DOT/FAA/AR-07/17

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

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

June 2007

Final Report

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

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

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

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT/FAA/AR-07/17		

4. Title and Subtitle	5. Report Date
OBJECT-ORIENTED TECHNOLOGY VERIFICATION PHASE 3 HANDBOOK—STRUCTURAL COVERAGE AT THE SOURCE-CODE AND OBJECT-CODE LEVELS	June 2007
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
John Joseph Chilenski and John L. Kurtz	

9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
The Boeing Company P.O. Box 3707 Seattle, WA 98124-2207	
	11. Contract or Grant No.

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 R&D Division COTR was Charles Kilgore.

16. Abstract

The purpose of this Handbook is to provide guidelines into issues and acceptance criteria for the use of structural coverage analysis (SCA) at the source-code (SC) versus object-code (OC) or executable object-code (EOC) levels when using object-oriented technology (OOT) in commercial aviation to satisfy Objectives 5 through 8 of Table A-7 in RTCA DO-178B/EUROCAE ED-12B. OOT has been used extensively throughout the non-safety-critical software and computer-based systems industry. OOT has also been used in safety-critical medical and automotive systems and is now being used in the commercial airborne software and systems domain. However, as with any new technology, there are concerns and issues relating to its adoption within safety-critical systems.

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

The differences between SC and OC/EOC coverage analyses for the OOT features and MCDC are identified. An approach for dealing with the differences is provided for each issue identified.

17. Key Words	18. Distribution Statement
Object-oriented technology, Structural coverage, Decision coverage, Statement coverage, Verification, Coupling-based integration testing, Analysis	This document is available to the U.S. public through the National Technical Information Service (NTIS) Springfield, Virginia 22161.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	24	

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

CAST	Certification Authorities Software Team
EOC	Executable object-code
EOCC	Executable object-code coverage
FAA	Federal Aviation Administration
LHS	Left-hand side
MCDC	Modified condition decision coverage
OBC	Object-code branch coverage
OC	Object code
OCC	Object-code coverage
OOT	Object-oriented technology
OOTiA	Object-Oriented Technology in Aviation
RHS	Right-hand side
SC	Source code
SCA	Structural coverage analysis
SCC	Source-code coverage

EXECUTIVE SUMMARY

Object-oriented technology (OOT) has been used extensively throughout the non-safety-critical software and computer-based systems industry, in safety-critical medical and automotive systems and is now being used in the commercial airborne software and systems domain. However, as with any new technology, there are concerns and issues relating to its adoption within safety-critical airborne systems. Previous Federal Aviation Administration (FAA) research and two OOT in Aviation (OOTiA) workshops with industry indicate that there are some areas of OOT verification that are still a concern in safety-critical systems. One of those areas of concern is the adequacy of performing structural coverage analysis (SCA) at either the source-code (SC) level or object-code (OC)/executable object-code (EOC) level.

This Handbook provides input to industry and the FAA into issues and acceptance criteria for the use of SCA at the SC versus OC/EOC levels within OOT in commercial aviation, as required by Objectives 5 through 8 of Table A-7 in RTCA DO-178B/EUROCAE ED-12B. The intent of the SCA is to provide an objective assessment (measure) of the completeness of the requirements-based tests and support the demonstration of the absence of unintended function.

Certain features of OOT requiring object-code coverage (OCC) or executable object-code coverage (EOCC) analysis are identified, as well as features requiring source-code coverage (SCC) analysis. The combination of these features indicates that either a combined SCC and OCC/EOCC analysis or source to OC/EOC traceability is needed for OOT software to satisfy Objectives 5 through 8 of Table A-7 in DO-178B/EUROCAE ED-12B for the following OOT features:

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

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

1. INTRODUCTION.

1.1 PURPOSE.

This Handbook is intended to provide guidelines to industry and the Federal Aviation Administration (FAA) into issues and acceptance criteria for the use of structural coverage analysis (SCA) at the source-code (SC) level versus object-code (OC) or executable object-code (EOC) level when using object-oriented technology (OOT) in commercial aviation, as required by Objectives 5 through 8 of Table A-7 in RTCA DO-178B/EUROCAE ED-12B (DO-178B hereinafter) [1]. The intent of the SCA is to provide an objective assessment (measure) of the completeness of the requirements-based tests and supports the demonstration of the absence of unintended
function [1].

This Handbook identifies that either a combined source-code coverage (SCC) analysis and object-code coverage (OCC) and executable object-code coverage (EOCC) analysis, or SCC analysis and SC to OC/EOC traceability is needed for OOT software to satisfy Objectives 5 through 8 of Table A-7 in DO-178B for the following OOT features:

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

This Handbook also identifies that object-code branch coverage (OBC) of short-circuited logic at the OC or EOC level is not equivalent to modified condition decision coverage (MCDC) of logic at the SC level in the general case. To cover the primary difference between MCDC and OBC requires that the independence of each condition be demonstrated. Note that the results concerning OBC of short-circuited logic apply equally to both OOT software and non-OOT software.

1.2 BACKGROUND.

DO-178B specifies the need for SCA in Objectives 5 through 8 of Table A-7 in reference 1.

OOT has been used extensively throughout the non-safety-critical software and computer-based systems industry, in safety-critical medical and automotive systems, and is now being used in the commercial airborne software and systems domain [2 and 3]. Previous FAA research [2, 3, and 4] and two Object-Oriented Technology in Aviation (OOTiA) workshops with industry (see http://shemesh.larc.nasa.gov/foot/ and reference 5 for more information) indicate that guidance for the application of SCA to OOTiA is needed.

The FAA requested that The Boeing Company conduct research to identify issues and provide input to the industry and the FAA on SCA at the SC versus OC and EOC level (satisfaction of Objectives 5 through 8 of DO-178B/ED-12B Table A-7 [1]) within OOTiA. This Handbook is a companion document to the research report [6] on structural coverage at the source- and object-code levels.

1.3 DOCUMENT OVERVIEW.

As stated, this Handbook is a companion document to the research report [6]. The research report contains the details behind the steps employed in this Handbook. This Handbook contains the practical how-to guidelines for performing SCA for certain aspects of OOT.

- Section 1 provides the purpose, background, and general overview of this Handbook.
- Section 2 identifies OOT features requiring additional consideration.
- Section 3 identifies issues concerning the substitution of OBC for MCDC needing additional consideration.
- Section 4 summarizes the approach of this Handbook.
- Section 5 provides a list of references used in this Handbook.
- Section 6 identifies activities and documents related to the work reported herein.

1.4 RELATED ACTIVITIES AND DOCUMENTS.

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

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

2. OBJECT-ORIENTED TECHNOLOGY FEATURES REQUIRING ADDITIONAL CONSIDERATION.

From the high-level perspective, coverage analysis at the SC level, OC level, and EOC level are relatively equal as each has strengths and weaknesses, independent of whether OOT is used or not [6]. To understand this, consider the basic need for coverage analysis depicted in figure 1.

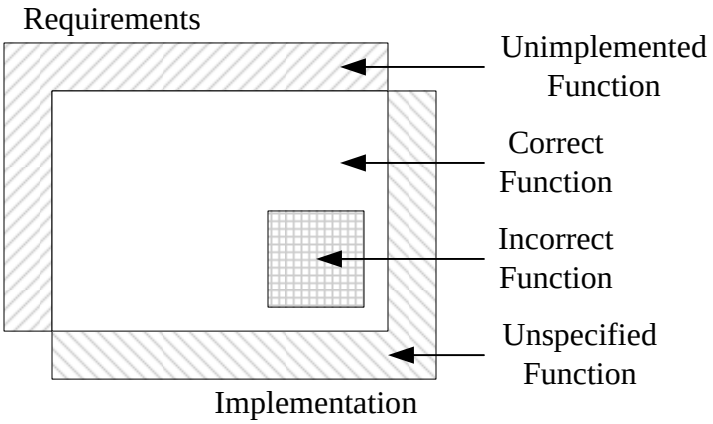

Figure 1. Requirements/Implementation Overlap

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

Consider how the intermediate life cycle artifacts between the requirements and the implementation fit into an analysis of overlaps as in figure 1. For this analysis, the simple five-level software-process life cycle model and the following corresponding artifacts are derived from DO-178B [1]:

- Requirements (high-level requirements)
- Design (low-level requirements and architecture)
- SC
- OC
- EOC (i.e., implementation)

In the software artifacts, requirements consist of both traceable requirements and derived requirements. The analysis of the overlap of the five-level life cycle artifacts is depicted in figure 2.

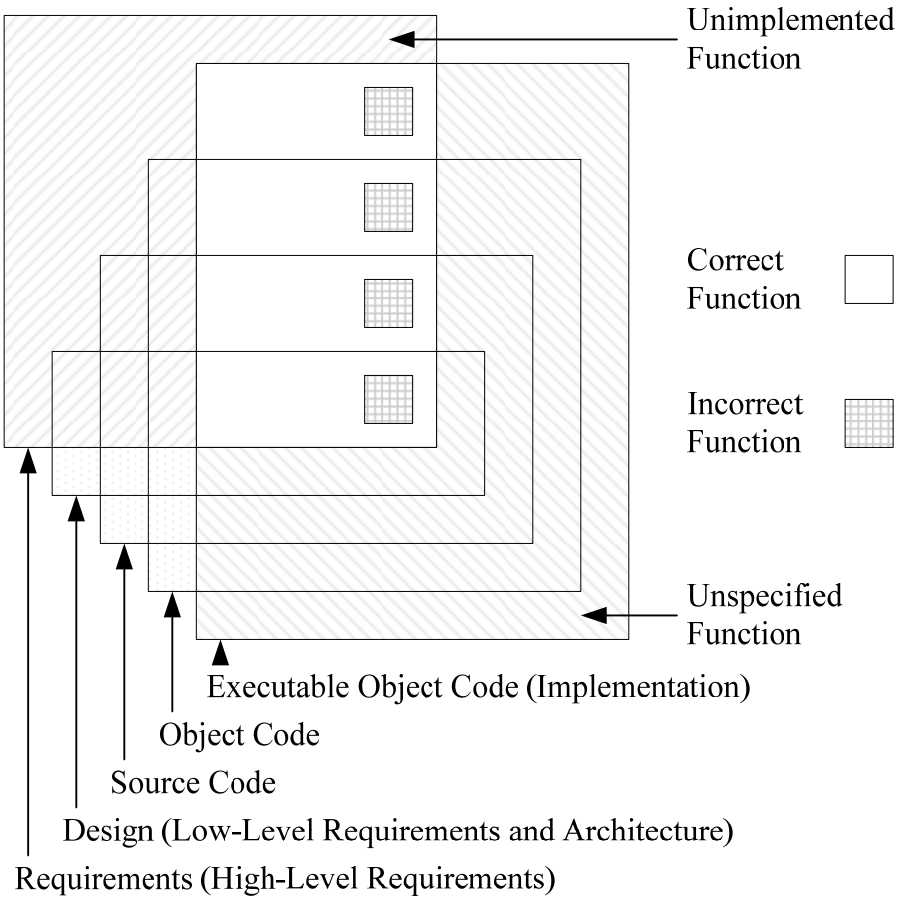

Figure 2. Five-Level Life Cycle Artifacts Overlap

Note that in figure 2 there are more subdomains than in figure 1. The four major subdomains from figure 1 are still present in figure 2 (unimplemented function, correct function, incorrect function, unspecified function), but the addition of the design artifacts, SC and OC has divided these subdomains further and added some new ones. Note that figure 2 is using a simple rectilinear Venn diagram representation, so all overlapping subdomains cannot be represented.

If one looks at injecting faults into the subdomains of figure 2, then the SCC analysis, OCC analysis, and EOCC analysis will each find faults in the same number of mutually exclusive subdomains. Since these subdomains are mutually exclusive, this means that each has its own strengths and weaknesses [6]. Note also that for the majority of subdomains, neither SCC analysis, OCC analysis, nor EOCC analysis is guaranteed to find the faults [6].

The previous analysis is independent of whether OOT is used or not. Therefore, if there are OOT-related structural coverage issues, they must exist with the specific features of OOT and the implementation of those features within specific programming languages. The specific features of OOT for the SCC, OCC, or EOCC issues are discussed in the following sections.

- Methods tables
- Constructors
- Initializers
- Finally blocks

2.1 METHODS TABLES.

Methods tables (also known as dispatch tables, virtual method tables, and vtables) are mechanisms used to support dispatching within OOT [7]. Figure 3 depicts the implementation for methods tables found in a previous study [7].

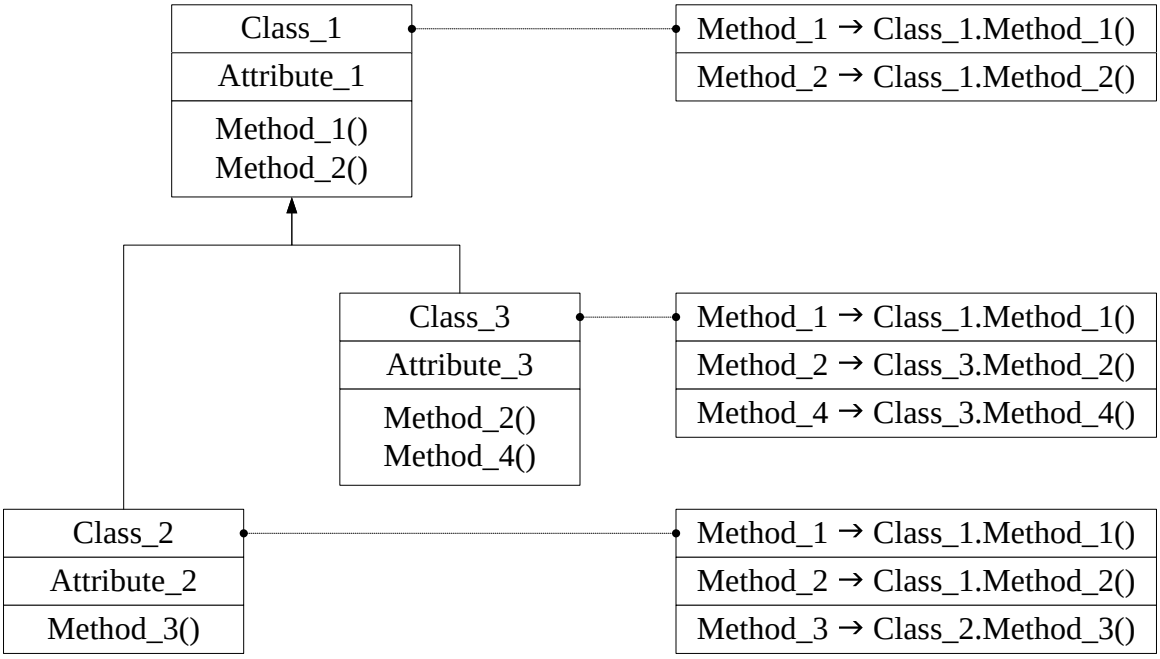

Figure 3. Methods Tables Within a Class Hierarchy

The implementation depicted in figure 3 builds a method table for each class containing a set of pointers to the methods applicable to that class. Other implementations using a single table for a class and all children are used by different compilers. Within these implementations, child classes add to the parent method table for both new methods and methods that override parent methods. Overridden methods are added to the table offset by a constant value from the parent method. This is invisible from a SC level, and can only be viewed by looking at the OC/EOC that allocates this memory, and calculates the offset. This mechanism is demonstrated in figure 4 where there is an offset for the overridden Method_2.

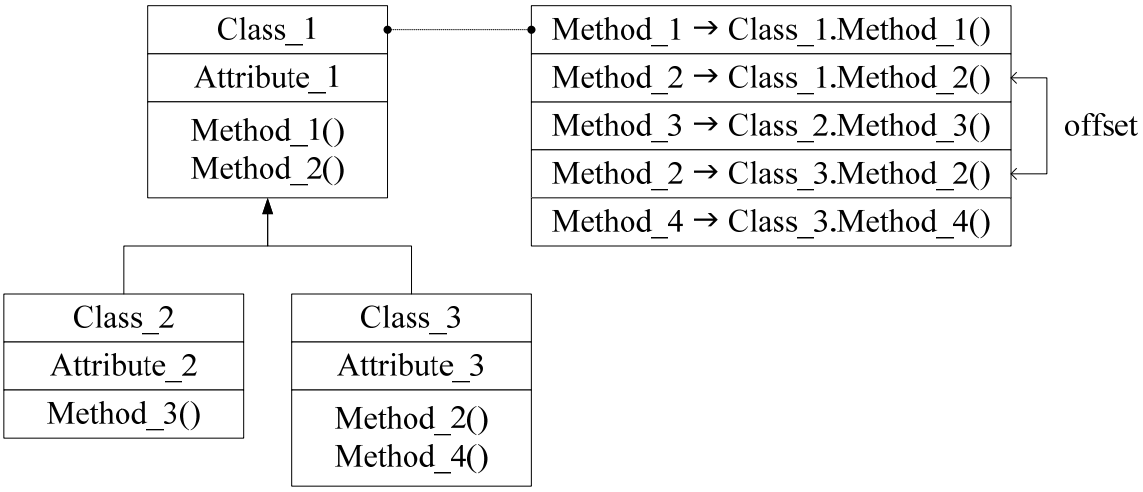

Figure 4. Class Methods Table With Offset

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

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

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

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

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

When doing an evaluation, the depth of the class hierarchy has to be considered as well as the amount of polymorphism used. C++ compounds this problem by allowing multiple inheritances, further complicating any analysis. The number of classes between the last child class and the base class (or classes for multiple inheritances) determines the complexity of tracing program flow as each class within the hierarchy re-uses and adds its own memory requirements to the vtable. When polymorphism is used, additional space is allocated within the vtable. Tracing program flow then becomes even more complex and has to examine the memory within the vtable to make sure that it is sufficient and contains the proper code. Further adding to the complexity of the analysis is the possibility that the memory for polymorphic functions could become corrupted if the offsets used to determine their location become corrupted or somehow altered.

2.2 CONSTRUCTORS.

As part of the class mechanism for C++ and Java, supporting methods known as constructors are required for a class [7]. As the name implies, constructors create an instance of a class by initializing the attributes (internal variables) of the object necessary to establish its initial state.

The issue is that, in both languages, there will always be a discrepancy between the SC and code created by the compiler (OC/EOC), because code has to be added to the constructors to at least initialize the object's variables and the object itself. In both languages, when no constructor is specified, the compiler automatically generates an initial constructor with no code in it except what is needed to create a reference to the object. In Java, the reference is the variable "this."

The SC and OC/EOC will differ, because there will be OC/EOC that appears like any other method signature. Specific things to look for are:

- Has a default constructor been created if there is none specified?
- For C++, if memory needs to be allocated and a default constructor created, does the memory get allocated within the object?
- In the OC, have all arguments in the constructor been accounted for? In C++, has memory been allocated?
- If there are multiple constructors, are all of them accounted for?

To resolve the issue, SCA tools should be developed, qualified, and used to examine every constructor and the corresponding SC or OC/EOC to ensure each has been successfully generated and corresponds with the constructor's method signature. For SC without a constructor, the tool should ensure a default constructor has been created. For all constructors,

there should be a reference to the particular object initialized. Some example constructor code is presented in figure 5.

```
class Airplane extends java.lang.Object {
public Airplane(String,int);
}

Code

Airplane Boeing747 = new Airplane("747", 1000);


Constructor – Method public Airplane(java.lang.String,int)
```

Figure 5. Constructor Example

The bytecodes for the constructor example presented in figure 5 are presented in table 1. In table 1, and all following tables where bytecode examples are given, the first column provides a line reference number, the second column provides the bytecode, and the third column provides a description.

Table 1. Constructor Example Bytecodes

Line Reference Number	Bytecode	Description
0	aload_0	Push instance of object –
1	Invokespecial #3	<Method java.lang.Object() – Base class for all objects
4	aload_0	
5	aload_1	
6	Putfield #5	String input argument
9	aload_0	
10	Iload_2	
11	Putfield #4	Integer input argument
14	aload_0	
15	aload_1	
16	Iload_2	
17	invokespecial #6	Store constructor data
20	Return	

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

A real danger exists when the compiler creates a copy constructor of an object requiring a deep copy, that is, an object that allocates dynamic memory. When this is used to create an object, memory will not be allocated and a program crash will occur when the object's variable is used. The following should be looked for when evaluating a C++ object's copy constructor:

- Has the copy constructor been created by the developer or compiler?
- Does it take into account dynamically allocating memory?

2.3 INITIALIZERS.

In addition to constructors, Java also has initializers that automatically move blocks of code into constructors. This, again, yields OC/EOC that is different from SC. A single-code block is declared at the beginning of the class. When the class is compiled, all code in the initialization block is moved into the constructor.

When initializers are used in Java, the SC and OC/EOC will always be different, as the compiler moves the code from the initialization blocks into the constructor. The more constructors, the more differences between the SC and the OC/EOC.

When analyzing Java code that uses initializers, one should ensure that each constructor contains the code within the initialization block. When no constructor is specified but there is initialization code, then the complier should create a new constructor with this code within it.

To resolve this issue, SCA tools need to be developed, qualified, and used to read the initialization code block from the beginning of the class. Then the number of constructors needs to be counted. As the tool looks over the OC/EOC, it should ensure that the initialization code is contained in every constructor. The code can be contained explicitly or as a branch to a subroutine that contains the initializer code.

The most important thing to look for is whether the code within the initializer block has been added to all constructors. The companion report shows an example of this, including bytecode [6].

2.4 FINALLY BLOCKS.

As part of the class mechanism for C++, supporting methods known as destructors are required for a class [7]. Destructors release the memory for the object and close any shared or managed resources. The destructor methods carry out whatever activities must be performed before an object is no longer needed (e.g., free-managed resources and tasks).

In Java, shared or managed resources are closed within finally blocks within methods. These always execute regardless of whether an exception was thrown or program execution exits before the end of the code in the method.

The Java compiler implements the finally capability by adding a branch to a subroutine containing the finally block code or by simply duplicating all the code within the finally block

before each exit point within a method whose code is surrounded by a try/catch block. Adding these subroutines or finally code blocks causes a gap between the SC and compiled Java OC/EOC.

The issue is that SC and OC/EOC will differ anytime there is a finally block of code, because the compiler adds a branch to a subroutine at each exit point in a method that branches to the code in the finally block or has a number of duplicate blocks of code inserted at each exit point. A major concern is how the compiler determines where the exit points are located within the method. The complexity of the method, the objects used, and the total number of lines of code in the method will determine the number and position of the exit points within a method.

The first thing to look for is the accuracy of the rules that determine the endpoints within a method. For safety-critical applications, these should be stricter than those used for conventional Java. The second item to look for is whether these rules are correctly applied. Lastly, the SC and OC/EOC code need to be compared to make sure that all subroutines branch to the correct location.

The following example demonstrates the analysis for Java. Table 2 shows the bytecodes used for branching to the "finally" subroutine. Depending on the size of the program, the compiler will use either a "jsr" or "jsr_w" opcode. The "et" opcode just uses the "pointer" (address) to return to program flow.

Table 2. Example Java Bytecodes

Opcode	Operand(s)	Description
Jsr	branchbyte1, branchbyte2	pushes the return address, branches to offset
jsr_w	branchbyte1, branchbyte2, branchbyte3, branchbyte4	pushes the return address, branches to wide offset
et	Index	returns to the address stored in local variable index

The example code is presented in figure 6.

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

Figure 6. Finally Example Code

The bytecode for the try/catch block code in figure 6 is presented in table 3.

Table 3. Java Bytecodes for the Finally Example

Line Reference Number	Bytecode	Description
0	iload_200	Push local variable 200 (arg passed as divisor)
1	ifeq 1100	Push local variable 100 (arg passed as dividend)
4	iconst_1	Push int 100
5	istore_3	Pop an int (the 100), store into local variable 3
6	jsr 24	*Jump to the subroutine for the finally clause <inserted by compiler>*
9	iload_3	Push local variable 3 (the 100)
10	ireturn	Return int on top of the stack (the 100)
11	iconst_200	Push int 200
12	istore_3	Pop an int (the 200), store into local variable 3
13	jsr 24	*Jump to the subroutine for the finally clause<inserted by the compiler>*
16	iload_3	Push local variable 3 (the 200)
17	ireturn	Return integer on top of the stack (the 200)
18	astore_1	Pop the reference to the thrown exception store Pop the reference to the thrown exception
19	jsr 24	Jump to the subroutine for the finally clause<inserted by the compiler>
20	aload_1	Push the reference (to the thrown exception) from local variable 1
23	athrow	Rethrow the exception
24	astore_2	Pop the return address, store it in local variable 2
25	Getstatic #8	Get a reference to java.lang.System.out
28	ldc #1	Push <String “Values Returned.”> from the constant pool
30	invokevirtual #7	Invoke System.out.println()
33	Ret 2	Return to return address stored in local variable 2

In table 3, the bytecodes are shown for the code in figure 6. In lines 6, 13, and 19, the compiler has inserted code that will branch to the finally clause, located at line 24. Lines 0 through 17 contain the code with the conditional loop and return values. From line 18 to 23, the exception is handled; notice the call to the finally subroutine at line 19. From line 24 to 33 is the actual finally clause code.

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

For smaller programs, the compiler simply recreated the code within the finally block a number of times, creating even more discrepancy between the SC and bytecode. Table 4 shows another bytecode representation of the same SC, however, this was compiled as a stand-alone class with no optimization. Here, the functionality in the SC's finally block is repeated for each exit point: lines 7-15, 21-29, and 45-53. Lines 31 through 43 are not executed and are dead code. The optimized version in table 8 is more efficient and has no dead code, using branches to a subroutine. Traceability is easier without the subroutine branching and could be considered to be the optimal way to compile the finally clauses.

Table 4. Alternate Java Bytecodes for the Finally Example

Line Reference Number	Bytecode	Description
0	iload_0	
1	ifeq 17	
4	bipush 100	
6	istore_1	
7	getstatic #2	//Field java/lang/System.out;Ljava/io/PrintStream;
10	ldc #3	//String done
12	invokevirtual #4	//Method java/io/PrintStream.println;(Ljava/lang/String;)V
15	iload_1	
16	ireturn	
17	sipush 200	
20	istore_1	
21	getstatic #2	//Field java/lang/System.out;Ljava/io/PrintStream;
24	ldc #3	//String done
26	invokevirtual #4	//Method java/io/PrintStream.println;(Ljava/lang/String;)V
29	iload_1	
30	ireturn	
31	astore_1	
32	iconst_0	
33	istore_2	

Table 4. Alternate Java Bytecodes for the Finally Example (Continued)

Line Reference Number	Bytecode	Description
34	getstatic #2	//Field java/lang/System.out;Ljava/io/PrintStream;
37	ldc #3	//String done
39	invokevirtual #4	//Method java/io/PrintStream.println;(Ljava/lang/String;)V
42	iload_2	
43	ireturn	
44	astore_3	
45	getstatic #2	//Field java/lang/System.out;Ljava/io/PrintStream;
48	ldc #3	//String done
50	invokevirtual #4	//Method java/io/PrintStream.println;(Ljava/lang/String;)V
53	aload_3	
54	athrow	

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

3. OBJECT-CODE BRANCH COVERAGE ISSUES REQUIRING ADDITIONAL CONSIDERATION.

The companion report established that OBC is not equivalent to MCDC in the general case [6]. The issues that need to be addressed when using OBC in place of MCDC are:

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

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

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

- The identification and verification of independence pairs. Automated tools that monitor conditions only when they are associated with a branch point (i.e., perform OBC) are not capable of determining the independence of all conditions in certain decisions consisting of three or more conditions.

To address these issues, one should understand what an SCA tool is monitoring, and thereby measuring, and the implications for the SCA results. For the first three issues, if the SCA tool is not capable of detecting and monitoring all decisions and conditions, then additional analysis will need to be performed to cover the gaps left by the tool.

For the fourth issue, MCDC requires that a specific combination of condition values be executed in order to show a condition's independence [6]. At the OC/EOC level, this means that a specific path must be taken through the predicate graph of the decision. Figure 7 shows the predicate graph for the expression "(A OR B) AND C" when short-circuit forms are used.

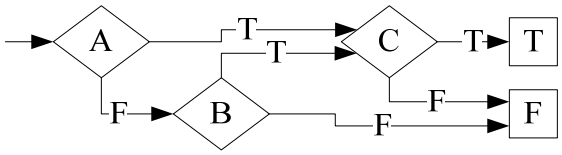

Figure 7. Predicate Graph for (A OR B) AND C

For MCDC, for a condition's independence to be demonstrated, it must be executed both True and False, and the decision's outcome must be different, and that condition's change must be the only significant change [9]. When that condition is the left-hand side (LHS) (right-hand side (RHS)) of a short-circuiting AND, the RHS (LHS), when executed, must return a True result [9]. When that condition is the LHS (RHS) of a short-circuiting OR, the RHS (LHS), when executed, must return a False [9]. This means that for A's independence to be demonstrated, B must be False when executed and C must be True when executed. For B's independence to be demonstrated, A must be False to execute B and C must be True when executed. For C's independence to be demonstrated, either A or B must be True to execute C.

Most OBC analysis tools will monitor whether each condition (A, B, and C) has been both True and False. Given that this is known for the expression and predicate graph in figure 7, it is known that C's independence has been demonstrated. It is not known if B's independence has been demonstrated because it is possible that after B was True that C was False instead of the required True. It is also not known if A's independence was demonstrated, because it is not known what values B and C executed after A was False and True, respectively.

To know if a condition's independence was demonstrated or not, it must be known what paths were taken through the predicate graph. If the coverage analysis tool's monitoring and measuring can be used to determine what paths were taken, and thereby what combinations of conditions were executed, then a proper MCDC analysis can be conducted. Otherwise, additional analysis to determine the paths will be required. As mentioned in Certification Authorities Software Team (CAST)-17, OCC/EOCC analysis should provide equivalent results to SCC analysis [10], and the only way discovered to accomplish that in this study was the execution of independence pairs [6]. Note that the results concerning OBC of short-circuited logic apply equally to both OOT software and non-OOT software.

4. SUMMARY.

This Handbook provides guidelines for developers, verifiers, and acceptors (generally regulators or their designees) into issues and acceptance criteria for the use of SCA at the SC versus OC/EOC levels within OOT in commercial aviation, as required by Objectives 5 though 8 of Table A-7 in DO-178B. The intent of the SCA is to provide an objective assessment (measure) of the completeness of the requirements-based tests and support the demonstration of the absence of unintended function.

Guidelines were provided for the combined use of SCC analysis and OCC/EOCC analysis for the following OOT features:

- Methods tables
- Constructors
- Initializers
- Finally blocks

The combined SCC and OCC/EOCC analyses differ from the current practice where the SCA is conducted at either the SC level or the OC level. If combined analyses are not desired, appropriate SC to OC/EOC traceability may be substituted. This traceability for all software levels requiring SCA also differs from the common practice where SC to OC/EOC traceability is needed for Level A only.

5. REFERENCES.

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

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

2. Chilenski, J.J., Heck, D., Hunt, R., and Philippon, D., "Object-Oriented Technology (OOT) Verification Phase 1 Report-Survey Results," FAA contract DTFACT-03-P-10383; deliverable, August 2004.

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

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

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

6. Chilenski, J.J. and Kurtz, J.L., “Object-Oriented Technology Verification Phase 3 Report—Structural Coverage at the Source and Object Code Levels,” FAA report DOT/FAA/AR-07/20, August 2007.

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

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

9. Chilenski, J.J., “An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion,” FAA report DOT/FAA/AR-01/18, March 2001, available at: http://faa.gov/aircraft/air_cert/design_approvals/air_software/research/.

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