On_Off_Type := True;
   Off : constant On_Off_Type := False;
   Switch : On_Off_Type;

   type BoolArr is array (Character) of Boolean;
   type Rec is
      record
         A : Integer;
         B : Boolean;
         C : Character;
         D : BoolArr;
      end record;
   type RecArr is array (Natural range <>) of Rec;

   B1 : Boolean := False;
   CB : BoolArr := ('A' .. 'Z' => True, others => False);
   Rc : constant Rec :=
     (10, False, 'J', ('a' .. 'z' => False, others => True));
   R1 : Rec := Rc;
   RA : RecArr := (1 .. 100 => Rc);
   I1 : Integer := 0;

   function Is_Positive (I : Integer) return On_Off_Type;
   --  Return On if I > 0, Off otherwise.  In addition,
   --  the global variable I1 is incremented on each call.

   function Add_Bools (A, B : in Boolean) return Integer;
   --  Add the numerical values of A and B, using False = 0
   --  and True = 1.

   procedure Codify (Variable : out Boolean;
                     Set_Not2 : in  Boolean);
   --  Sets Variable to opposite of Set_Not2

   procedure Modify (Variable  : in out Boolean;
                     New_Value : in Boolean);
   --  Sets Variable to New_Value
```

```
end Bool_3;

--  Conditions that should be reported as evaluating to
--  True only are underlined, those evaluating to False
--  only are italicized, those evaluating to both True and
--  False are underlined and italicized, and those that are
--  never evaluated are shown in bold face.
with Bool_3;
with Support;
with Progress_Report;
procedure Foo_7 is
   package Progress renames Progress_Report;
   Always_True : constant Boolean := True;
   A : Boolean;
   B : Boolean := Support.Identity (1) > 1;
begin
   --  Local variable references
   Progress.Verify_Bool (B, False, 1);
   Progress.Verify_Bool (Support.Identity (9) < 10, True, 2);
   Progress.Checkpoint (3);
   A := False;
   Bool_3.Codify (A, A or B);
   Progress.Checkpoint (4);

   --  Global variable references
   Bool_3.Codify (Bool_3.Switch, not A);
   Progress.Checkpoint (5);

   A := True;
   --  Function call
   for B in Boolean'Range loop
      Bool_3.Modify (A, B);
      Progress.Checkpoint (6);
   end loop;

   --  Boolean attribute reference
   Progress.Verify_Bool (Support.Identity (0) = 1, False, 7);
   Bool_3.Codify (Bool_3.RA (2).B, Support.Identity (1) = 1);
   Progress.Checkpoint (8);

   --  Relational and range test expressions
   Progress.Verify_Bool (Bool_3.Is_Positive
                           (Support.Identity (10)),
                         True, 9);

   --  Expressions exercised both ways and never exercised
   B := True;
   Bool_3.R1.D ('b') := True;
```

```
      Progress.Checkpoint (10);

      for I in Support.Identity (0) .. 1 loop
         Bool_3.Codify (Bool_3.R1.B, Bool_3.Is_Positive (I));
         Progress.Checkpoint (11);
         Bool_3.R1.D ('a') := abs (I) < Support.Identity (0);
         Progress.Checkpoint (12);
         Bool_3.Modify (Bool_3.R1.D ('a'), True);
         Progress.Checkpoint (13);
         Bool_3.Codify (Bool_3.R1.D ('b'), B);
         Progress.Checkpoint (14);
         Progress.Verify_Bool (Bool_3.Add_Bools
                                  (Bool_3.R1.D ('b'), B) = 1,
                               True, 15);
         B := not B;
         Progress.Checkpoint (16);
         Bool_3.R1.D ('c') := I not in Bool_3.RA'Range;
         Progress.Checkpoint (17);
         Progress.Verify_Bool (Bool_3.R1.D ('c'),
                               not Boolean'Val (I),18);
      end loop;

      A := Always_True;
      B := not Always_True;
      Progress.Verify_Bool (Bool_3.Add_Bools (A, B) = 1,
                            True, 19);
   end Foo_7;
```

(B.2.4) [Ada only] Verify that Boolean expressions used as "default expressions" in subprogram parameter specifications, discriminant specifications, and record component specifications are handled correctly.

Ada allows the user to specify default expressions, which are used to supply values for objects or parameters omitted from certain constructs. There are at least three contexts in which the user is allowed to omit an expression yet have one supplied by a default expression specified elsewhere.

A subprogram actual parameter expression may be omitted from a call provided the corresponding actual parameter has an associated default expression.

A discriminant value may be omitted from an object declaration provided the corresponding type declaration includes a default expression for the discriminant.

A record object without an initializing aggregate may nevertheless have one or more of its components implicitly initialized provided those components are declared with default expressions in the record type declaration.

Tests in this category are designed to verify that the coverage analysis tool is capable of monitoring and reporting coverage of default expressions that are of the type Boolean or contain subexpressions of the type Boolean.

In each case noted above, the default expression is written where the subprogram or type is declared, but it is evaluated every place where a call or an object declaration is coded without supplying a corresponding expression. This leads to an implicit evaluation of the default expression at those sites. Consider the following example:

```
procedure Example (A : in Integer := 63;
                   B : in Boolean := False);
--
-- now a few calls:
(a)     Example (7, True);  -- param B is explicitly supplied
(b)     Example (I);        -- B is omitted, defaults to False
(c)     Example (B => False);  -- A is omitted, defaults to 63
(d)     Example;            -- Both A, B omitted, default to
                            -- 63, False, same as call above.
```

In examples (b) and (d), there is a Boolean expression implicitly evaluated for which there is no textual representation at the call site. In this simple example, that Boolean expression is the constant False, whose coverage is more of academic interest than of practical use. However, if that same expression consists of a nonstatic expression, as would be the case if the procedure had been declared differently (for example, where On_Ground is a (global) Boolean variable), then calls such as (b) and (d) should be exercised at least twice to obtain coverage for the missing parameter B.

```
procedure Example (A : in Integer := 63;
                   B : in Boolean := On_Ground);
```

Next, the question of how coverage of default expression should be measured is considered. Because there is only one place where the default expression is coded in the source language, it could be argued that it is sufficient to achieve coverage through a combination of all the calls made to the subprogram. For example, under this interpretation, executing call (b) when On_Ground is False and executing call (d) when On_Ground is True should suffice to demonstrate coverage of the default expression. On the other hand, a compiler may generate code to evaluate any omitted arguments at each call site, leading to a model similar to:

```
(b)     Example (I);     => Example (I, On_Ground);

(d)     Example;   => Example (63, On_Ground);
```

where implicit evaluations are shown in the single boxes.

If indeed the compiler implicitly evaluates each missing argument at each call site, obtaining coverage at the declaration site will not be sufficient to meet decision coverage criterion. Therefore, the test suite should include test cases that discriminate between the two approaches.

(B.2.4.1) Verify that Boolean expressions used in "default expressions" are monitored for coverage at the point where the expression is written.

The following examples all use a package named Defp along with distinct test programs, Foo_1, Foo_2,… Foo_4, to illustrate a distinct level of coverage expected for the default expressions contained in Defp. The examples use only subprogram parameter default expressions, but the principles hold for other forms of default expressions as well.

```
package Defp is
  function G (B : Boolean := True) return Boolean;
  -- When B is False, G returns False. When B is omitted or
  -- True, G returns the opposite of what it returned on
  -- last call.
  -- If the first call is G (True), it returns True.
  function H (A : Boolean := G) return Integer;
  -- if B = True return 10, else return 9.
  procedure P (X : out Boolean;
               Y : in Integer := H (G);
               Z : in Integer := H);
  -- Set X to G (Y = Z).
end Defp;
```

Example (1):

```
-- Conditions that should be reported as evaluating to
-- True only are underlined, those evaluating to False
-- only are italicized, those evaluating to both True and
-- False are underlined and italicized, and those that are
-- never evaluated are shown in bold face.

with Defp;
with Progress_Report;
procedure Foo_1 is
  package Progress renames Progress_Report;
begin
  if Defp.G (not Defp.G (Defp.G)) then
    Progress.Checkpoint (10);
  else
    Progress.Unexpected (20);
```

```
  end if;
  if Defp.H (False) < 0 then
    Progress.Unexpected (30);
  end if;
end Foo_1;
```

Decision coverage in Defp resulting from calling Foo_1 once is indicated below (comments have been deleted to reduce clutter):

```
package Defp is
  function G (B : Boolean := True) return Boolean;
  function H (A : Boolean := G) return Integer;
  procedure P (X : out Boolean;
               Y : in Integer := H (G);
               Z : in Integer := H);
end Defp;
```

The default expression for parameter B in function G is evaluated once, due to the omitted argument for the innermost call to G in the **if** statement in Foo_1. The default expression in function H is not evaluated because the only call to H in Foo_1 supplies the argument explicitly. The default expressions in procedure P are not evaluated because there are no calls to P in Foo_1.

Example (2):

```
with Defp;
with Progress_Report;
with Support;
procedure Foo_2 is
  package Progress renames Progress_Report;
begin
  if Defp.H (Support.Identity (9) = 9) > 10 then
    Progress.Unexpected (10);
  elsif Defp.G (False) or else Defp.H = 10 then
    Progress.Checkpoint (20);
  elsif Defp.H < 10 then
    Progress.Unexpected (30);
  end if;
end Foo_2;
```

Decision Coverage in Defp resulting from calling Foo_2 once is indicated below:

```
package Defp is
  function G (B : Boolean := True) return Boolean;
  function H (A : Boolean := G) return Integer;
  procedure P (X : out Boolean;
               Y : in Integer := H (G);
               Z : in Integer := H);
end Defp;
```

There are three calls to function H and one call to function G in Foo_2. The first call to H explicitly supplies the argument A and, therefore, the default expression is not evaluated. The argument evaluates to True (9 = 9) and therefore H returns 10. This makes the decision in the **if** clause False, causing the first **elsif** clause to be evaluated. The condition in that **elsif** clause has the short-circuit form: **A or else B**. Evaluation of the first term results in a call to G with an explicitly supplied argument False; G returns False, causing the second term to be evaluated. That evaluation calls H without an argument, causing the default expression G to be evaluated, which in turn results in a call to G with its default parameter True evaluated. Calling G (True) returns True because the previous call to G had returned False. Now, the evaluation of H (True) returns 10, which is equal to 10 making the second term of the short-circuit decision True, and hence the overall decision True as well. Because the first elsif decision evaluates to True, the second elsif's decision (another call to H) is not evaluated. Finally, procedure P is never called from Foo_2.

Example (3):

```
with Defp;
with Progress_Report;
procedure Foo_3 is
  package Progress renames Progress_Report;
  A : Boolean := Defp.G (False) or else Def.G;
begin
  Defp.P (A, Z => 10);
  if A then
    Progress.Unexpected (10);
  else
    Progress.Checkpoint (20);
  end if;
end Foo_3;
```

Decision Coverage in Defp resulting from calling Foo_3 once is indicated below:

```
package Defp is
  function G (B : Boolean := True) return Boolean;
  function H (A : Boolean := G) return Integer;
  procedure P (X : out Boolean;
               Y : in Integer := H (G);
               Z : in Integer := H);
end Defp;
```

Foo_3 begins by calling G to initialize its Boolean variable A. The first call, resulting from the first term of the **or else** short-circuit form, passes the argument False explicitly. G returns False for that call, leading to the evaluation of the second term in the short-circuit form. That term invokes G again, but without an argument. G returns True, which initializes A.

The first statement in the body of Foo_3 calls P, omitting the argument corresponding to Y. This causes the default expression H (G) to be evaluated. The call to G returns False (the opposite of what it returned last), and the call to H returns 9. Because Z is 10, the out parameter X gets set to False, which sets the actual parameter A to False. Thus, there are no calls to H needing default expression evaluation.

Example (4):

```
with Defp;
with Progress_Report;
procedure Foo_4 is
  package Progress renames Progress_Report;
  A : Boolean := Defp.G (False) or else Def.G;
begin
  Defp.P (A, Y => 10);
  Defp.P (A);
  if A then
    Progress.Unexpected (10);
  else
    Progress.Checkpoint (20);
  end if;
end Foo_4;
```

Decision Coverage in Defp resulting from calling Foo_3 once is indicated below:

```
package Defp is
  function G (B : Boolean := True) return Boolean;
  function H (A : Boolean := G) return Integer;
  procedure P (X : out Boolean;
               Y : in Integer := H (G);
               Z : in Integer := H);
end Defp;
```

Foo_4 begins by calling G twice to initialize its local variable A to True as in Foo_3. In the body of Foo_4, the first statement invokes P omitting argument Z. That causes the default expression H to be evaluated. That evaluation causes the default expression for its parameter A, namely G, to be evaluated. G returns False, and H (False) returns 9. With Y being 10, procedure P sets its formal parameter X to False, which sets A to False.

The next statement in Foo_4 calls P again, this time omitting both Y and Z parameters. Both default expressions call H; the first one, corresponding to Y, calls G to get a return value of True. H (True) returns 10. The second call to H, leads to a call to G without an argument to get a result of False, and H (False) returns 9. Because 10 does not equal 9, A gets set to False. It is worth noting that the default expression G of parameter A in H does not get full coverage

because the last call to G, even though it returns True, is not the result of evaluation of the default expression for A.

(B.2.4.2) Verify that Boolean expressions used in "default expressions" are monitored for coverage at each point where the expression is implicitly evaluated.

The same examples from (B.2.4.1) are used to illustrate coverage monitoring at call sites. Coverage of implicitly evaluated default expressions is indicated by duplicating the statement containing the implicit evaluations as comments with the default expressions explicitly indicated in italics. The explanations provided in (B.2.4.1) should be helpful here as well.

Example (1):

```
-- Conditions that should be reported as evaluating to True
-- only are underlined, those evaluating to False only are
-- italicized, those evaluating to both True and False are
-- underlined and italicized, and those that are never
-- evaluated are shown in bold face.

with Defp;
with Progress_Report;
procedure Foo_1 is
  package Progress renames Progress_Report;
begin
  if Defp.G (not Defp.G (Defp.G)) then
  -- if Defp.G (not Defp.G (Defp.G (True))) then
    Progress.Checkpoint (10);
  else
    Progress.Unexpected (20);
  end if;
  if Defp.H (False) < 0 then
    Progress.Unexpected (30);
  end if;
end Foo_1;
```

Example (2):

```
with Defp;
with Progress_Report;
with Support;
procedure Foo_2 is
  package Progress renames Progress_Report;
begin
  if Defp.H (Support.Identity (9) = 9) > 10 then
    Progress.Unexpected (10);
  elsif Defp.G (False) or else Defp.H = 10 then
  -- elsif Defp.G (False) or else Defp.H (G) = 10 then
    Progress.Checkpoint (20);
  elsif Defp.H < 10 then
```

```
  -- elsif Defp.H (G) < 10 then
    Progress.Unexpected (30);
  end if;
end Foo_2;
```

Example (3):

```
with Defp;
with Progress_Report;
procedure Foo_3 is
  package Progress renames Progress_Report;
  A : Boolean := Defp.G (False) or else Def.G;
  -- A : Boolean := Defp.G (False) or else Def.G (True);
begin
  Defp.P (A, Z => 10);
  -- Defp.P (A, Y => H (G), Z => 10);
  if A then
    Progress.Unexpected (10);
  else
    Progress.Checkpoint (20);
  end if;
end Foo_3;
```

Example (4):

```
with Defp;
with Progress_Report;
procedure Foo_4 is
  package Progress renames Progress_Report;
  A : Boolean := Defp.G (False) or else Def.G;
  -- A : Boolean := Defp.G (False) or else Def.G (True);
begin
  Defp.P (A, Y => 10);
  -- Defp.P (A, Y => 10, Z => H (G));
  Defp.P (A);
  -- Defp.P (A, Y => H (G), Z => H (G));
  if A then
    Progress.Unexpected (10);
  else
    Progress.Checkpoint (20);
  end if;
end Foo_4;
```

(B.2.5) [Ada only] Verify that Boolean expressions used in various other nonbranching contexts are handled correctly.

There are a number of contexts other than assignment, procedure call, and default expressions where a Boolean expression might occur in an Ada program:

Array subscript

Component of an aggregate
Actual parameter of a generic instantiation
Entry barrier in an task entry declaration
Guard expression in a selective accept

Boolean expressions in all these contexts are subject to decision coverage under CAST Position Paper 10. Test cases should be included in the test suite to ensure the automated coverage analysis tools are designed to handle them correctly, especially because of their relative rarity in embedded applications.

(B.3) [Ada only] Verify that derived Boolean types are handled correctly.

When a new Boolean type is derived in a package specification, the new type could be used in place of the predefined BOOLEAN type in almost all contexts. A coverage analysis tool should be capable of identifying expressions of such new types and subject them to full decision coverage analysis as it would expressions of the predefined type.

(B.3.1) Verify that derived Boolean types are handled correctly in branching contexts.

Example:

```
package DT is
  type MyBool is new Boolean;
  function "+" (A, B : MyBool) return MyBool;
  -- Same as "or"
  function "*" (A, B : MyBool) return MyBool;
  -- Same as "and"
  function "-" (A, B : MyBool) return MyBool;
  -- Same as "xor"
  function Eql (A, B : MyBool) return MyBool;
  -- Same as "="
  function Sum (A, B : MyBool) return Integer;
  -- Returns sum of A, B using False = 0, True = 1
end DT;

-- Conditions that should be reported as evaluating to
-- True only are underlined, those evaluating to False
-- only are italicized, those evaluating to both True and
-- False are underlined and italicized, and those that are
-- never evaluated are shown in bold face.
with DT;
with Support;
with Progress_Report;
procedure Foo is

  package Progress renames Progress_Report;
  use type DT.MyBool;

  X, Y : DT.MyBool;
```

```
begin
  X := False;
  Y := not X;
  if X then
    Progress.Unexpected (10);
  elsif Y then
    Progress.Checkpoint (20);
  else
    Progress.Unexpected (30);
  end if;

  -- Try some of the predefined operators
  if (X and not Y) or (not X and Y) then
    Progress.Checkpoint (40);
  else
    Progress.Unexpected (50);
  end if;

  -- Try the same operators using prefix notation
  if "or" ("and" (X, "not" (Y)), "and" ("not" (X), Y)) then
    Progress.Checkpoint (60);
  else
    Progress.Unexpected (70);
  end if;

  -- Try the user-defined operators
  for A in DT.MyBool'Range loop
   for B in DT.MyBool'Range loop
    I := DT.Sum (A, B);
    case I is
    when 0 =>
       Progress.Verify_Bool (Boolean (A + B),      False, 111);
       Progress.Verify_Bool (Boolean (A * B),      False, 112);
       Progress.Verify_Bool (Boolean (A - B),      False, 113);
       Progress.Verify_Bool (Boolean (Eql (A, B)), True, 114);
     when 1 =>
       Progress.Verify_Bool (Boolean (A + B),      True,  121);
       Progress.Verify_Bool (Boolean (A * B),      False, 122);
       Progress.Verify_Bool (Boolean (A - B),      True,  123);
       Progress.Verify_Bool (Boolean (Eql (A, B)), False, 124);
     when 2 =>
       Progress.Verify_Bool (Boolean (A + B),      True,  131);
       Progress.Verify_Bool (Boolean (A * B),      False, 132);
       Progress.Verify_Bool (Boolean (A - B),      True,  133);
       Progress.Verify_Bool (Boolean (Eql (A, B)), False, 134);
     when 3 =>
       Progress.Verify_Bool (Boolean (A + B),      True,  141);
       Progress.Verify_Bool (Boolean (A * B),      True,  142);
```

```
        Progress.Verify_Bool (Boolean (A - B),      False, 143);
        Progress.Verify_Bool (Boolean (Eql (A, B)), True,  144);
    when others =>
        Progress.Unexpected (150);
     end case;
    end loop; -- B
   end loop; -- A

end Foo;
```

(B.3.2) Verify that derived Boolean types are handled correctly in nonbranching contexts.

Tests in this category can be derived from tests in (B.2.1), (B.2.2), and (B.2.4) as illustrated in the example below.

Example:

```
package DT is
  type MyBool is new Boolean;
  function "+" (A, B : MyBool) return MyBool;
  -- Same as "or"
  function "*" (A, B : MyBool) return MyBool;
  -- Same as "and"
  function "-" (A, B : MyBool) return MyBool;
  -- Same as "xor"
  function Eql (A, B : MyBool) return MyBool;
  -- Same as "="
  function Sum (A, B : MyBool) return Integer;
  -- Returns sum of A, B using False = 0, True = 1
end DT;

with DT;
package Bool is
  use type DT.MyBool;
  subtype On_Off_Type is DT.MyBool;
  On : constant On_Off_Type := DT.True;
  Off : constant On_Off_Type := DT.False;
  Switch : On_Off_Type;

  type BoolArr is array (Character) of DT.MyBool;
  type Rec is
  record
    A : Integer;
    B : DT.MyBool;
    C : Character;
    D : BoolArr;
  end record;
  type RecArr is array (Natural range <>) of Rec;

  B1 : DT.MyBool := DT.False;
  CB : BoolArr := ('A' .. 'Z' => DT.True, others => DT.False);
```

```
  Rc : constant Rec :=
       (10, DT.False, 'J',
      ('a' .. 'z' => DT.False, others => DT.True));
  R1 : Rec := Rc;
  RA : RecArr := (1 .. 100 => Rc);
  I1 : Integer := 0;

  function Is_Positive (I : Integer) return On_Off_Type;
  -- return True if I > 0, false otherwise; in addition,
  -- the global variable I1 is incremented on each call.

  function Add_Bools (A, B : in DT.MyBool) return Integer;
  -- Add the numerical values of A and B, using False = 0
  -- and True = 1.

  procedure Verify (Actual   : in DT.MyBool;
                    Expected : in DT.MyBool);
  -- Verifies that the Boolean value Actual equals Expected

  procedure Codify (Variable : out DT.MyBool;
                    Set_Not2 : in  DT.MyBool);
  -- Sets Variable to opposite of Set_Not2

  procedure Modify (Variable : in out DT.MyBool;
                    New_Value: in DT.MyBool);
  -- Sets Variable to New_Value

end Bool;

-- Conditions that should be reported as evaluating to
-- True only are underlined, those evaluating to False
-- only are italicized, those evaluating to both True and
-- False are underlined and italicized, and those that are
-- never evaluated are shown in bold face.
with DT;
with Bool;
with Support;
with Progress_Report;
procedure Foo is
  use type DT.MyBool;
  package Progress renames Progress_Report;
  Always_True : constant DT.MyBool := DT.True;
  K_99 : constant Integer := 99;
  A : DT.MyBool;
  B : DT.MyBool := DT.MyBool (Support.Identity (1) > 1);
begin
  -- local variable references
  Progress.Checkpoint (10);
  Bool.Verify (B, DT.False);
  Progress.Checkpoint (20);
  Bool.Verify (DT.MyBool (Support.Identity (9) < 10),
               DT.True);
  Progress.Checkpoint (30);
```

```
A := DT.False;
Bool.Codify (A, A or B);
Progress.Checkpoint (40);

-- global variable references
Bool.Codify (Bool.Switch, not A);
Progress.Checkpoint (50);

A := DT.True;
-- function call
for B in DT.MyBool'Range loop
  Bool.Modify (A, B);
  Progress.Checkpoint (60);
end loop;

-- DT.MyBool attribute reference
Bool.Verify (DT.MyBool (Support.Identity (0) = 1),
             DT.False);
Progress.Checkpoint (70);
Bool.Codify (Bool.RA (2).B,
             DT.MyBool (Support.Identity (1) = 1));
Progress.Checkpoint (80);

-- relational and range test expressions
Bool.Verify (Bool.Is_Positive (Support.Identity (10)),
             DT.True);
Progress.Checkpoint (90);

-- expressions exercised both ways and never exercised
B := DT.True;
Bool.R1.D ('b') := DT.True;
Progress.Checkpoint (100);

for I in Support.Identity (0) .. 1 loop
  Bool.Codify (Bool.R1.B, Bool.Is_Positive (I));
  Progress.Checkpoint (110);
  Bool.R1.D ('a') :=
    DT.MyBool (abs (I) < Support.Identity (0));
  Progress.Checkpoint (120);
  Bool.Modify (Bool.R1.D ('a'), DT.True);
  Progress.Checkpoint (130);
  Bool.Codify (Bool.R1.D ('b'), B);
  Progress.Checkpoint (140);
  Bool.Verify
    (DT.MyBool (Bool.Add_Bools(Bool.R1.D ('b'), B) = 1),
     DT.True);
  B := not B;
```

```
    Progress.Checkpoint (150);
    Bool.R1.D ('c') := DT.MyBool (I not in Bool.RA'Range);
    Progress.Checkpoint (160);
    Bool.Verify (Bool.R1.D ('c'), not DT.MyBool'Val (I));
    Progress.Checkpoint (170);
  end loop;

  A := Always_True;
  B := not Always_True;
  Bool.Verify (DT.MyBool (Bool.Add_Bools (A, B) = 1),
               DT.True);
  Progress.Checkpoint (180);
end Foo;
```

(B.4) [Ada only] Verify that Boolean operators are handled correctly under overloading and renaming.

Ada operators can be overloaded (i.e., redefined and used in conjunction with other operators of the same name applying to other operand types) or renamed (i.e., given a new function name or operator symbol) so as to obscure their original identity. Although such uses are often discouraged or disallowed as unsafe programming practice, a software verification tool may not be able to assume that such restrictions apply to the software at hand or that they are enforced rigorously. The tests in this category are intended to evaluate the extent to which the coverage analysis tool is able to discriminate between syntax-evident Boolean expressions and those that are hidden under the cloak of renaming and overloading. Examples in the subsections below use the following package, aptly named Obscure.

```
package Obscure is

  type InverseBool is new Boolean;
  F : constant InverseBool := False;
  T : constant InverseBool := True;
  False : constant InverseBool := T;
  True  : constant InverseBool := F;

  function "and" (A,B : InverseBool) return InverseBool;
  function "or"  (A,B : InverseBool) return InverseBool;

end Obscure;

package body Obscure is
  function "and" (A,B : InverseBool) return InverseBool is
  begin
    if A = True and
       B = True then
      return True;
    else
```

```
      return False;
    end if;
  end "and";
  function "or"  (A,B : InverseBool) return InverseBool is
  begin
    if A = False and
       B = False then
      return False;
    else
      return True;
    end if;
  end "and";
end Obscure;
```

(B.4.1) Verify that overloaded Boolean operators are handled correctly in branching contexts.

Example: The following example invokes the overriding definitions of **and** and **or** over the derived type InverseBool. If the intrinsic versions of these functions are invoked, the test will take a wrong branch, leading to an unexpected checkpoint.

```
with Support;
with Progress_Report;
with Obscure;
procedure Foo is
  package Progress renames Progress_Report;
  use type Obscure.InverseBool;
  A, B : Obscure.InverseBool;
begin
  A := Obscure.False;
  if not A then
    Progress.Unexpected (10);
  else
    Progress.Checkpoint (20);
  end if;
  B := Obscure.True;
  if (A or not B) then
    Progress.Checkpoint (30);
  else
    Progress.Unexpected (40);
  end if;
  B := Obscure.True;
  if (A or B) then
    Progress.Unexpected (50);
  else
    Progress.Checkpoint (60);
  end if;
end Foo;
```

(B.4.2) Verify that overloaded Boolean operators are handled correctly in nonbranching contexts.

Tests similar to those outlined in section (B.2) can be employed in conjunction with a package such as Obscure to verify that derived types and overloaded functions are correctly handled in nonbranching contexts.

(B.5) Verify that entry and exit points are handled correctly.

As noted in the introduction to Level B (decision coverage) test requirements, explicit entry and exit points of Ada and C programs are adequately addressed by Level C (statement coverage) test requirements. Therefore, this subsection delves mostly in implicit entry and exit points.

(B.5.1) Verify that subprogram entry points are handled correctly.

(B.5.1.1) Normal (top of subprogram) entry points.

Normal subprogram entry points of Ada and C programs are adequately addressed by Level C test requirements.

(B.5.1.2) Implicit entry points (exception handlers).

In Ada, implicit entry points consist of entry points of exception handlers. Coverage of these entry points are also adequately addressed by Level C test requirements, as such coverage is equivalent to statement coverage for the first statement in the handler.

In C, exception handlers are ordinary functions and, as in Ada, the coverage objectives of implicit entry points are met by those of Level C.

(B.5.2) Verify that subprogram exit points are handled correctly.

(B.5.2.1) Explicit return statements.

The coverage objectives of explicit return statement are met by the test requirements of Level C.

(B.5.2.2) Implicit return statements.

In both C and Ada, an implicit return is generated by the compiler to be executed if control flow reaches the physical end of the subprogram. In Ada, such implicit returns are valid only in procedures; all functions must execute an explicit return or else a Program_Error will be raised. In ANSI C, however, any function may drop off the end and will return garbage (which makes for interesting challenges in testing!). In safety-critical applications, such returns are usually forbidden by

coding standards. However, because the language does permit implicit returns, the test suite for C should include tests for implicit return coverage.

Example:

```
procedure Foo (A : Integer;
                  B : out Integer) is
begin
  B := A - 1;
  if A > 0 then
    B := A;
    return;
  end if;
end Foo;
```

Calling Foo with a positive value for A should return that value in B and coverage should indicate that the implicit return at the end of the procedure was not executed. On the other hand, calling Foo with a zero for A should return a -1 in B and should show deficiency in the coverage of the decision A> 0.

How many implicit exit points could there be in an Ada procedure? Consider the following program segment:

```
procedure Foo (A : Integer;
                  B : out Integer) is
begin
  B := A - 1;
  if A > 0 then
    B := A;
    return;
  end if;
exception
when Constraint_Error =>
  B := 0;
when others =>
  B := Integer'Last;
end Foo;
```

It is debatable whether this procedure has one or three implicit exit points; it depends to some extent on the code generated by the compiler. From an abstract semantic viewpoint, any procedures can be modeled as having at most one implicit return at the end of the subprogram; any other implicit exit point becomes a branch to this common implicit return. It is recommended that this semantic model be used to avoid becoming compiler dependent within the test suite.

(B.5.2.3) RAISE statements.

Ada RAISE statements are another form of explicit exit from a subprogram. Coverage of these exits is, however, implied in the statement coverage at Level C. Therefore, there is no need to develop separate tests for this case.

(B.5.2.4) Implicit exit points (via exceptions raised implicitly).

In addition to the RAISE statement, which is a form of explicit exit, an Ada subprogram may exit abruptly if an exception is raised implicitly. Exact circumstances under which such implicit exception may be raised are not specified in Ada; instead, it is left up to the implementation to determine when and where an implicit exception is raised. For example, if an access value is corrupted or uninitialized, dereferencing that value may cause access to addresses beyond the address space of the program. Such an access may be intercepted by the hardware and an exception raised, or ignored and garbage returned in response. Either model is a legal implementation in Ada, because the program making such an access request is erroneous in the first place.

Given this uncertainty, it is not practical to monitor and report subprogram exits resulting from implicitly raised exceptions. Even if some software verification tools are capable of doing this, perhaps based on object code generated by the compiler, it is impractical to devise test cases to evaluate that capability while remaining compiler-independent. Tests developed to exercise implicit exit point coverage should be treated as compiler-dependent and selected only if applicable.

(B.5.3) Verify that construct (low level) entry points are handled correctly.

Though uncommon elsewhere, DO-248B refers to various alternatives of a **case** or **switch** statement, or the two legs of an **if** statement as entry points. Discussions in DO-248B indicate that there are low-level entry points, called construct entry points, that should be covered at Level B. These are points in the program to which control can be transferred in a manner other than normal sequential flow. For example, there would be a construct entry point for each alternative sequence of statements in a **case** statement. A test is said to cover such an entry point if that point is reached via the nonsequential transfer of control.

In Ada, a case alternative cannot be reached via a sequential flow of control; it can only be entered due to the transfer of control effected by the value of the selector expression. Furthermore, Ada does not allow an implicit default case, as C does. For these reasons, statement coverage implies coverage of the construct entry points.

In C, control can be allowed to flow from one case alternative to the next sequentially. Therefore, statement coverage may not always imply entry point

coverage. A set of negative tests (tests that verify coverage deficiency is reported correctly) will be useful to discriminate between the two paths to a case alternative.

The introduction of construct entry point raises new questions about how decision coverage should be defined for constructs that could have multiple paths of control leading to single point in the program. Consider the Ada case statement:

```
case Integer'(I) is
when 1 | 2 =>
  Statement_1;
when 3 .. 99 =>
  Statement_2;
when others =>
  Statement_3;
end case;
```

Statement coverage can be achieved if this **case** statement is executed three times, with I = 1, 3, and 100. Will that also achieve decision coverage per DO-178B and DO-248B? There is no clear indication in either of the guidance documents to answer this question definitively. The position in the test suite is that it will, based on the following rationale.

- There are a number of strategies a compiler could use to produce optimal code for a **case** statement. Assuming that there are distinct branches into a case alternative corresponding to the distinct values of the case expression that select that alternative would be second-guessing the compiler's strategy.

- If the test suite is to minimize implementation dependencies, the **if-elsif-else** model of the case statement semantics is a more suitable model than the branch table model. It will keep the test suite from falsely failing implementations that use other models.

(B.5.4) Verify that construct (low level) exit points are handled correctly.

Construct exit points can be defined as points in the program from which control can be transferred to a nonsequential point within the same subprogram. This definition mirrors the definition of control entry points, which are points to which control is transferred from a nonsequential point in the program. For example, the Ada **exit** statement represents a construct exit point, as does the **goto** statement. Exits from **while** and **for** loops as well as exits via the **exit-when** statement are also examples of construct exits where the nonsequential control transfer is conditional.

Exits from case alternatives (to the end of the **case** statement), exits from **if** clauses (to the end of the **if** statement), and from **block** statements (to its end) are

all construct exits as well. However, these exits, like the **goto** and the unconditional **exit** statement, are of limited interest from a coverage point of view because their coverage is implied by statement coverage. Even the conditional construct exits are not significant from a decision coverage point of view because their coverage can be deduced from the coverage of the overarching statement and the Boolean expression that represents the exit condition.

In summary, the coverage of construct exit points in Ada or C can be deduced from statement coverage and Boolean expression coverage (each decision takes on all possible outcomes), and therefore do not warrant special consideration.

B.4 DISCRIMINATING TESTS FOR LEVEL B.

In Level B tests, variations commonly arise from the following sources:

a. Tracking all Boolean expressions versus tracking only those that appear in a branching context.

b. Not considering Boolean operands of short-circuit operators to be independent decisions.

Level B test requirements outlined above seek to discriminate between these variants. For example, all tests in category (B.2) are expected to fail on tools that do not consider Boolean expressions in nonbranching contexts. Likewise, several suggested tests in (B.1.1) will fail if the verification tool does not treat terms of short-circuit expressions as top-level decisions. Nevertheless, the following simple example provides a specific test case that serves as a discriminator for the above variations.

Example:

```
function Foo (A, B : Integer) return Boolean is
begin
  return
      (A > 0 and then
        (A = abs B))
   or else
      (A <= 0);
end Foo;
```

If decisions outside of branch contexts are recognized, the Boolean expression in the **return** statement should be recognized as a decision. Furthermore, if short-circuit terms are tracked as decisions by themselves, five distinct decisions should be tracked:

```
(1)  A > 0
(2)  A = abs B
(3)  (A > 0 and then (A = abs B)
(4)  A <= 0
(5)  (A > 0 and then (A = abs B) or else (A <= 0)
```

None of the five decisions will attain decision coverage with a single call, such as

```
Foo (A => 0, B => 1)
```

However, a pair of calls such as

```
Foo (A => 0, B => 1) and then Foo (A => 9, B => 1)
```

should provide decision coverage for decisions (1), (4), and (5), leaving decisions (2) and (3) only partially covered.

B.5 REQUIREMENTS FOR LEVEL A TESTS.

Structural coverage requirements for Level A consist of statement coverage, decision coverage and MCDC. To satisfy the Level A test requirements, a tool must first satisfy the Level B and C test requirements. The Level B and C requirements are not repeated here.

The requirements for Level A tests are structured largely along the lines of the Level B tests, namely:

a. Boolean expressions in branching contexts.

MCDC is applicable to all Boolean expressions regardless of context. However, in some languages, an otherwise non-Boolean expression may take on a Boolean variant in certain contexts, such as **if** statements. C is an example of such a language where a numeric expression takes on the Boolean meaning if it occurs in the condition of an **if** statement, **while** loop, or any other condition that controls program flow. Consequently, verification tools developed specifically for such languages may not track MCDC in quite the same way as expressions occurring in all contexts. Therefore, as in the Level B test requirements, special treatment is given to the branching context.

- Simple Boolean expressions
- Various forms of simple Boolean terms including variables, function calls, relational expressions, and attribute references
- Boolean expressions containing Boolean operators
  - Expressions containing uncoupled Boolean terms
    - Expressions containing **and**, **or**, and **not** operators
    - Expressions containing other binary Boolean operators
    - Expressions containing short-circuit operators
  - Expressions containing coupled Boolean terms

- Expressions containing **and**, **or**, and **not** operators
- Expressions containing other binary Boolean operators
- Expressions containing short-circuit operators

b. Boolean expressions in nonbranching contexts.

These test requirements can be divided into subclasses similar to those under Boolean expressions in branching contexts. However, such a subdivision is not very useful because the real issue between branching and nonbranching contexts is whether or not the tool recognizes Boolean expressions as such. Experience with the construction of coverage analysis tools suggests that, once recognized, the coverage treatment of the Boolean expressions is not likely to be dependent on the differences in the context in which they appear. Therefore, the range of tests needed to meet the test requirements in this category could be limited to a sampling of tests from all the subcategories under Boolean expressions in branching contexts.

There are several distinct interpretations of MCDC used by the tools in use today, and there are several other alternatives being studied as possible candidates to replace MCDC in a future revision of DO-178B. For this reason, the test requirements for Level A tests delve into ways of discriminating between the following eight alternative forms of MCDC described in appendix A:

- Unique-Cause MCDC (UCM)
- Black-Box MCDC (BBM)
- Coupled-Cause MCDC (CCM)
- Masking MCDC (MSM)
- Unique-Cause + Masking MCDC (USM)
- Operator Coverage Criterion (OCC)
- Object-Code Branch Coverage (OBC)
- Reinforced Condition/Decision Coverage (RC/DC)

Detailed descriptions of the above variants of MCDC criterion can be found in "Study of Qualification Criteria for Software Verification Tools, Phase 2 Position Paper" [B-1]. These eight variants differ in the composition of the test set that they consider to satisfy the MCDC criterion. Although subtle differences exist in the way Boolean expressions with uncoupled terms are treated by these variants, the most noticeable differences occur in Boolean expressions with coupled terms. Therefore, differences between variants are dealt with in greater depth under the Boolean expressions with coupled terms category. Finally, discussion of distinctions between variants is restricted to branching contexts only, because they are equally applicable to nonbranching contexts.

(A.1) Boolean expressions in branching contexts.

The objective of this section is to verify that Boolean expressions occurring in branching contexts are recognized and tracked properly for MCDC coverage.

(A.1.1) Simple Boolean expressions.

Simple expressions containing no Boolean operators are not of particular interest for the MCDC criterion because for these expressions it reduces to the simple decision coverage criterion addressed in the Level B tests.

(A.1.2) Boolean expressions containing Boolean operators.

The objective of the tests in this category is to verify that Boolean expressions containing various binary Boolean operators and the unary **not** operator are handled correctly. This category is further subdivided into expressions containing uncoupled terms alone and those containing coupled terms.

(A.1.2.1) Boolean expressions containing uncoupled terms only.

The tests in this category verify that Boolean expressions containing Boolean terms, each of which can be set independently of the others, are handled correctly. This category represents the cases in which the five variants of MCDC, namely, UCM, BBM, CCM, MSM, and USM, converge in their interpretation of the DO-178B wording. Consequently, the tests in this category represent the least common denominator that all Level A structural coverage analysis tools should be expected pass.

(A.1.2.1.1) Boolean expressions containing operators **and**, **or**, and **not**.

The primary reason why the operators **and**, **or**, and **not** are distinguished from the rest (such as **xor**, **nor**) is that virtually every high-order programming language in which the Boolean type is recognized as a basic type implement these operators as primitive operations of that type. Thus, the tests in this category can be expected to be applicable to a wide range of tools.

Example 1:

```
with Support;
with Progress_Report;
procedure Foo (A, B, C : in Boolean) is
   package Progress renames Progress_Report;
begin
  if (A and B) then              -- decision 1
    Progress.Checkpoint (11);
  else
    Progress.Checkpoint (12);
  end if;
  if (B and C) then              -- decision 2
    Progress.Checkpoint (21);
  else
    Progress.Checkpoint (22);
  end if;
  if (C and A) then              -- decision 3
```

```
    Progress.Checkpoint (31);
  else
    Progress.Checkpoint (32);
  end if;
  if (A and B and C) then       -- decision 4
    Progress.Checkpoint (41);
  else
    Progress.Checkpoint (42);
  end if;
  if (A and not B) then         -- decision 5
    Progress.Checkpoint (51);
  else
    Progress.Checkpoint (52);
  end if;
  if (B and not C) then         -- decision 6
    Progress.Checkpoint (61);
  else
    Progress.Checkpoint (62);
  end if;
  if (C and not A) then         -- decision 7
    Progress.Checkpoint (71);
  else
    Progress.Checkpoint (72);
  end if;
  if not (not A and not B and not C) then  -- decision 8
    Progress.Checkpoint (81);
  else
    Progress.Checkpoint (82);
  end if;
end Foo;

with Foo;
with Support;
procedure Test_Foo_0 is
      -- achieves MCDC for none of the 8 decisions in Foo
      T : Boolean := Support.Identity (1) = 1;
      F : Boolean := Support.Identity (2) = 3;
begin
      Foo (T, T, F);
      Foo (T, F, T);
end Test_Foo_0;

-- Test_Foo_1, similar to Test_Foo_0, with this body:
      Foo (T, T, F);
      Foo (T, F, T);
      Foo (T, T, T);
-- achieves MCDC for decision 2

-- Test_Foo_2, similar to Test_Foo_0, with this body:
      Foo (T, T, F);
      Foo (T, F, T);
      Foo (F, T, T);
-- achieves MCDC for decisions 1, 2, 3
```

```
-- Test_Foo_3, similar to Test_Foo_0, with this body:
     Foo (T, T, F);
     Foo (T, F, T);
     Foo (F, T, T);
     Foo (T, T, T);
-- achieves MCDC for decisions 1, 2, 3 and 4

-- Test_Foo_4, similar to Test_Foo_0, with this body:
     Foo (T, T, F);
     Foo (T, F, T);
     Foo (F, T, T);
     Foo (F, F, F);
-- achieves MCDC for all decisions except 4, 8.

-- Test_Foo_5, similar to Test_Foo_0, with this body:
     Foo (T, T, F);
     Foo (T, F, T);
     Foo (F, T, T);
     Foo (F, F, F);
     Foo (T, T, T);
-- achieves MCDC for all decisions except 8.

-- Test_Foo_6, similar to Test_Foo_0, with this body:
     Foo (T, T, F);
     Foo (T, F, T);
     Foo (F, T, T);
     Foo (T, F, F);
     Foo (F, T, F);
     Foo (F, F, T);
     Foo (F, F, F);
     Foo (T, T, T);
-- achieves MCDC for all decisions.
```

This example only exercises the **and** operator (along with **not**); a similar set of tests could be constructed to exercise the **or** operator.

It should also be noted that a full suite of tests in this category should use a number of other variants of the basic Boolean terms, including calls to Boolean functions, Boolean array elements, relational expressions delivering Boolean results, and so on. Furthermore, the placement of the unary **not** operator should be varied to provide greater variations in the Boolean terms participating in the binary operations **and**, **or**.

(A.1.2.1.2) Boolean expressions containing other binary operators.

This category of tests will focus on binary Boolean operators other than **and** and **or**. These operators are:

xor - exclusive **or**, same as /=
= - equal to, opposite of **xor**
<= - less than or equal to, same as “implies”

>= - greater than or equal to, same as “implied by”
> - same as A **and not** B
< - same as **not** A **and** B
nor - same as **not** A **and not** B
nand - same as **not** A **or not** B

Of these, **nor** and **nand** are not available as primitive operators in Ada and the operator **xor** can also be written as Boolean equality.

In C, none of the above operators are available as Boolean operators, i.e., operators that treat their operands as Boolean terms. Writing an expression such as (A < B) will be interpreted as numeric comparison of A and B (assuming A and B are numeric terms). To force a Boolean interpretation of A, for example, one would have to rewrite it as (A != 0) or (A && 1), or (A || 0).

Test cases to exercise the available operators can be modeled after the test cases described in section (A.1.2.1.1).

(A.1.2.1.3) Boolean expressions containing short-circuit operators.

Boolean expressions involving certain binary operators could be evaluated in a manner that delivers the result without actually evaluating all the terms all the time. For example, if **A and B** is evaluated left to right, the result of False can be delivered without evaluating **B** when **A** has evaluated to False. This manner of evaluating a Boolean expression is known as short-circuit evaluation. The operators in these cases are commonly referred to as short-circuit operators and the expression as a short-circuited expression.

In some languages, notably C, the **and** (&&) and **or** (||) are always evaluated in short-circuit mode. In other languages, either there is clearly distinct syntax for specifying short-circuit operations or the decision to short circuit or not is left up to the implementation.

While **and** and **or** are the operators that are most commonly recognized as being eligible for short-circuit evaluation, there are other binary operators whose evaluation could be short circuited. For example, **A** < **B** will deliver False if **A** is True, regardless of the value of **B**, and hence could be short circuited. In fact, all relational operators except equality and inequality could be evaluated using the short-circuiting technique. Therefore, in the absence of knowledge of the internals of the specific compiler to be used, the test suite should include cases of these other short-circuitable operators as well.

A decision involving short-circuit operators is expected to be treated as multiple decisions. For example,

```
(A and then B) or else C
```

should be treated as if **A**, **B,** and **C** are separate decisions. This amounts to interpreting the short-circuited expression as a control flow construct equivalent to:

```
if A then
  if B then
    True
  else
    C
else
  C
```

as outlined in DO-248B.

To achieve MCDC for short-circuit expressions, the test cases should ensure each short-circuited term is evaluated at least twice, once for a True result and once for False. It should be noted, however, that it is possible to achieve MCDC with one set of test cases on a short-circuited expression, while the same set of test cases may not yield MCDC coverage on the equivalent conventional expression. In fact, Test_Foo_1 from the example in section (A.1.2.1.1) does not achieve MCDC on decision 6, whereas the same test driver will achieve MCDC on the short-circuited form of that decision. This seemingly anomalous behavior raises some interesting questions with respect to the languages where the implementation may decide to short circuit the evaluation or not. It also serves as a reminder that the tests of section (A.1.2.1.1) are not directly applicable to languages, such as C, where certain operators always evaluate in a short-circuited sequence.

Example 1:

The example from (A.1.2.1.1) can be adapted to short-circuit operators by replacing each **and** in Foo with an **and then**, although the coverage results from running the test drivers 0 through 6 should not be expected to match, as noted above. The following table compares the results of all seven tests run on the original example and the one that was adapted to short-circuit operators.

Test Driver	Decisions Achieving MCDC	
	Original Example From (A.1.2.1.1)	Example Adapted With Short-Circuit Operators
Test_Foo_0	None	None
Test_Foo_1	2	2, 6
Test_Foo_2	1, 2, 3	1, 2, 3, 5, 6, 7
Test_Foo_3	1, 2, 3, 4	1, 2, ,3, 4, 5, 6, 7
Test_Foo_4	1, 2, 3, 5, 6, 7	1, 2, 3, 5, 6, 7
Test_Foo_5	1, 2, ,3, 4, 5, 6, 7	1, 2, ,3, 4, 5, 6, 7
Test_Foo_6	All (1-8)	All (1-8)

Example 2:

This example illustrates the difference between MSM and the rest of the variants of MCDC as observable in Boolean expressions containing uncoupled terms only.

```
with Support;
with Progress_Report;
procedure Foo_MSM (A, B, C : in Boolean) is
   package Progress renames Progress_Report;
begin
  if (A and B) or C then
    Progress.Checkpoint (11);
  else
    Progress.Checkpoint (12);
  end if;
end Foo_MSM;
--
with Foo_MSM;
with Support;
procedure Test_Foo_MSM is
      T : Boolean := Support.Identity (1) = 1;
      F : Boolean := Support.Identity (2) = 3;
begin
      Foo (T, T, F);
      Foo (T, F, F);
      Foo (F, T, F);
      Foo (F, F, T);
end Test_Foo_MSM;
-- Achieves MCDC under MSM, but not under UCM, BBM, CCM, or
-- USM, as the independence of term C is not established by
-- toggling just C while holding all other inputs fixed.
```

(A.1.2.2) Boolean expressions containing coupled terms.

A Boolean expression is said to contain coupled terms if one or more terms depend on the value of or are the same as other terms. For example, the expression (**A and not B**) **or** (**not A and B**) contains two pairs of coupled terms: the terms **A** and **not A** are dependent on each other, as are the terms **B** and **not B**. Coupling can also be less direct. For example, the expression F(I) **and** G(I), where F and G are Boolean functions, may deliver results that are not independent of each other due to the fact that they share a common argument I. For example, if F(I) returns the Boolean result I < 0 and G(I) returns the result I < 10, both F(I) and H(I) will return True for any negative value of I. Furthermore, it is impossible to get F(I) to return True and G(I) to return False for the same value of I. This form of coupling has been referred to in the literature as weak coupling.

In real life applications, coupling can be even more subtle. For example, the seemingly uncoupled variables Weight_On_Wheels and Nose_Gear_Stowed may

indeed be coupled due to what they represent and the design constraints that govern them.

For MCDC analysis, the test suite focuses on directly or strongly coupled terms, where a Boolean term has multiple occurrences within the expression. Expressions containing more subtle forms of coupling could be treated as if the terms were uncoupled by stubbing the test environment and forcing such terms to take on desired values independently of each other. In that case, independent test cases are not warranted.

(A.1.2.2.1) Boolean expressions containing operators **and**, **or**, and **not**.

The objective of the tests in this category is to establish that the coverage analysis tool correctly reports coverage (or lack thereof) for Boolean expressions containing coupled conditions combined with the logical operators **and**, **or**, and **not**. Attention is focused first on the Affirmative tests and then on the Negative tests. All Affirmative tests will be rejected by UCM because UCM requires each condition to be varied while holding all other conditions fixed, an impossible feat in the presence of coupled terms. Of course, all Negative tests will also be rejected by UCM, though not for the correct reason, but because they contain coupled terms.

Example 1:

```
with Progress_Report;
procedure Xor2 (A, B : Boolean) is
-- Implements A xor B with AND, OR, and NOT.
      package Progress renames Progress_Report;
begin
      if (A and not B) or (not A and B) then
            Progress.Checkpoint (11);
      else
            Progress.Checkpoint (12);
      end if;
end Xor2;

with Support;
with Xor2;
procedure Test_Xor2_1 is
      T : Boolean := Support.Identity (1) = 1;
      F : Boolean := Support.Identity (2) = 3;
begin
      Xor2 (F, F);
      Xor2 (F, T);
      Xor2 (T, F);
      Xor2 (T, T);
end Test_Xor2_1;
```

The four calls to Xor2 represent exhaustive test cases in the two Boolean inputs. All variants of MCDC except UCM should accept this test set as satisfying MCDC.

```
-- Test_Xor2_2, similar to Test_Xor2_1, with this body:
      Xor2 (F, F);
      Xor2 (F, T);
      Xor2 (T, F);
```

This test set provides MCDC under BBM and CCM, but not under other variants. MSM, OCC, and RC/DC will report that the first **and** operator is not adequately covered, because {(F, T/F), (F, F/F), (T, T/T)} lacks the (T, F/F) test vector. OBC is applicable only for short-circuit operators, discussed in section (A.1.2.2.3).

Example 2:

```
with Progress_Report;
procedure Majority3 (A, B, C : Boolean) is
-- Implements 3-Input Majority voter with AND and OR.
      package Progress renames Progress_Report;
begin
      if (A and B) or (B and C) or (C and A) then
            Progress.Checkpoint (11);
      else
            Progress.Checkpoint (12);
      end if;
end Majority3;
with Support;
with Majority3;
procedure Test_Majority3_1 is
      T : Boolean := Support.Identity (1) = 1;
      F : Boolean := Support.Identity (2) = 3;
begin
      Majority3 (F, F, T);  -- call (1)
      Majority3 (F, T, T);  -- call (2)
      Majority3 (T, F, T);  -- call (3)
      Majority3 (T, F, F);  -- call (4)
end Test_Majority3_1;
```

The four calls to Majority3 provide MCDC under BBM and CCM, because each input has been shown to have an impact on the outcome while the other two inputs remain fixed. Calls 1 and 2 show the independence of **B**, calls 1 and 3 show the independence of **A**, and calls 3 and 4 show the independence of **C**. Under all other variants of MCDC, the four calls will be considered inadequate for MCDC. For example, MSM will find that the first **and** has not been adequately covered because there is never an outcome of True from it.

Adding two more calls will satisfy MSM, USM and OCC.

```
-- Test_Majority3_2, adds test cases (5) and (6)
      Majority3 (F, F, T);  -- call (1)
      Majority3 (F, T, T);  -- call (2)
      Majority3 (T, F, T);  -- call (3)
      Majority3 (T, F, F);  -- call (4)
      Majority3 (T, T, F);  -- call (5)
      Majority3 (F, T, F);  -- call (6)
```

Test cases (1, 4, 5) combine to show coverage of the first **and**, (1, 2, 6) combine to achieve coverage of the second **and**, and (1, 3, 4) combine to achieve coverage of the third **and**. Likewise, test cases (1, 2, 5) will establish coverage for the first **or**, and (2, 3, 6) for the second **or**.

To achieve RC/DC, a demonstration that toggling each input will not toggle the output for certain combination of values for the other inputs is also required. For example, given that **B**=True and **C**=True, toggling **A** should not toggle the output; likewise, given that **B**=False and **C**=False, toggling **A** should not affect the outcome. Thus, test cases {T, T, T/T; F, T, T/T; T, F, T/T; T, T, F/T; F, F, F/F; T, F, F/F; F, T, F/F; F, F, T/F} should be added to a test set that achieves MSM coverage to achieve RC/DC. Adding these test cases makes the test set exhaustive in this case.

(A.1.2.2.2) Boolean expressions containing other binary operators.

To generalize MSM to apply to operators other than **and** and **or**, it is important to identify the masking properties, if any, of the **and** and **or** operators. For example, the **and** operator has the masking property that if one of its inputs is False, it masks the effect of the other input, i.e., the output remains False regardless of the second input. Thus, a False input is said to mask the effect of the other input. MSM demands that at each operator, the operand values be such that the effect of toggling an input value is never masked by the values of the operands all the way up the expression tree.

There are 16 binary operators, labeled as F_0 through F_f, in the following table:

A	B	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_a	F_b	F_c	F_d	F_e	F_f
F	F	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T
F	T	F	F	F	F	T	T	T	T	F	F	F	F	T	T	T	T
T	F	F	F	T	T	F	F	T	T	F	F	T	T	F	F	T	T
T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T
Common Operator Symbols			AND	>		<		XOR	OR	NOR	=		>=		<=	NAND	

Of these, F_0 (always False), F_f (always True), F_3 (always **A**), F_5 (always **B**), F_a (always **not B**), and F_c (always **not A**) are of little interest because they represent pathological cases where one or both operands are ineffective and, hence will

never achieve MCDC under any of the variants. That leaves eight binary operators to be dealt with in this section: the six relational operators =, /=, >, >=, <, and <=, and **nor** and **nand**. The masking properties of these operators are tabulated below, along with those of **and** and **or** for reference.

Operator Name	**A** masks the effect of **B** if **A** is	**B** masks the effect of **A** if **B** is
and	False	False
or	True	True
nand	False	False
nor	True	True
=	(none)	(none)
/=	(none)	(none)
<	True	False
<=	False	True
>	False	True
>=	True	False

Now consider an expression such as:

(**A and B**) /= (**B nor C**)

An independence pair for **A** could be (T, T, F/T; F, T, F/F). An independence pair for **B** could be (T, F, F/T; T, T, T/F). Both **B** and **C** toggle at once to form the independence pair for MSM in this case.

(A.1.2.2.3) Boolean expressions containing short-circuit operators.

As noted in section A.1.2.2.2, eight Boolean binary operators exhibit the masking property, namely, for a certain value of one of the operands, the other operand has no affect on the outcome of the expression. As one might expect, the same set of operators is also the set that can be short circuited. In the case of **and**, **or**, **nand**, and **nor**, the masking property has a certain symmetry, arising from the commutative property of those operators: the short-circuit evaluation can begin with either the left or the right operand and the same evaluation rule will apply. For example, in **A and B**, either **A** or **B** can be evaluated first. If either **A** or **B** is False, the result will be False. If either is True, the other operand must be examined to determined the outcome. In programming languages, the order is usually specified as left to right, giving the programmer the liberty to assume that the second operand is always evaluated under the protection of a guaranteed value of the first.

In the case of the remaining four short-circuitable operators, the symmetry is absent. For example, consider the (**A** < **B**). Evaluation of this expression could be short circuited in two ways:

```
if A then
  False;
else
  B;
end if;
or, alternatively,
if not B then
  False;
else
  not A;
end if;
```

In order for the short-circuit evaluation to make a meaningful difference in the semantics of a programming language, the language should specify which order is applicable.

In C, **&&** and **||** are the only short-circuitable Boolean operators, because **nand** and **nor** are not available as primitive operators, and the relational operators do not interpret their operands as Boolean. In Ada, **and then** and **or else** are the only short-circuitable Boolean operators; all others must evaluate both operands before delivering a result[2]. Like C, Ada does not provide primitive operators for **nand** and **nor**. As a consequence, the test suite only considers two familiar short-circuit operators: **&&** and **||** in C, **and then** and **or else** in Ada. In what follows, the Ada notation is used, although the discussions apply to the C counterparts as well.

DO-248B clarifies how short-circuit operators are to be dealt with in the context of MCDC.

- In each case, the evaluation is to be represented by a control flow structure:

A AND THEN B	**A OR ELSE B**

if A then
B;
else
False;
end if;
	if A then
	True;
	else
	B;
	end if;

- MCDC must be achieved for each decision represented in the control flow structure.

[2] Any optimization that avoids evaluating either operand is permitted only if the effect can be guaranteed to be the same as evaluating both operands.

Thus, attaining MCDC for **A and then B** or **A or else B** entails achieving MCDC for **A** and **B** individually. However, because **B** is evaluated only when the value of **A** does not mask it, it would be necessary to hold the value of **A** at the nonmasking value to achieve MCDC for **B**. Thus, for **A and then B**, the test cases (T, T/T; T, F/F; F, -/F) will provide MCDC.

It should be noted that, in the above discussion, operands **A** and **B** could themselves be complex decisions; if so, the usual MCDC criterion will apply to each of those decisions recursively.

Example 1: Consider two different implementations of **A and B** using short-circuit constructs:

```
with Progress_Report;
procedure A_AND_THEN_B (A, B : Boolean) is
      -- Logical conjunctions using then AND THEN construct
      package Progress renames Progress_Report;
begin
      if (A and then B)                    -- decision 1
      then
            Progress.Checkpoint (11);
      else
            Progress.Checkpoint (12);
      end if;
end A_AND_THEN_B;

with Progress_Report;
procedure B_AND_THEN_A (A, B : Boolean) is
      -- Logical conjunctions using then AND THEN construct
      package Progress renames Progress_Report;
begin
      if (B and then A)                    -- decision 2
      then
            Progress.Checkpoint (21);
      else
            Progress.Checkpoint (22);
      end if;
end B_AND_THEN_A;
```

Now consider two tests, one each to test A_AND_THEN_B and B_AND_THEN_A.

```
with Support;
with A_AND_THEN_B;
procedure Test_1 is
      T : Boolean := Support.Identity (1) = 1;
      F : Boolean := Support.Identity (2) = 3;
begin
      A_AND_THEN_B (T, T);
      A_AND_THEN_B (T, F);
      A_AND_THEN_B (F, F);
```

```
end Test_1;
------------------------------------------------------------
with Support;
with B_AND_THEN_A;
procedure Test_2 is
      T : Boolean := Support.Identity (1) = 1;
      F : Boolean := Support.Identity (2) = 3;
begin
      B_AND_THEN_A (T, T);
      B_AND_THEN_A (T, F);
      B_AND_THEN_A (F, F);
end Test_2;
```

Test_1 will achieve MCDC for decision 1 for all eight variants of MCDC (because each of the top-level decisions in the flow graph model is a simple decision and has been varied to take on both True and False values). However, Test_1 would not achieve MCDC under any variant had the decision been implemented using the traditional **and** operator because the test vector {T, T/T; T, F/F; F, F/F} does not provide MCDC coverage for **and**. Furthermore, Test_2 fails to achieve MCDC for decision 2 under any of the variants. This is so because, even though the arguments passed against **A** and **B** take on the values of True and False, the two cases where **B** is set to False do not evaluate **A** in the body of B_AND_THEN_A.

There are two important observations to make from the above example:

- Whether or not an operator is implemented as a short-circuit operator impacts the test cases needed to achieve MCDC.

- The order of evaluation of a short-circuit operator also impacts the test cases needed to achieve MCDC.

Example 2:

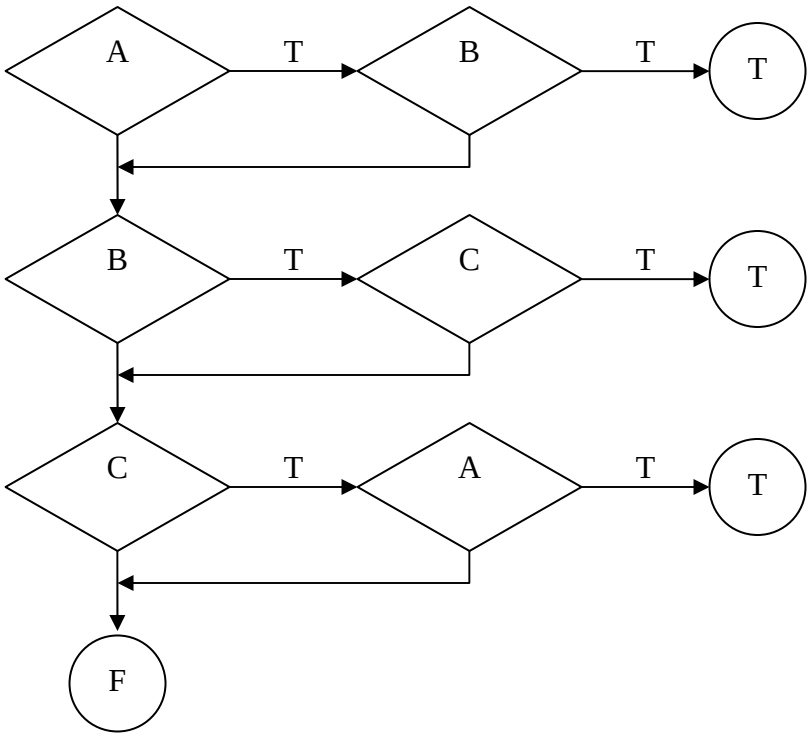
```
with Progress_Report;
procedure Majority3_SC (A, B, C : Boolean) is
-- Implements 3-Input Majority voter with AND THEN and
-- OR ELSE.
      package Progress renames Progress_Report;
begin
      if (A and then B)
            or else (B and then C)
            or else (C and then A)
      then
            Progress.Checkpoint (11);
      else
            Progress.Checkpoint (12);
      end if;
end Majority3_SC;
```

```
with Support;
with Majority3_SC;
procedure Test_Majority3_SC_1 is
      T : Boolean := Support.Identity (1) = 1;
      F : Boolean := Support.Identity (2) = 3;
begin
      Majority3_SC (F, F, T);  -- call (1)
      Majority3_SC (F, T, T);  -- call (2)
      Majority3_SC (T, F, T);  -- call (3)
      Majority3_SC (T, F, F);  -- call (4)
end Test_Majority3_SC_1;
```

This test driver is equivalent to Test_Majority3_1 but does not achieve MCDC under any of the variants of MCDC because the first occurrence of **B** never evaluates to True. More interestingly, all variants of MCDC (even UCM) can achieve MCDC for Majority3_SC with the following test cases.

```
Majority3_SC (T, T, F);  -- call (1)
Majority3_SC (F, T, T);  -- call (2)
Majority3_SC (T, F, T);  -- call (3)
Majority3_SC (T, F, F);  -- call (4)
Majority3_SC (F, T, F);  -- call (5)
```
Majority3_SC (F, F, T); -- call (6)

To see how these test cases achieve MCDC for all variants, one only needs to examine the control flow model of the expression (**A and then B**) **or else** (**B and then C**) **or else** (**C and then A**).



First, observe that the flow graph model has no decisions containing coupled terms; hence, coverage under UCM is not ruled out.

Second, observe that covering every branch will ensure that each decision in the flow graph model evaluates to True and False, which is a sufficient condition (in this case) for achieving MCDC for those decisions under all variants.

Third, observe that the first three test cases cover all branches labeled T. In addition, several of the False branches are also covered by these test cases, leaving only three False branches uncovered: (1) the False branch of first occurrence of **C**, (2) the False branch of the second occurrence of **A**, and (3) the False branch of the second occurrence of **C**.

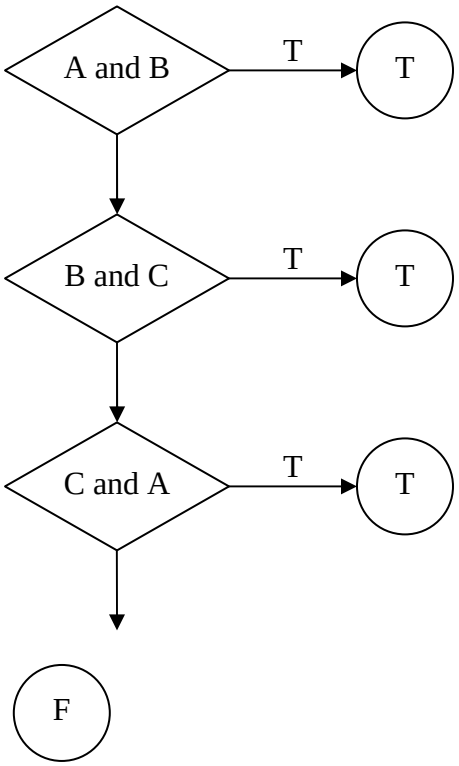
Finally, observe that calls (5), (6), and (4) cover the three uncovered False branches in order.

It is also easy to see that nothing less than the six test cases shown above will achieve MCDC under any of the variants. This is because the use of short-circuit operators has reduced the coverage criterion to one of branch coverage and six test cases at a minimum are needed to achieve branch coverage on the equivalent flow graph.

Example 3: This example uses a mixture of short-circuit and traditional operators in a single decision. As such, the test is not applicable to C tools.

```
with Progress_Report;
procedure Majority3_Mix (A, B, C : Boolean) is
-- Implements 3-Input Majority voter with the non-short-
-- circuit operator AND and OR ELSE.
      package Progress renames Progress_Report;
begin
      if (A and B)
            or else (B and C)
            or else (C and A)
      then
            Progress.Checkpoint (11);
      else
            Progress.Checkpoint (12);
      end if;
end Majority3_Mix;
```

The flow graph model of this implementation differs from the previous one in a subtle way.



This procedure contains three top level decisions, namely (**A and B**), (**B and C**), and (**C and A**), none of which contain coupled terms. That makes this implementation potentially acceptable to all variants of MCDC including UCM. The following test driver will achieve MCDC coverage for all variants.

```
with Support;
with Majority3_Mix;
procedure Test_Majority3_Mix_1 is
      T : Boolean := Support.Identity (1) = 1;
      F : Boolean := Support.Identity (2) = 3;
begin
      Majority3_Mix (T, T, F);  -- call (1)
      Majority3_Mix (F, T, T);  -- call (2)
      Majority3_Mix (T, F, T);  -- call (3)
      Majority3_Mix (T, F, F);  -- call (4)
      Majority3_Mix (F, T, F);  -- call (5)
      Majority3_Mix (F, F, T);  -- call (6)
end Test_Majority3_Mix_1;
```

Three observations are needed to see that Test_Majority3_Mix_1 achieves MCDC under all variants.

- First, note that each of the top-level decision involves two uncoupled terms joined by an **and**. The criterion for achieving MCDC for such a decision is the same for all variants, namely, the test vector {T, T/T; F, T/F; T, F/F}.

- Second, note that the first three calls constitute the test case (T, T/T) for each of the three top-level decisions.

- Third, calls (4) and (5) complete the required test vector for the decision (**A and B**); calls (5) and (6) complete the set for (**B and C**); and the calls (4) and (6) complete the set for decision (**C and A**).

As a final example, consider a variant of Majority3_Mix in which the **and**s are short circuited but the **or**s are not.

```
with Progress_Report;
procedure Majority3_Mix2 (A, B, C : Boolean) is
-- Implements 3-Input Majority voter with AND THEN and
-- non-short-circuit OR.
      package Progress renames Progress_Report;
begin
      if (A and then B)
            or (B and then C)
            or (C and then A)
      then
            Progress.Checkpoint (11);
      else
            Progress.Checkpoint (12);
      end if;
end Majority3_Mix2;
```

If the three short-circuit subexpressions are designated as **X** = (**A and then B**), **Y** = (**B and then C**), and **Z** = (**C and then A**), then four test cases are needed to

achieve MCDC for the top-level decision in the **if** statement: (F, F, F/F; T, F, F/T; F, F/T; F, F, T/T). In addition, each of the subexpressions must achieve MCDC by achieving branch coverage within its respective flow graph model and by achieving MCDC for the two decisions involved in the flow graph model.

To achieve MCDC for the subexpression (**A and then B**), three test cases are needed:

A	**B**	**C**
T	T	-
F	T	-
T	F	-

Similarly, to achieve MCDC for the subexpressions (**B and then C**) and (**C and then A**), three test cases are needed for each:

A	**B**	**C**
-	T	T
-	F	T
-	T	F
A	**B**	**C**
T	-	T
T	-	F
F	-	T

The union of these nine test cases leads to a minimum set of three test cases:

A	**B**	**C**
F	T	T
T	F	T
T	T	F

These test cases also provide three of the four test cases needed for coverage of (**X or Y or Z**). The addition of the test case (**A**=F, **B**=F, **C**=F) will complete the test set.

A	**B**	**C**
F	T	T
T	F	T
T	T	F
F	F	F

This is not sufficient to show the independence of any condition according to reference B-2 or the Chilenski study using Boolean Differences to define independence [B-3]. In the following three pair analyses, two significant

conditions change (bold columns) in all three instances. Independence requires that only one single significant condition change.

A	B	C	?	A	B	**B**	C	**C**	A
F	F	F	F	F	x	**F**	x	**F**	x
F	T	T	T	F	x	**T**	T	**T**	F

A	B	C	?	**A**	B	B	C	**C**	A
F	F	F	F	**F**	x	F	x	**F**	x
T	F	T	T	**T**	T	F	x	**T**	T

A	B	C	?	**A**	B	**B**	C	C	A
F	F	F	F	**F**	x	**F**	x	F	x
T	T	F	T	**T**	F	**T**	F	F	x

The same number of tests are needed for MCDC under Chilenski/Miller or Chilenski/BD, independent of the mix of short circuit and normal form operators.

It is interesting to compare this smaller set of test cases that achieves MCDC on the Majority3_Mix2 implementation with the set necessary to achieve MCDC on the Majority3_SC implementation. The smaller set is not sufficient to achieve MCDC on Majority3_SC because all of the branches are not covered due to the short-circuit evaluation occurring at the top level (**or** level).

(A.2) Boolean expressions in nonbranching contexts.

The objective of the tests in this category is to establish the correctness of MCDC coverage of Boolean expressions that appear in nonbranching contexts, including the right-hand side of an assignment, subscript of an array, parameter to a subprogram and so on. A comprehensive list of such contexts can be found under the Level B test requirements.

In a full test suite, an elaborate set of MCDC tests for the nonbranching contexts is not necessary because the Level B tests already include tests for coverage of decisions in nonbranching contexts. Reformulating a small cross section of the tests from (A.1) from a branching context to a nonbranching context should be sufficient.

B.6 DISCRIMINATING TESTS FOR LEVEL A.

As discussed in appendix A, there are a number of alternate approaches to MCDC. Distinguishing between the five variants of MCDC (UCM, CCM, BBM, MSM, and USM) and the three alternate forms of structural coverage for Boolean expressions (OCC, OBC, and RC/DC) is a goal for the test objectives for Level A, because such variants are prevalent in practice. Being able to discriminate between them will help evaluate the relative merits of each.

The unique cause and masking approaches to MCDC are the two forms of MCDC that are most often discussed. As such, the test suite should consider test cases that compare these two approaches with the other alternatives.

B.6.1 THE UCM VERSUS OTHER COVERAGE MEASURES.

Because UCM will reject any decision containing replicated terms, it is easy to find a test case that will distinguish it from the rest of the variants. For example, the exclusive **or** function written in terms of **and**, **or** and **not** is such a test case.

(**A and not B**) **or** (**not A and B**)

UCM fails because it is not possible to demonstrate the independence of any of the conditions by varying just one condition and holding all others fixed.

However, CCM and BBM view this expression as a function in two (uncoupled) conditions **A** and **B**, and the independence of both is established by the test set: {(F, F/F), (T, F/T), (F, T/T)}.

MSM can be satisfied by the test set: {(F, F/F), (T, F/T), (F, T/T), (T, T/F)}. Four test cases are required because each occurrence of **A** and **B** needs its own independence pair. Each test case in the pair must hold the other half of the disjunction (**or**) at False, so that the effect of changing the condition is not masked by a True output from the other half of the disjunction.

(F, F/F), (T, F/T) => independence of first occurrence of **A**
(T, T/F), (T, F/T) => independence of first occurrence of **B**
(T, T/F), (F, T/T) => independence of second occurrence of **A**
(F, F/F), (F, T/T) => independence of second occurrence of **B**

This test set also satisfies USM because only one uncoupled condition changes value in each independence pair.

B.6.2 THE MSM VERSUS OTHER COVERAGE MEASURES.

While it is relatively simple to create a test case to distinguish UCM from the other alternate forms of MCDC, more effort is required to distinguish MSM from the other alternatives.

Comparing Masking with Masking + Unique Cause
Comparing Masking with Black-Box and Coupled Cause MCDC
Comparing Masking with Ooperator Coverage
Comparing Masking with Object-Code Branch Coverage
Comparing Masking with Reinforced Condition/Decision Coverage

B.6.2.1 Comparison with USM.

USM, which is masking with unique case MCDC, can be distinguished from masking alone. Consider the following decision:

(**A and B**) **or** (**C and D**)

When demonstrating the independence of **A**, the term (**C and D**) needs to remain False, which can be accomplished with any of the following three pairs of values: (**C**=F, **D**=F), (**C**=T, **D**=F), or (**C**=T, **D**=F). MSM does not dictate which of these is selected. Thus, the test set {(T, T, F, T/T), (T, F, F, T/F), (F, T, T, F/F), (F, T, T, T/T)} will achieve MSM, but it does not achieve USM because there is no pair that establishes the independence of condition **B**, where all other uncoupled condition, are held fixed. It is not possible to construct an expression that is acceptable to USM but not MSM, because USM is MSM with additional constraints.

B.6.2.2 Comparison With CCM/BBM.

CCM and BBM are essentially the same coverage criteria. The primary distinction is that BBM is applicable to nonBoolean inputs, while CCM is defined in terms of Boolean inputs. For this test suite, CCM and BBM are the considered the same. To distinguish CCM from MSM, a test case is needed where an input condition occurs redundantly, such as

(**A and A**)

Because CCM allows all coupled terms to change at once, both occurrences of **A** may be changed simultaneously from (T, T/T) to (F, F/F) to demonstrate independence of either occurrence of **A**. Under MSM, these two tests cannot pair up to demonstrate the independence of either occurrence of **A** because the change in the other occurrence will make the output change.

The expression (**A and B**) **or** (**C and D**) that was used to discriminate between MSM and USM can be used to discriminate between CCM and MSM.

Because USM is an extension of MSM, the expression (**A and A**) and the test set {(T, T/T), (F, F/F)} can be used to discriminate between CCM from USM. However, no expression can be constructed to achieve coverage under USM that fails to achieve coverage under CCM. In USM, every independence pair has the property that all but one of the uncoupled conditions remains fixed. Such a pair is always an independence pair under CCM as well. Therefore, the coverage set for USM is a coverage set for CCM.

B.6.2.3 Comparison With OCC.

OCC addresses coverage for each operator individually, whether or not the effect of the test cases that achieve the coverage can be measured at the output. This is a weaker condition than MSM, which requires that any change in the output of an operator be observable at the output of the decision. Therefore, it is possible to construct an expression that can meet OCC but not meet MSM. For example, consider

(**A and B**) **or not A**

The test set {(T, F/F), (F, T/T), (T, T/T)} meets the coverage requirements for each of the two binary operators in the decision. The **and** requires the three cases {(T, T/T), (T, F/F), (F, T/F)} and the **or** operator needs the set {(F, F/F), (T, F/T), (F, T/T)}.
No test set exists that will satisfy MSM for the above expression, because an independence pair for the first occurrence of **A** will need to hold the second operand of **or** at False while toggling the value of the first occurrence of **A**, which is impossible. Also, it is not possible to construct an expression that could achieve MSM but not OCC, because achieving coverage at each operator is a necessary (but not sufficient) condition to achieve MSM coverage.

B.6.2.4 Comparison With OBC.

OBC measures branching within the object code of the program, and by definition, bypasses expressions that are evaluated without branching. This means that a Boolean expression containing no short-circuit operators will meet OBC no matter what the terms are. Therefore, it is simple to construct test cases to distinguish OBC from any variant of MCDC. Consider the C assignment statement:

A = **B**;

where **A** and **B** are uncoupled Boolean variables. Under each of the five variants of MCDC in the variants group, two test cases are needed to satisfy the coverage criterion. Under OBC, a single test case will suffice.

It is interesting to try to find a more complex expression that will satisfy OBC without satisfying MSM, especially in a language such as C in which every logical operator is a short-circuit operator. OBC treats each short-circuited term as a separate decision. If the original expression occurs in a branching context, each decision will have to achieve decision coverage before OBC will be satisfied for the whole expression, which is the requirements for achieving MCDC as per DO-248B. If the original expression occurs in a nonbranching context, whether or not OBC achieves coverage is dependent on the generated object code. Consider, for example:

A = **B** && **C**;

If the semantic model used by the compiler is

```
if (B) then
  if (C) then
    A = 1;
  else
    A = 0;
else
  A = 0;
```

and generates code to faithfully duplicate this flow model in the object code, then the object code will contain sufficient number of branches to make OBC equivalent to MSM. If the compiler optimizes this into a nonbranching sequence of code, OBC will be achieved with fewer test cases and, therefore, will fall short with respect to any variant of MCDC.

Constructing an expression that can satisfy MSM (or any other variant) but not OBC is not possible. The reasoning is as follows: Let **e** be such an expression and τ be a test set that satisfies MSM. If **e** contains short-circuit operators, each of the operands of those short-circuit operators is a decision by itself and must be covered (take on True and False values) in τ. The same test cases will provide coverage for that decision under OBC. Because every such decision in **e** must be covered in τ, τ must satisfy OBC as well.

It is also worth noting that OBC goes beyond the usual source-based structural coverage criteria such as statement coverage, decision coverage, and MCDC. For example, consider the following Ada source statement for integer operands A, B and C.

```
A := (B mod C);
```

The object code will likely contain branching constructs because a hardware supported modulus function rarely matches Adas definition of the **mod** operator, especially for negative operands. OBC will ascertain that the source code has been exercised with a sufficient number of tests cases to achieve full coverage of the object code.

B.6.2.5 Comparison With RC/DC.

RC/DC is an extension of MCDC. In addition to demonstrating that each condition can independently affect the outcome of the parent decision, RC/DC requires that each condition is demonstrated to independently keep the outcome of the parent decision. A condition is shown to independently keep the outcome by toggling the value of the condition while holding fixed all other conditions and observing that the outcome remains unchanged.

Unlike the requirement to independently affect the outcome, which can be expected to be met for all reasonable expressions, the requirement to independently keep the outcome cannot be met for all reasonable expressions. Consider the simple case where the decision is a single condition. The requirement that the condition keep the outcome for some fixed values of the other conditions, of which there are none, cannot be met. Another not so trivial example is involves the **xor** operator. There are no test vector pairs that can show that condition **A** can independently keep the outcome of **A xor B** for any fixed value of **B**. Therefore, RC/DC has significance only for a subset of expressions for which MCDC can be achieved. For example, consider the expression:

A and B

The minimal test set that satisfies MCDC (all variants) is: {(T, T/T), (T, F/F), (F, T/F)}. To satisfy RC/DC, the test case (F, F/F) must be added to show that both **A** and **B** can independently keep the outcome.

Because RC/DC levies additional requirements on the test set beyond what MCDC levies, it is impossible to construct an expression that satisfies RC/DC without also achieving MCDC.

B.7 REFERENCES.

B-1. The Boeing Company, "Study of Qualification Criteria for Software Verification Tools, Phase 2 Position Paper," a contract deliverable to NASA under contract NAS1-00106, Task Order 1008, September 25, 2003.

B-2. Chilenski, John Joseph and Miller, Steven P., "Applicability of Modified Condition Decision Coverage to Software Testing," *Software Engineering Journal*, Vol. 7, No. 5, September 1994, pp. 193-200.

B-3. Chilenski, J.J., "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion," FAA Report DOT/FAA/AR-01/18, April 2001.