DOT/FAA/AR-02/113

Office of Aviation Research
Washington, D.C. 20591

Issues Concerning the Structural Coverage of Object-Oriented Software

November 2002

Final Report

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

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

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

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

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT/FAA/AR-02/113		

4. Title and Subtitle	5. Report Date
ISSUES CONCERNING THE STRUCTURAL COVERAGE OF OBJECT-ORIENTED SOFTWARE	November 2002
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
John Joseph Chilenski, Thomas C. Timberlake, and John M. Masalskis	
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. NAS1-20341
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
U.S. Department of Transportation Federal Aviation Administration Office of Aviation Research Washington, D.C. 20591	14. Sponsoring Agency Code AIR-100

15. Supplementary Notes

The FAA William J. Hughes Technical Center Research Program Manager was Charles Kilgore.

16. Abstract

There is a desire and an emerging trend by suppliers of commercial airborne safety-critical systems towards the use of object-oriented technology (OOT). There are issues concerning the structural coverage of software and systems built using OOT. One of the issues is that RTCA DO-178B, which uses structural coverage as one of the adequacy measures for the requirements-based testing of software for commercial airborne computer-based systems, does not address OOT. This report identifies those issues for OOT features in general, the implementation of those features in the programming languages Ada95, C++, and Java, and the monitoring of the feature and implementation by structural coverage analysis tools. Alternatives for resolution of the issues are given, and the most appropriate resolution proposed. Some of the proposed solutions require that structural coverage be considered in ways that are not typically expected and possibly issue supplementary material to RTCA DO-178B and its supplement DO-248B. Some of the issues are not new to OOT, but it is anticipated that OOT will make their occurrence either more frequent or occur in new or different ways.

17. Key Words	18. Distribution Statement
Object-oriented technology, OOT, Structural coverage	This document is available to the public through the National 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	39	

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

TABLE OF CONTENTS

	Page
EXECUTIVE SUMMARY	vii
1. INTRODUCTION	1
1.1 Background	1
1.2 Related Documents	2
1.3 Approach	2
1.3.1 Literature Search	3
1.3.2 Literature Study	3
1.3.3 OO Criteria	4
1.3.4 Language Criteria	4
1.3.5 Tool Criteria	4
1.4 Organization of the Report	4
2. OBJECT-ORIENTED FEATURES	5
2.1 Class	6
2.1.1 Language Issues	7
2.1.2 Tool Issues	8
2.1.3 Acceptance Criteria	8
2.2 Inheritance	8
2.2.1 Language Issues	13
2.2.2 Tool Issues	14
2.2.3 Acceptance Criteria	14
2.3 Aggregation	15
2.4 Polymorphism and Dynamic Binding	15
2.4.1 Language Issues	18
2.4.2 Tool Issues	19
2.4.3 Acceptance Criteria	19
2.5 Encapsulation and Information Hiding	19
2.6 Run Time Type Identification	21
2.7 Implicit Type Conversions	21
2.7.1 Language Issues	22

2.7.2 Tool Issues 22
2.7.3 Acceptance Criteria 22

2.8 Templates 22
2.9 Exceptions 22
2.10 Excluded Features 23

2.10.1 Dynamic Class Loading 23
2.10.2 Dynamic Reclassification 23
2.10.3 Just-In-Time Compiling 23
2.10.4 Reflection 24
2.10.5 Garbage Collection 24

3. RELATED ISSUES 24

3.1 Data and Control Coupling 24
3.2 Deactivated Code 26
3.3 Object State Testing 27

4. RESULTS AND FURTHER WORK 29

5. REFERENCES 32

LIST OF FIGURES

Figure		Page
1	Workflow of the Study	2
2	Class Graphical Representation	6
3	Inheritance Representation	8
4	Flattened Inheritance	9
5	Dependency Changes Due to Inheritance	11
6	Override Effects	11
7	Call Change Effects	12
8	Polymorphic Hierarchy	15
9	Polymorphic Call	16
10	Conditional Call	17
11	Method Tables for Polymorphic Calls	17
12	Collaboration Diagram	20
13	Deactivated Code Example	26
14	Stack Class	27
15	Stack State Machine	28

LIST OF TABLES

Table		Page
1	Results Summary	29

EXECUTIVE SUMMARY

Object-oriented technology (OOT) has been extensively used throughout the non-safety-critical software and computer-based systems industry (e.g., the various window-based graphical-user interface icon-based operating systems, applications, and the internet). This technology is touted as solving many of the problems seen in software production through the reduction of complexity in the development of software and the potential for massive reuse and adaptation of previously developed software (e.g., objects) and patterns. In particular, the compilers and development environments for object-oriented programming generate a large amount of code automatically. There is also a large market of support tools, object libraries, and training supporting OOT. In response to this movement, there is interest in moving OOT into 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.

This report identifies issues concerning the effect of certain features of OOT on structural coverage. These issues can concern a specific OOT feature, its implementation within the programming languages Ada95, C++, or Java, or the monitoring of the feature and implementation by structural coverage analysis tools. Structural coverage is used within DO-178B as one of the adequacy measures for the requirements-based testing of software for commercial airborne computer-based systems. However, neither DO-178B nor DO-248B, which contains supplementary material to DO-178B, addresses OOT. Structural coverage of object-oriented software is one of the concerns highlighted by the FAA Chief Scientific and Technical Advisor (CSTA) for Aircraft Computer Software in 1998.

Some of the issues addressed in this report are not new to OOT. The issues exist in current software technologies, but their occurrence in OOT may be more frequent or may occur in new or different ways. Certain issues that are new to OOT, related to the increasing abstraction level between source and object code, may require rethinking of structural coverage in ways that are typically not expected. For these issues, new guidance supplementary to DO-178B and DO-248B may be needed. For those issues for which new guidance on the appropriate use of structural coverage may be needed, options and arguments are presented for consideration.

1. INTRODUCTION.

This report provides information to international certification authorities to assist with the development of policy and guidance for the use of object-oriented technology (OOT) to develop software for commercial airborne computer-based systems. The research focuses on the aspects of structural coverage that are impacted by the use of OOT.

1.1 BACKGROUND.

Object-oriented technology has been extensively used throughout the non-safety-critical software and computer-based systems industry (e.g., the various window-based graphical-user interface icon-based operating systems, applications, and the internet). Many people tout OOT as the solution to many of the problems seen in software production through the reduction of complexity in the development of software and the potential for massive reuse and adaptation of previously developed software (e.g., objects, frameworks, and patterns). In particular, the compilers and development environments for object-oriented programming generate a large amount of code automatically. There is also a large market of commercially available publications, support tools, frameworks, object libraries, and training supporting OOT. In response to this movement within the general software industry, there is interest in moving OOT into 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.

One area of concern, identified by Rierson [1], the CSTA for Aircraft Computer Software, is the issues surrounding the proper application of structural coverage analysis to certain features of OOT. Within this report, not only are features of OOT examined, but their implementation within the programming languages Ada95, C++, and Java are examined, as well as the monitoring of the feature and implementation by structural coverage analysis tools. Structural coverage is used within DO-178B [2] as one of the adequacy measures for the requirements-based testing of software for commercial airborne computer-based systems. However, DO-178B does not specifically address OOT, possibly because the use of OOT in safety-critical systems was not contemplated in 1992 when DO-178B was released. A document containing supplementary material to DO-178B published in 2001, DO-248B [3], also does not address OOT.

Some of the structural coverage issues addressed are not new to OOT. They exist in current software technologies, but their occurrence in OOT may be more frequent or may occur in new and different ways (e.g., deactivated code). Other structural coverage issues require the use of structural coverage in a different manner than is currently used (e.g., inheritance). For these issues in general, new guidance in addition to DO-178B may be needed. For those issues for which new guidance on the appropriate use of structural coverage may be needed, options are presented for consideration.

1.2 RELATED DOCUMENTS.

The Aerospace Vehicle Systems Institute (AVSI) has done some complementary work to this study and has published a number of guidelines. The AVSI documents contain some material on the structural coverage of certain OOT features:

a. AVSI. "Guide to the Certification of Systems with Embedded Object-Oriented Software," version 1.4, January 2002.

b. AVSI. "Guide to the Use of Dynamic Dispatch in Embedded Object-Oriented Software," version 1.7, October 2001.

c. AVSI. "Guide to the Use of Multiple Interface Inheritance in Embedded Object-Oriented Software," version 1.2, October 2001.

d. AVSI. "Guide to the Use of Multiple Implementation Inheritance in Embedded Object-Oriented Software," version 1.7, October 2001.

1.3 APPROACH.

This study was conducted using a five-element process, as shown in figure 1, to produce this report. In figure 1 the major flows are depicted with solid lines and the minor flows are depicted with dashed lines. In addition, there were feedback flows between each of the elements that are not shown in order to simplify the drawing (e.g., when looking at language criteria for certain language issues, OO issues were raised).

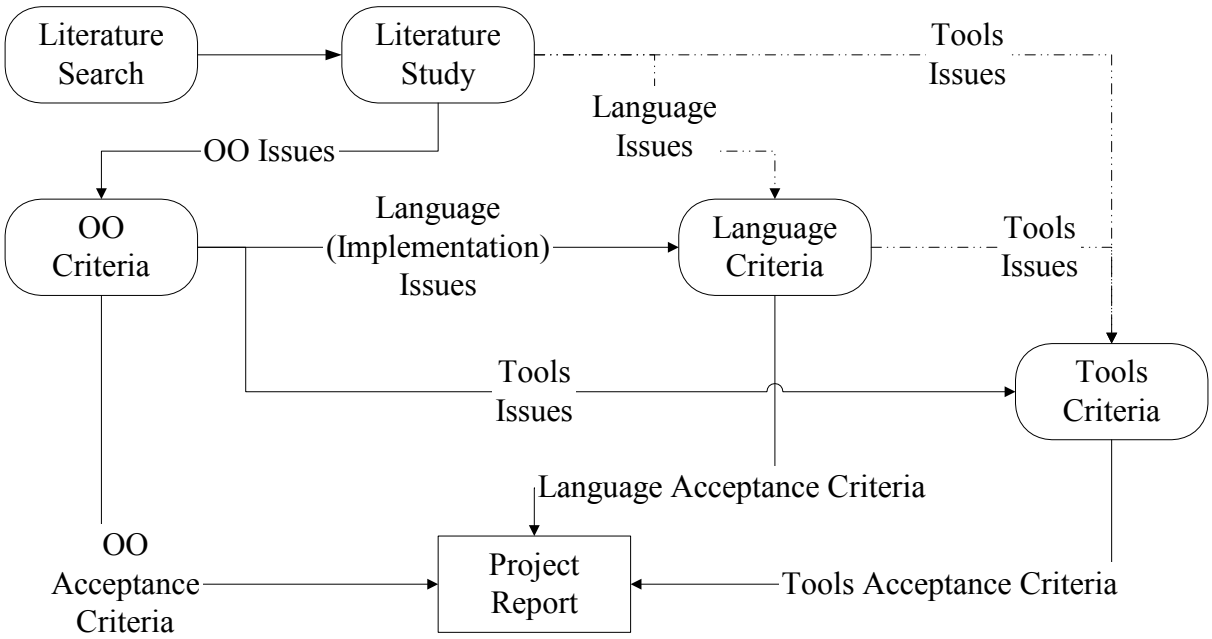

FIGURE 1. WORKFLOW OF THE STUDY

The literature search was conducted first, with the other four process elements from figure 1 conducted in parallel. Each of the five process elements from figure 1 is discussed in more detail in the following subsections.

1.3.1 Literature Search.

A search of peer-reviewed publications and web-based literature was conducted to locate previous work done on this topic. It was discovered that there are no peer-reviewed publications dealing with structural coverage, as defined within DO-178B [2], of object-oriented software. Instead, publications dealing with two topics that potentially lead to issues affecting structural coverage were identified.

1. The adequacy of testing object-oriented software, both requirements-based testing as well as implementation-based testing, and

2. The implementation-based (structural, white-box) testing of object-oriented software.

Even though there are no peer-reviewed publications, there are some white papers from structural coverage tool vendors available on the web.

1.3.2 Literature Study.

The publications identified in the literature search were examined for issues that need to be addressed in the definition or proper application of structural coverage for object-oriented software. Issues can be associated with:

- Object-oriented software features in general.

- The implementation of those features within a particular programming language. For this study, Ada95, C++, and Java.

- The tools used in the structural coverage verification (analysis) of object-oriented software.

Each of these categories was addressed in its own process element, as documented in sections 1.4.3-1.4.5. As mentioned in section 1.4.1, the issues had to be discovered indirectly. Issues concerning the adequate testing of general object-oriented features were examined to see if there were any implications for the structural coverage analysis of the implementation of those features. The methods for the implementation-based (structural, white-box) testing of object-oriented software and the issues concerning the application of those methods were examined to see if any of the methods could be adapted for structural coverage. Also, if the issues carried over to structural coverage, these issues that did apply to structural coverage were raised.

The methodology used to identify issues resulted in a number of object-oriented features being examined that resulted in no issues for structural coverage. Surprisingly, all the issues uncovered by this study concern issues associated with object-oriented software features, in

general, and the measurement of those features within the three programming languages. Though the object-oriented issues applied to the programming languages, there were no implementation issues beyond the general object-oriented (OO) issues. For some of the issues, the purpose and proper application of structural coverage may need to be extended beyond the current practice.

1.3.3 OO Criteria.

Both the general OO software features and the issues associated with those general features identified during the study were examined for their impact on structural coverage analysis. All of the features examined are documented in this report. For certain features, there are no issues impacting structural coverage, and the analysis stopped there. For those features where there were issues, broad solution possibilities for defining what it means to accomplish structural coverage in the verification of OO software in general were identified. These solution possibilities were targeted to the use of the feature without regard to how that feature was implemented in any particular programming language.

1.3.4 Language Criteria.

Both the general OO software features and the issues associated with those general features identified during the study were examined to determine if there were any specific issues in the implementation of the OO features in the programming languages Ada95, C++, and Java. Both the language standards and the outputs from compilers were examined.

Surprisingly, there were no structural coverage issues with the implementation of the general OO features in the three target languages beyond those of the general OO feature itself. Note that this study only considered the structural coverage aspects of the language implementations.

1.3.5 Tool Criteria.

How current structural coverage verification (analysis) tools deal with the issues defined during the study, both general OO as well as the specific implementations within Ada95, C++, and Java, were identified. For certain issues, other options for tool behavior were identified. Where multiple tool approaches exist, recommendations for a preferred approach are given.

1.4 ORGANIZATION OF THE REPORT.

This report is organized into four sections plus references. Section 1 introduces the context of the study.

Section 2 contains the specific OOT features and issues raised during the course of this study. There are three forms of features:

- Features that have no issues associated with them,
- Features that have issues that can be resolved, and
- Features that have issues that may make their use in safety-critical software unacceptable.

All the features are documented in the order that they were examined, except the excluded features, which have their own section.

- Class—the fundamental concept
- Inheritance—one of the fundamental building blocks
- Aggregation—one of the fundamental building blocks
- Polymorphism and Dynamic Binding—one of the fundamental principles
- Encapsulation and Information Hiding—one of the fundamental principles
- Run Time Type Identification—support mechanism
- Implicit Type Conversions—programming support mechanism
- Templates—support mechanism
- Exceptions—support mechanism
- Excluded Features
    - Dynamic Class Loading
    - Dynamic Reclassification
    - Just-In-Time Compiling
    - Reflection
    - Garbage Collection

A brief explanation is given of the feature. For those features with no issues, the feature is identified as having no issues. For those features with issues, the issues (general OO, language, and tools) are identified along with recommendations for acceptance criteria. For those features that were deemed unacceptable, the reason for unacceptability is given.

Section 3 contains some related issues concerning the use of OOT in general. These issues are not related to specific features of OOT, but are side effects of the use of OOT.

- Data and Control Coupling
- Deactivated Code
- Object-State Testing

Section 4 summarizes the conclusions of the study and identifies issues for further study, and section 5 lists the references.

2. OBJECT-ORIENTED FEATURES.

The results of examining features of OOT are documented within this section of the report. Classes, being the major organizational feature of OOT, are reported on first. Following classes, each of the features of a class examined during the course of this study is reported on, essentially in the order they were examined. For some of these features, issues concerning structural coverage were identified and are, therefore, documented. For other features, no issues were found. For certain issues, it was determined that these would not be acceptable in safety-critical systems, so no further investigation was conducted. These features are grouped together in section 2.10 regardless of the order they were examined.

2.1 CLASS.

OOT is a software development methodology based on the concept that all systems are composed of objects, and the relationships and interactions between those objects. The fundamental concept upon which OOT is founded is that of the class. Classes form the fundamental building blocks out of which object-oriented software and systems are created. All objects appearing within the system and software must belong to a class. The class is a template that defines the attributes and methods applicable to all objects of that class.

Attributes define the data structure and hold the corresponding values that describe the current state of the objects of a class. Note that there are two forms of attributes, i.e., class variables or static variables, which are shared by all objects of a class. Instance variables are attributes that apply to each individual object of a class. Each object of the class will contain its own set of instance variable attributes and share the common set of class variable attributes.

Methods define the operations that may be performed on the objects of a class, thereby defining the possible behavior of the objects of that class. These operations may be sensitive to the current state of the object and may update that state by changing the values of attributes. Note that there are two forms a method can take. If a method merely defines an interface without an implementation, a specification without a body in Ada terms, it is known as an operation. If it defines both an interface and an implementation it is known as a method.

By defining the set of values for an object and the operations that may be performed on those objects, the concept of a class is equivalent to the concept of a type as it appears in standard programming languages (e.g., Ada). However, the class also identifies the relationships, known as associations in the Unified Modeling Language (UML) [4], that can exist between objects of that class and objects of other classes (clients).

Figure 2 shows the diagrammatic representation used for classes in this report. This representation is based on the UML specification [4].

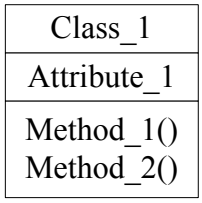

FIGURE 2. CLASS GRAPHICAL REPRESENTATION

As shown in figure 2, a class is represented as a box with three compartments. The top compartment lists the name of the class. The middle compartment lists the names of the attributes, if any. The bottom compartment lists the names of the methods, if any.

Classes support encapsulation through separation of the external (public) and internal (private) aspects of a class and its objects. Generally, the external aspects are known as the interface, while the internal aspects are known as the implementation. Clients of a class may only have

access to the interface of the objects of that class. This is also known as data hiding or information hiding.

Classes come in two forms: concrete and abstract. A class is concrete if objects may be created of that class. This means that the class must have both its interface and its implementation defined. A class is abstract if objects may not be created of that class. This means that the class will have either only its interface defined or only a portion of its implementation defined.

Depending on how one wants to look at it, there are either no OOT structural coverage issues associated with classes or all OOT structural coverage issues are associated with classes. In this report, the fundamental concepts have been broken out and applied to classes into their own sections.

2.1.1 Language Issues.

All three languages under consideration have a class mechanism. C++ and Java implement classes directly, using a classical class hierarchy. Ada95 uses an approach based on type derivations within packages. The difference is mainly syntactic, with each approach having advantages and disadvantages.

As part of the class mechanism for C++ and Java, supporting methods known as constructors (C++, Java) and destructors (C++) are required for a class. In Java, a finalizer, similar to a C++ destructor, may be provided.

As the name implies, constructors are responsible for creating an object of a class at the beginning of its existence, including the initialization of the attributes of the object necessary to establish its initial state. For example, consider that an airspeed indicator object should draw itself on the display when it is created.

The destructor and finalizer are responsible for cleaning things up when an object ends its existence. These methods perform whatever activities must be performed before an object is no longer needed. For example, consider again an object that draws itself on a display when it is created. When that object is no longer needed and is about to be destroyed, one of the things that it should do is erase itself from the display. The C++ destructor has an additional responsibility in that it is also responsible for cleaning up the memory that the object occupied. The Java finalizer does not perform this function because memory cleanup is the responsibility of the garbage collector.

Ada95 provides automatic initialization and finalization for instantiated objects. Ada95 also provides a mechanism, called controlled types, that gives the user additional control of these functions. The user can define, for a controlled type, an Initialize procedure that is invoked immediately after the normal default initialization and a Finalize procedure that is invoked immediately before default finalization. A third procedure, Adjust, is called for controlled types following any copy or assignment operation.

These supporting methods, constructors, destructors, and finalizers are not known to introduce any structural coverage issues in and of themselves. However, they can be implemented with side effects that can raise issues because these methods are employed by other features of the language (e.g., exceptions, destructors, finalizers; implicit type conversion and constructors). These potential interactions are discussed later in the report.

2.1.2 Tool Issues.

There are no tool issues associated with classes in general. Specific issues are brought out in subsequent sections dealing with specific features of classes.

2.1.3 Acceptance Criteria.

There are no constraints on the acceptance of classes in general. Specific issues are brought out in subsequent sections dealing with specific features of classes.

2.2 INHERITANCE.

Inheritance is one of the fundamental building blocks of OOT. It is a mechanism whereby a class is defined in terms of other classes (its parents), adding the features of its parents to its own without disturbing either the relationships between its parents and their clients or the parent's concrete implementations. It is generally used to define an Is_A or Is_A_Kind_Of relationship (e.g., a conic section is_a geometric shape, an ellipse is_a_kind_of conic section).

A class may have a single parent (single inheritance) or multiple parents (multiple inheritance). Figure 3 presents an example of single inheritance where two subclasses (Class_2 and Class_3) are inheriting from Class_1. Either the interface or the interface and implementation can be inherited. Where multiple inheritance is allowed, repeated inheritance is a possibility (two or more parents have a common ancestor in the class hierarchy).

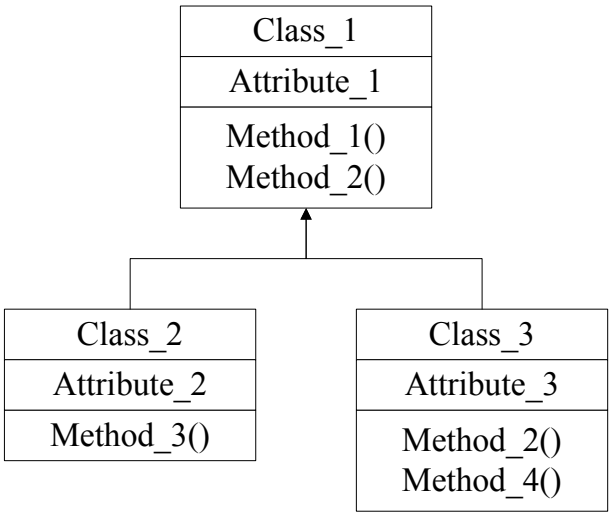

FIGURE 3. INHERITANCE REPRESENTATION

Inheritance is implemented by three basic mechanisms:

- Extension is the inclusion of the attributes and methods of ancestor classes (parents, their parents, etc.) in a subclass.

- Overriding is the definition of either an attribute or method in a class with the same signature as that in a parent class. A signature consists of the name of the feature, the type for attributes, parameter signatures for methods, and in some languages a return type for methods (e.g., Ada).

- Specialization is the definition of attributes and methods that are unique to that class.

Inheritance can be pure, where a subclass only extends its parent, or not pure. In pure inheritance, only extension and specialization are allowed. Class_2 in figure 3 is an example of pure inheritance. Class_2 has inherited Attribute_1, Method_1, and Method_2 from Class_1 (extension) and has added an Attribute_2 and Method_3 (specialization). Class_3 on the other hand is not pure. It has inherited Attribute_1 and Method_1 from Class_1 (extension) and has added Attribute_3 and Method_4 (specialization). However, it has overridden (or redefined) Method_2 from Class_1 with its own Method_2 (overriding).

To fully understand a class, one can flatten it. A flattened class is one where all attributes and methods, including those inherited from a parent class, are shown. A flattened inheritance hierarchy is one where all classes are shown in flattened form. Figure 4 presents a flattened inheritance diagram for the inheritance diagram given in figure 3.

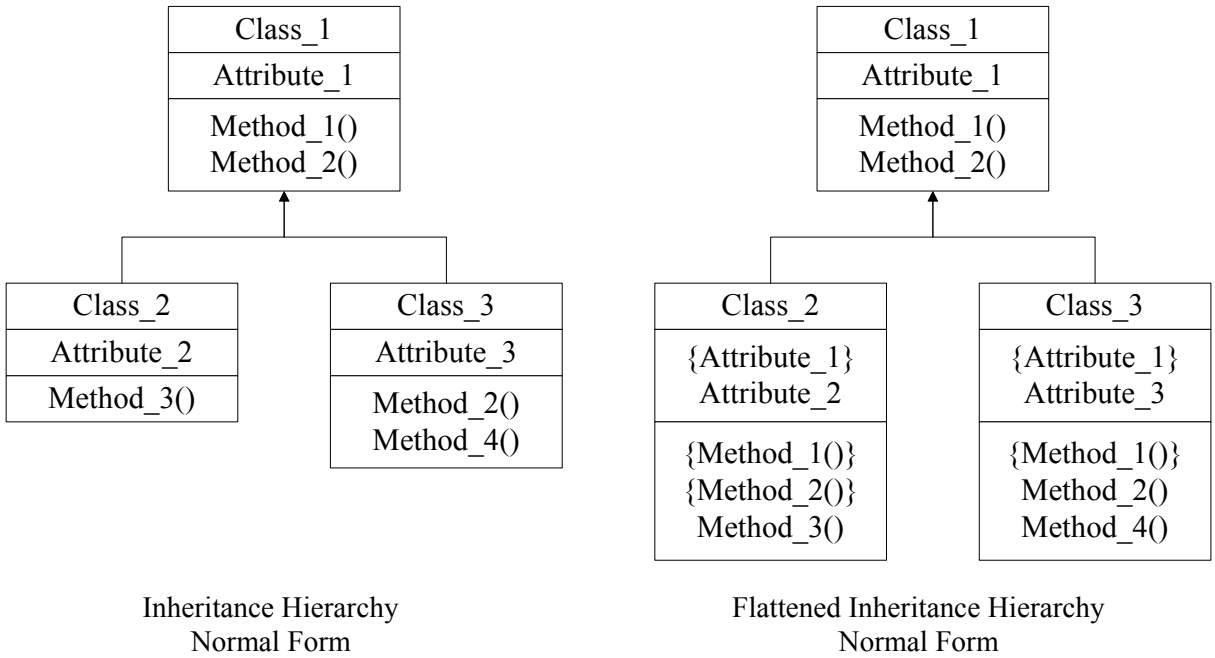

FIGURE 4. FLATTENED INHERITANCE

On the left-hand side of figure 4, the inheritance is presented in normal form, just as in figure 3. On the right-hand side of figure 4, the inheritance is presented in flattened form. In this report, the flattened form will enclose all inherited attributes and methods in curly braces. In some forms of flattening, inherited attributes and methods are annotated with a comment identifying where they were inherited from (e.g., Attribute_1 – Class_1.Attribute_1).

The general issue inheritance brings to structural coverage is: against what should the coverage be measured, i.e., concrete features only or concrete and inherited features? To answer this question, look at how OOT is implemented. For classes, each class keeps its own copy of the class variable attributes. For objects, each object keeps its own copy of the instance variable attributes. This means that if there is an Object_A of Class_1, it will have its own copy of Attribute_1 that can be acted on by either Method_1 or Method_2 defined in Class_1. An Object_B of Class_2 will have its own copy of Attribute_1 and Attribute_2. Attribute_1 can be acted on by Method_1 and Method_2 defined in Class_1 and by Method_3 defined in Class_2. Attribute_2 can only be acted on by Method_3. An Object_C of Class_3 will have its own copy of Attribute_1 and Attribute_3. Attribute_1 can be acted on by Method_1 defined in Class_1 and by Method_2 and Method_4 defined in Class_3. Attribute_3 can be acted on by Method_2 and Method_4 defined in Class_3.

Since multiple copies of attributes are made when using inheritance, it is clear that all attributes need to be tested in the context of all concrete classes within which they appear, whether defined explicitly within that class or inherited by it. For methods, however, multiple copies are not made. Only a single copy of each concrete method is created. That means that there is only a single copy of Method_1, Method_3, and Method_4, as well as Method_2 from Class_1 and Method_2 from Class_3.

Given the single copy of methods, one may be tempted to measure coverage against the concrete method only. One may even be tempted to test and record coverage for the concrete method in the context of the class that defined it and reuse that testing and coverage for all inherited instances (i.e., no retesting needed for inherited instances). However, this ignores the fact that the context of objects of one class is not guaranteed to be the same as the context of objects of other classes within the same hierarchy. In essence, the data and control coupling relationships between the defining class and the inheriting class may be different if overriding or specialization are present.

Specialization can indirectly change the behavior of inherited methods previously verified in the context of other classes. For an example, consider from figure 3 that Method_1 and Method_2 in Class_1 may have an interface through, and thereby a dependency on, Attribute_1. This is illustrated in the dependency graph on the left-hand side of figure 5, which shows that there is a dependency between Method_1 and Method_2 through Attribute_1. Consider further that Method_3 in Class_2 may act upon Attribute_1 in ways that effect the operation of Method_1 and Method_2. This is illustrated in the dependency graph on the right-hand side of figure 5. Note that Class_2 has a different set of dependencies than Class_1. This means that the testing of Method_1 and Method_2 within the context of Class_1, even though they have been shown to be correct in the base class (Class_1), does not guarantee that it will work correctly in the context of the derived class (Class_2).

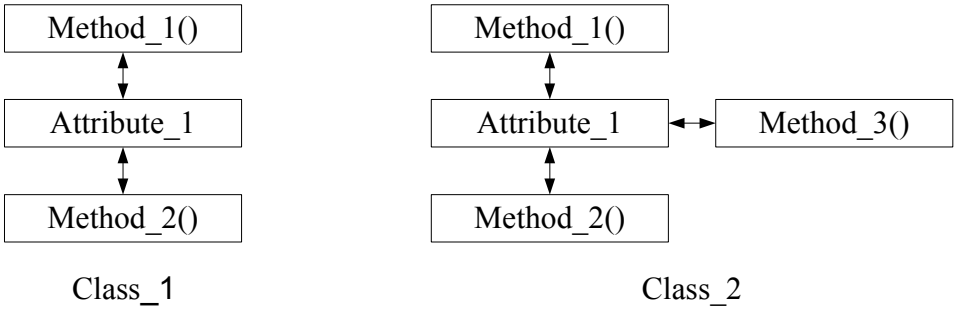

FIGURE 5. DEPENDENCY CHANGES DUE TO INHERITANCE

Overriding can directly change the behavior of inherited methods previously verified in the contexts of other classes. For example, consider the class given in figure 6. The normal form appears on the left-hand side of the figure while the flattened form appears on the right-hand side. Within figure 6, the methods have been annotated with the attributes they access and the methods they call. In the flattened form, the names were qualified with the class names (e.g., C_1.A_1).

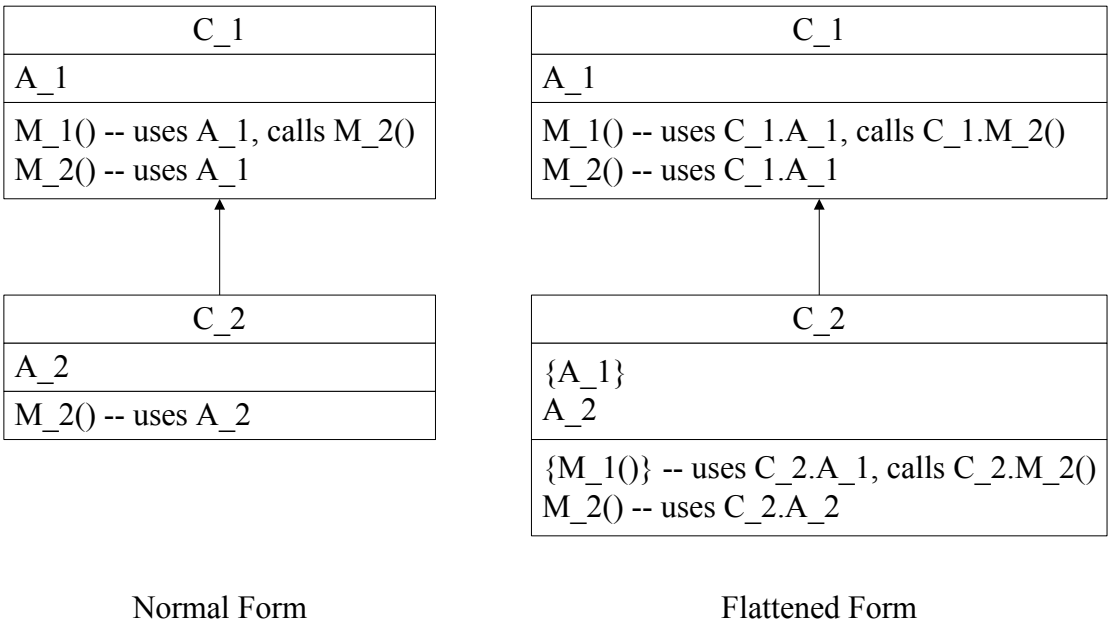

FIGURE 6. OVERRIDE EFFECTS

Examination of the right-hand side of figure 6 shows how overriding can modify the behavior of previously tested methods. Objects of class C_1 when called with M_1 will end up calling C_1.M_2 and using C_1.A_1. However, objects of class C_2 when called with M_1 will end up calling C_2.M_2 and using C_2.A_1 and C_2.A_2 instead. These changes will occur even though there has been no change to the source code for M_1! This example clearly shows the need for verification and coverage of all concrete attributes and methods within the contexts of the concrete classes that use them when overriding is involved.

Overriding can have an impact on data and control coupling by changing the data and control coupling relationships present within the software. Figure 6 shows that objects of C_1 have a data dependence on A_1, while objects of C_2 have a data dependence on A_2 instead. In addition, there is a change in the control flow of M_1 between the two classes that must be considered. C_1.M_2 and C_2.M_2 are both control dependent on M_1 in that M_1 is the only caller for those methods. This dependence is resolved by the class to which the object invoking M_1 belongs to.

This is equivalent to the change in a subprogram in a non-object-oriented calling sequence depicted in figure 7. The original calling tree is shown on the left side of the figure. M_1 calls M_2, which calls M_3. On the right side of the figure is the calling tree with a changed version of M_2: M_2′. Given this change to the system, does either subprogram M_1 or M_3 need reverification (i.e., did the change between M_2 and M_2′ invalidate any of the verification used to comply with DO-178B)? The answer is that it depends on the impact of the changes on the data and control coupling (i.e., integration or interaction) between (M_1, M_2′) versus (M_1, M_2), and between (M_2′, M_3) versus (M_2, M_3). For example, if the difference between M_2 and M_2′ is a changed Boolean expression, the tests for the integration verification of (M_1, M_2′) and (M_2′, M_3), as well as the tests providing MCDC for M_2′, may no longer be complete.

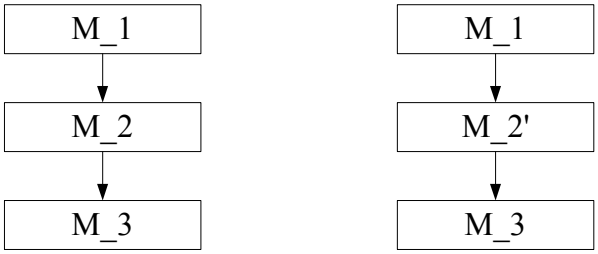

FIGURE 7. CALL CHANGE EFFECTS

The examples for specialization and overriding demonstrate that using these inheritance mechanisms may result in changes in the data and control coupling relationships between the defining class and the inheriting class. However, both of these examples can be extended to any base class (whether it defined or inherited the method) and a derived class.

For this reason, most of the published literature on the testing of object-oriented software requires the testing of each concrete method within the context of each concrete class that uses it (i.e., test the flattened class) [5]. Since structural coverage is supposed to be an adequacy measure of the requirements-based tests, structural coverage should be separately recorded for each concrete method within each concrete class that uses it. In essence, there is a need to measure structural coverage against the flattened class.

Always testing and covering the flattened class is a simple and straightforward approach. However, it should be acknowledged that not all instances of inheritance require reverification. For this reason, it is reasonable to allow a change impact analysis to identify exactly how much, and what type, of reverification is necessary for each instance of inheritance.

Note that the issues inheritance raises, concerning structural coverage, are independent of whether it is considered single or multiple inheritance. Multiple inheritance raises more issues concerning its use in safety-critical software than single inheritance does, but they are outside the domain of structural coverage [1].

The above discussion has concentrated on the inheritance of implementation. There is another inheritance mechanism known as inheritance of interface. An interface is known alternatively as an abstract class or method (UML, Ada95, Java), interface class (Java), virtual function (C++), or pure virtual function (C++). With the inheritance of an interface, there is only a promise to provide functionality, but no implementation. Because there is no code associated with an implementation when only an interface is inherited, there are no structural coverage issues associated with interface inheritance. Recall, however, that there is code associated with the implementation of an interface. Once there is an implementation, there may be structural coverage issues as discussed throughout this report.

2.2.1 Language Issues.

The following are some known differences in inheritance between the three object-oriented languages studied for this report. None of these differences are known to have any structural coverage issues associated with them. Ada95 and Java support only single inheritance of implementation, while C++ supports multiple inheritance of implementation. Ada95 and Java classes inherit from a single class hierarchy, while C++ allows multiple hierarchies. C++ and Java support both single and multiple inheritance of interface. In addition, there are ways in Ada95 to obtain the effect of multiple inheritance of either interface or implementation.

In Java, if a constructor is not provided explicitly, the compiler will create one that calls the parent class constructor. If the constructor exists but does not explicitly call the parent class constructor, Java will insert the call to do so. This compiler-generated code appears to have no structural coverage issues because it will always be executed when an object is created. A finalize method is not required by Java. However, if a finalize method is supplied in a derived class, it must call super.finalize or finalization will not occur for the base class. This is currently not required by the language specification, but appears to be enforced by some compilers.

In C++, if a constructor or a destructor are not provided explicitly, the compiler will generate a default. As with the Java case, this compiler generated code appears to have no structural coverage issues because it will always be executed when an object is created or destroyed.

A class is permitted to have more than one constructor. These different constructors may put the object into different initial states. As was pointed out in section 2.1, on classes in general, methods may be sensitive to the current state of an object. Thus, multiple constructors may have implications for data and control coupling for an initial object and its interaction with the

methods of its class. It does not appear that this data and control coupling is any different than that which is already present in the methods, and would be handled by normal verification of the methods.

2.2.2 Tool Issues.

There are two major approaches taken in current structural coverage analysis tools for object-oriented languages for recording the coverage of inherited methods.

- The first approach is to record coverage against the concrete methods only. For example, given the class in figure 4, all calls to Object_A.Method_1() will have coverage measured against Class_1.Method_1(), independent of the class Object_A belongs to. The tools in this category are the simplest to implement because they can use simple probes to instrument either the source or object code.

- The second approach is to record coverage against the flattened class. For example, given the class in figure 4, the call to Object_A.Method_1() will record coverage against Class_1.Method_1() if Object_A is of Class_1, the virtual Class_2.Method_1() if Object_A is of Class_2, and the virtual Class_3.Method_1() if Object_A is of Class_3. The tools in this category have slightly more complicated probes in that they must ascertain the class of the object invoking the method, and then record coverage against a virtual copy of the method appropriate to the flattened class.

There is an approach taken by some tool vendors where the vendor's tool records coverage against the concrete class, but the user documentation recommends the testing of the flattened class by only using objects of a single class to run the tests and record the coverage.

Given the direction seen in the object-oriented testing community toward testing of the flattened class [5], the preferred approach is to have the structural coverage analysis tools for object-oriented languages record coverage against the flattened class.

2.2.3 Acceptance Criteria.

As mentioned previously, the general issue inheritance brings to structural coverage is: against what should the coverage be measured: concrete features only or concrete and inherited features? As the previous examples have demonstrated, measuring structural coverage against the concrete methods is inadequate when the data and control coupling is impacted. Therefore, the following acceptance criteria is proposed for inheritance: structural coverage should be separately recorded for each concrete attribute and method within each concrete class that uses it, whether that class defines the attribute or method or inherits it (i.e., measure coverage against the flattened class). This recommendation is in agreement with the direction seen in the object-oriented testing community [5].

To support the testing of the flattened class, either full testing and structural coverage is required in each separate class context (i.e., test and cover the flattened class) or a change impact analysis

is required to identify the necessary reverification (testing and structural coverage) in derived classes.

2.3 AGGREGATION.

Aggregation is another one of the fundamental building blocks of OOT. It is a mechanism whereby a class is defined in terms of combinations of other classes. It is generally used to define a Has_A relationship (e.g., a circle has_a point as its center). It is generally implemented with attributes of one class being objects of another class. Aggregation is not unique to OOT, because existing programming languages allow composite types where each component of the composite can be of a different type (e.g., records in Ada). Aggregation was found to have no structural coverage issues associated with it.

2.4 POLYMORPHISM AND DYNAMIC BINDING.

Polymorphism is one of the fundamental principles of OOT. It is the ability of a name in software text to denote, at run time, one or more possible entities. The names of objects (in particular parameters), attributes, and methods may all be polymorphic in OOT. For example, given the flattened inheritance hierarchy from figure 4, repeated in figure 8, consider the programming text: Object_A.Method_2() appearing within the source code.

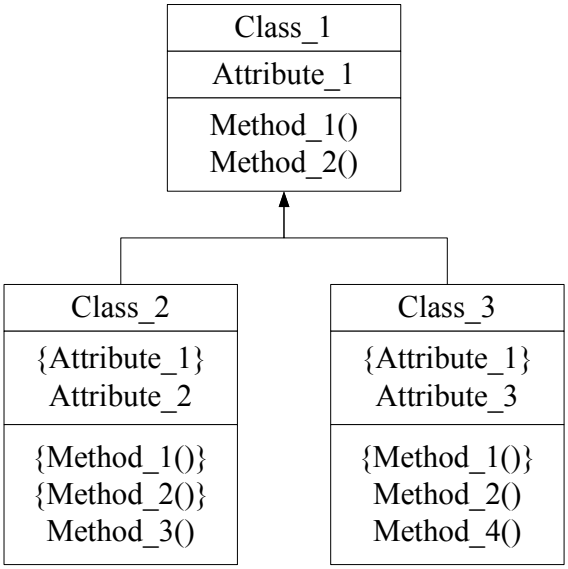

FIGURE 8. POLYMORPHIC HIERARCHY

Which Method_2 to call is dependent on which class Object_A belongs to, and Object_A may belong to multiple classes (Class_1, Class_2, Class_3), depending on the run-time state of the system. Object_A may be an instance of a more specific class (e.g., Class_2), while Method_2 may be a method of a more general class (e.g., Class_1). Polymorphism is generally supported by dynamic binding and dispatch.

Static binding, also known as static dispatch, is the matching of attribute references to attributes and calls to methods at compile time or link time. Static binding is currently what is implemented in traditional non-object-oriented languages. The binding is based on the signature of the element to be bound. Since traditional programming languages only allow, at most, one signature to be active in any particular scope, the reference is unambiguous.

Dynamic binding is the matching of attribute references to attributes and calls to methods at run time as opposed to compile time or link time. This results from a polymorphic reference or call. In essence, what is happening is that one of the key components of the signature, namely the class the object belongs to at the time of execution, is missing from the signature. Note that it is possible to have both static and dynamic binding present in OOT software and systems.

For static binding, it is sufficient to record coverage of the access or call statement itself because that access or call never changes. For example, given the class presented in figure 8, the call to Object_B.Method_3() is unambiguous, and a call to Class_2.Method_3 will appear in the object code.

However, when one sees the call to Object_C.Method_2(), it is not clear whether Class_1.Method_2 or Class_3.Method_2 is to be called. The appropriate method to call depends on the type (class) of Object_C. This is depicted graphically in the pseudo call tree depicted in figure 9.

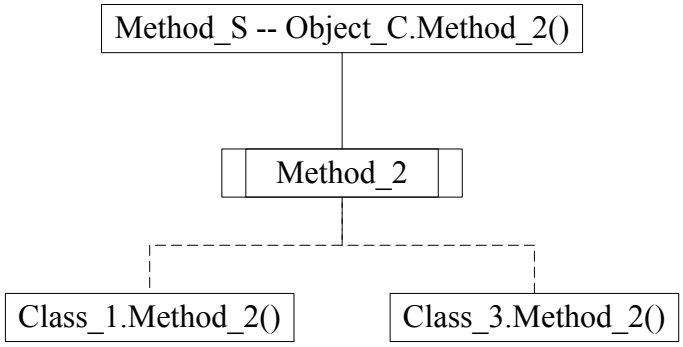

FIGURE 9. POLYMORPHIC CALL

This is equivalent to the non-object-oriented situation of a subprogram deciding which of multiple subprograms to call based on a parameter passed into the subprogram (i.e., a conditional call). This introduces at least control coupling between the calling subprogram and the callees and possibly data coupling between the subprograms and the caller of the calling subprogram. This is depicted graphically in figure 10 using standard Structured Analysis and Structured Design (SASD) notation [6], which is an alternate representation of figure 9.

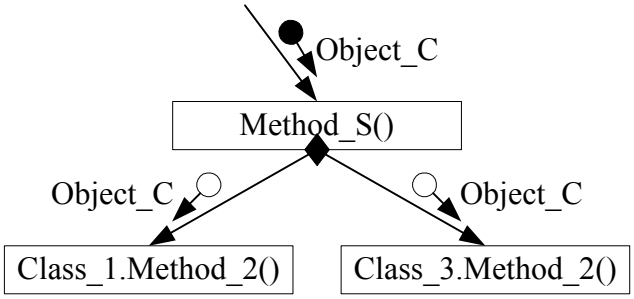

FIGURE 10. CONDITIONAL CALL

There are a number of ways in which dynamic binding can be supported, but only a single implementation was employed amongst the compilers examined during the language analysis portion of this study. This implementation builds a method table for each class containing a set of pointers to the methods applicable to that class. The table is built from flattening the class and is depicted in figure 11. In figure 11, the notation Method_m → Class_c.Method_m() denotes a pointer to the entry of the method with the qualified name given the unqualified name.

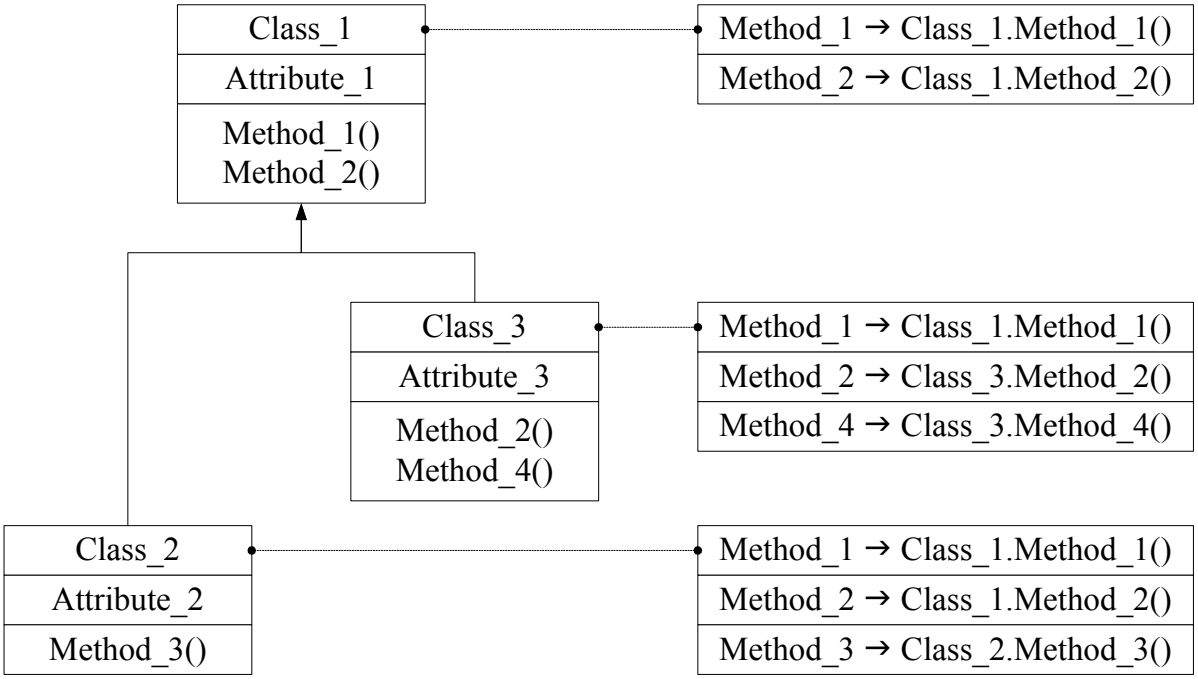

FIGURE 11. METHOD TABLES FOR POLYMORPHIC CALLS

This implementation then embeds within each object a pointer to the method table applicable to that object. The compiler and linker then emits code that will follow the pointer from the object to the method table, index the method table for the appropriate method, and then follow that pointer to the method to invoke (i.e., a jump table). Note that while the example used a method call, an attribute access could have been used just as easily.

The issue is: What level of structural coverage should be required for the polymorphic call or access? This study considered three possibilities.

1. One possibility is to treat the execution of the polymorphic reference (call or access) as sufficient. This approach covers the source code, but ignores the compiler- and linker-generated method (jump) table in the object code. It is possible that not every entry in the method (jump) table will be executed and, thereby, not every method will be executed for each class.

2. Another possibility is to treat the execution of all possible resolutions for the polymorphic reference as sufficient. This approach covers not only the source code, but also the method (jump) table the compiler and linker built in the object code. In fact, this approach will cover the method (jump) table multiple times: once for each polymorphic reference.

3. The final possibility is to require that every polymorphic reference and every entry in every method table be executed at least once. This approach covers the source code and the method (jump) table the compiler and linker built in the object code.

One of the goals of OOT is to remove some of the effort required to support certain functionality (e.g., polymorphism) from the software developer by having the compiler and linker provide that functionality and generate the object code for that functionality automatically. Essentially, the level of abstraction between source code and object code is being widened. Polymorphism and dynamic binding are the first instances in this report of OOT features where this auto-code feature of OOT was encountered, providing significant functionality. Auto-coding is not new to OOT. What OOT brings to the picture is that traditional auto-coding tools have accepted some form of specification as input and generated source code as output, while the OOT compiler and linker generates additional object code, both data and code structures, from the source code.

Traditionally, these data and code structures would have appeared in the requirements or architecture or source code, and would have gone through a DO-178B-compliant development and verification process (compliance to standards, traceability to requirements and design, coverage by requirements-based tests, structural coverage, etc.). In OOT, these data and code structures will only appear in the object code. A claim could be made that these data and code structures are traceable to the source code, just a broader interpretation of traceability than has been used for traditional languages. Under DO-178B, source code to object code traceability, and potentially additional verification of the object code, is only required for Level A software. In the context of OOT, this restriction may need to be broadened to include other levels (e.g., B and C). Further discussion is provided in section 2.4.2.

2.4.1 Language Issues.

All three languages studied for this report implement polymorphism and dynamic binding. No language-specific issues beyond the general issues already mentioned were discovered.

2.4.2 Tool Issues.

One of the major tool issues brought out by OOT is whether structural coverage should be obtained at the source code or object code level. This issue has broader applicability than just polymorphism and dynamic binding, but it is brought up here because this is the first time the issue of the compiler and linker generating additional object code and providing additional functionality beyond the source code has been encountered.

Currently, a number of structural coverage analysis tools provided for OO languages perform their analysis on the object code instead of the source code. There are two major reasons given in the literature of these tools for this choice. First, performing this analysis at the object code level can be done nonintrusively because the source code does not need to be instrumented. Secondly, coverage of the object code provides a better measure of testing thoroughness since it ensures that everything at the object code level has been executed, which source code coverage does not guarantee.

Currently, DO-178B [2, section 6.4.4.2.b, pg. 33] and DO-248B [3, FAQ#42, pg. 38] state that coverage of the object code is allowed as long as it provides the equivalent level of coverage provided by coverage of the source code. This guidance is still sound when OOT is being used.

Currently, DO-178B [2, section 6.4.4.2.b, pg. 33] and DO-248B [3, DP#12, pp. 97-98] state that source to object traceability is only necessary for Level A software. Because of the increased abstraction of source code in OOT, this guidance needs to be reconsidered, and potentially updated, when OOT is being used. Consideration should be given to source-to-object traceability for Levels A through C, or coverage of both source and object code for Levels A through C.

2.4.3 Acceptance Criteria.

The following acceptance criteria are proposed for polymorphism and dynamic binding: for software levels A-C, every polymorphic reference and every entry in every method table must be executed at least once.

The rationale for the above is as follows. Every class will have a methods table for all the methods appropriate to that class, whether inherited or defined (see figure 11). If every one of those entries has been executed, then every method will have been invoked at least once. Inherited methods will need to be invoked multiple times: once for the class that defines it and once for each class that inherits it. For example, consider Class_1.Method_1. This method will need to be invoked at least three times in order to achieve coverage of the methods tables. It will need to be invoked with an object of Class_1 in order to cover Class_1's method table entry for it, it will need to be invoked with an object of Class_2 in order to cover Class_2's method table entry for it, and the same for Class_3.

2.5 ENCAPSULATION AND INFORMATION HIDING.

Encapsulation and information hiding is one of the fundamental principles of OOT. As mentioned previously in section 2.1, encapsulation and information hiding is the separation of

the external (public) and internal (private) aspects of a class and its objects. Clients of a class may only have access to the interface of the objects of that class. Encapsulation and information hiding brings two issues to the table: access to the internals (e.g., private attributes, method implementations) for the purpose of coverage analysis and the effect on the visibility of data and control coupling.

Many publications on the testing of object-oriented software and systems identify the problems of verifying classes through only the interface to objects [5]. The main problem is how to verify the proper functioning of the object under test when the internals, particularly the state, cannot be set before a test is run or examined after a test is run to assure that the test passed for the correct reasons. The issue here is that encapsulation and information hiding can have a negative impact on the controllability and observability of an object for the purposes of class verification. However, if structural coverage analysis is to be performed with the assistance of a tool that instruments the source, then there should be no problem. Instrumenters, by their very nature, are intrusive and break information hiding by inserting probes into either the source or the object. These probes are capable of getting the internal information needed for a coverage analysis out into the open.

For methods, you may only have access to the interface, not the implementation (i.e., all method implementations are private). Not being able to look inside the methods can make it difficult to understand object interactions and prepare test cases to test such interactions.

It is even possible that for certain classes and objects (e.g., COTS library or framework) with standard software licensing, the developer will know nothing about the implementation. However, since current airborne software practice is for all software to comply with DO-178B, which includes reviews, analyses, and tests of the source code, access to the implementation will not be an issue. Given that access, tools can analyze the interactions between objects and make the results available for people.

For an example of the impact on data and control coupling, consider three objects as shown in the collaboration diagram presented in figure 12. In response to an event, Object_1 sends a message to Object_2 through Method_1. As a result of this message, Object_2 updates its attributes with values provided by Object_1. Later, Object_3 sends a message to Object_2 through Method_2. As a result of this message, Object_2 returns values from its attributes that Object_3 uses to update its own attributes. In this example there is a dependence between Object_3 and Object_1, but that dependence may only be discovered through the code.

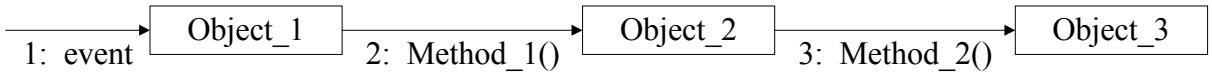

FIGURE 12. COLLABORATION DIAGRAM

2.6 RUN TIME TYPE IDENTIFICATION (RTTI).

Run time type identification (RTTI) is a support mechanism of OOT that allows an application to discover the exact class of an object, even when all that is available is a reference to an ancestor class. This mechanism requires that all classes used in the application be available at compile and link time. RTTI is commonly used to restore the original class type to an object that has been upcasted. For example, in order to store an ellipse object in an array of conic sections, you first upcast the ellipse object to a conic section and then store it in the array. When you retrieve it from the array, you use RTTI to restore it back to an ellipse (i.e., downcast). RTTI was found to have no structural coverage issues associated with it.

2.7 IMPLICIT TYPE CONVERSIONS.

Implicit type conversions are a programming support mechanism and not unique to OOT. Just as with implicit type conversions in traditional programming languages, what happens is that the compiler and linker generates additional object code. This additional object code will need to be examined for OOT, just as it is for non-OOT.

What OOT brings to the picture is the effect that implicit type conversions can have on the resolution of polymorphic references and on the data and control coupling occurring at the object or machine code level of the software.

Certain OO languages allow implicit type conversions to be performed dynamically to support any reference, polymorphic or not. This makes analysis and testing more difficult because the exact data type and implementation, both attributes and methods, cannot be determined statically, and the control flow of the OO program is less transparent. Static analysis cannot be used to precisely identify the dependencies in the program; instead, approximation techniques must be employed. This in turn makes it difficult to identify the change impact in regression testing.

For example, consider the following simple code sequence:

```
Object_A := Object_B + Object_C;
if Object_A.Valid() then
```

In this example, there may be temporary objects created for Object_A, Object_B, and Object_C requiring constructors and destructors to be called. These constructors and destructors may apply implicit type conversions in order for the creation and destruction operations to occur. These conversions can even change the type of a class object, impacting which attributes and methods are used, and can result in undesired behavior.

From the data and control coupling perspective, the constructors and destructors may have side effects that complicate the data and control coupling analyses. The addition of the implicit type conversions may further obscure the true data and control coupling relationships by changing the resolutions of polymorphic references.

2.7.1 Language Issues.

Ada95 and Java restrict implicit type conversions to those between primitive data types only. Further restriction is imposed in that implicit type conversions may only be widening from a less precise type to a more precise type so that no information would be lost.

C++ has a much more open policy on implicit type conversions. In particular, a C++ compiler is permitted to perform any implicit type conversion that conforms to the C++ rules, including those involving constructors. For example, a function expecting an argument of type T1 can be called not only with an argument of type T1, but also with an argument of type T2 if a conversion from T2 to T1 exists. Though this may cause unexpected results, it appears to have no structural coverage implications.

2.7.2 Tool Issues.

The analysis tools used to assist with the data and control coupling analysis may not be aware of everything the compiler or linker is doing when implicit type conversions are used. This may affect the trustworthiness of their analysis results.

2.7.3 Acceptance Criteria.

The following acceptance criteria are proposed for implicit type conversion: for software levels A-C, every implicit type conversion must be identified, its impact assessed, and executed at least once. Particular attention should be paid to analyze data and control coupling.

2.8 TEMPLATES.

Templates are a support mechanism of OOT. A template is a parameterized class, operation, or method with unbound (formal) parameters that must be bound to actual (type) parameters before it can be used. Templates are not unique to OOT as they appeared as generics in Ada83. What is unique in OOT is that other classes, both nontemplate and template, can inherit from a template class, whether or not the template class has been instantiated. Templates are currently provided, with some semantic differences, in both Ada95 and C++. There is a proposal to include them in the next version of the Java language specification. Templates were found to have no structural coverage issues associated with it.

2.9 EXCEPTIONS.

Exceptions, in the general sense, are a support mechanism of OOT for dealing with errors and other exceptional conditions that may arise during program execution. They are not unique to OOT because they have existed in previous languages like Ada83. What is unique in OOT is that when an exception is raised and propagates to a handler, the destructors or finalizers for all the objects that need to be removed must be invoked. Exception handling is provided, with some semantic differences, in all three languages examined in this report. Exceptions were found to have no structural coverage issues associated with it. Note, however, that if destructors or

finalizers have side effects, those side effects may have data and control coupling implications when they are invoked during the handling of an exception.

2.10 EXCLUDED FEATURES.

Certain features of OOT were not examined in this report because there were questions as to whether they should be used in safety-critical systems. These features include dynamic class loading, dynamic reclassification, just-in-time (JIT) compiling, reflection, and garbage collection. Primarily, these features impact either the predictability or repeatability of the resulting code expected in safety-rated software or the satisfaction of objectives in DO-178B. In the sections below, each feature is explained, and the issue that brings the feature into question is identified.

2.10.1 Dynamic Class Loading.

Dynamic class loading is the loading of classes during run time by the run-time support system. This is opposed to the loading of compiler and linker identified classes during start-up. This mechanism allows an application to be built without the necessity of precompiling and statically linking all of the classes. One use Java makes of this mechanism is to load classes it does not have access to when an application is built in order to support the running of distributed applications over the web. The dynamic loading has the side effect that the class tree is rewritten to include the dynamically loaded class. This self-modifying code feature brings into question its acceptability in safety-critical systems.

2.10.2 Dynamic Reclassification.

Dynamic reclassification is the ability of an object to change its class membership during run time between the time it is created and the time it is destroyed. Reclassification goes beyond the normal RTTI support for upcasting and downcasting used to support polymorphic data structures (e.g., an array of conic sections) (see section 2.6). An example of undesired dynamic reclassification would be a square object reclassifying itself as a parabola partway through its lifetime. Note that polymorphic parameters can change their membership in different invocations of their method depending on the actual parameter supplied. This is considered normal because the parameter does not change its membership during its invoked lifetime. Dynamic reclassification is probably not acceptable in safety-critical software because it raises data and control flow undecidability issues, which impacts the ability to perform data and control coupling confirmation.

2.10.3 Just-In-Time Compiling.

There are two forms of JIT compiling used in OOT. The first is a mechanism to allow an application to be built when not all of the source or object code is available. As the application is running and it comes to a point where it needs the missing piece, it will compile and install that piece and then resume running. Not only is this a self-modifying code issue, but the absence of complete source code prevents satisfaction of numerous objectives in DO-178B, particularly those concerning timing and memory usage.

The second form of JIT is a technique used in some virtual machine implementations, particularly Java (JVM), to improve program execution time. Through JIT, a bytecode method is translated into a native method on the fly, to reduce the overhead of executing the method by interpreting the bytecodes. It is more likely that traditional ahead-of-time compilation will be used in hard real-time embedded systems to generate native code. Should JIT methods be employed in the future, they will require further analysis for acceptability.

2.10.4 Reflection.

Reflection is a Java-specific mechanism that allows an application to interrogate a class to find out the method names, field names, and constructor names for the class at run time. Reflection is used to support dynamic class loading, discussed in section 2.10.1. The Reflection Application Programmers Interface (API) allows a running program to invoke the methods and modify the fields of the class, thus, interrogated, even when that class is not available at compile and link time. The self-modifying code aspects of reflection bring into question its acceptability in safety-critical systems.

2.10.5 Garbage Collection.

For systems using dynamic random access memory (DRAM) there is an issue as to whether that memory is to be explicitly managed, generally by the application, or implicitly managed by the run-time support. For explicit management, the application makes explicit calls to allocate and deallocate sections of memory, while for implicit management the application makes calls to allocate sections of memory, but does not make calls to deallocate it. The implicit management technique is generally known as garbage collection. In this technique, a garbage collector is periodically run and reclaims memory that is no longer in use. Additionally, it may move the contents of memory in order to compact it. The nondeterminism introduced by the timing impact of when the garbage collector will run, and how long it will take, as well as the question of where in memory the data will be, probably makes the use of a garbage collector unacceptable in safety-critical software.

3. RELATED ISSUES.

This section of the report looks at three issues that came up during the course of the study. These issues are not directly related to OOT features, but instead, result as side effects of OOT. The issues are data and control coupling, deactivated code, and object-state testing. Each is covered in its own subsection.

3.1 DATA AND CONTROL COUPLING.

Data and control coupling are not unique to OOT. What OOT brings to the picture is that data and control coupling relationships can be far more complicated and obscure in OOT than they are in traditional (functional) systems and software. This report has included the issues of data and control coupling within the sections of the affected object-oriented features. This section provides some higher level discussion as well as pointers to the sections in the report dealing with the details.

One impact on data and control coupling is in the nature of OOT. OOT encourages the development of many small, simple methods to perform the services provided by a class. Most of the control flow is moved out of the source code through the use of polymorphism and dynamic binding. In essence, the control flow, and thereby the control coupling, will become implicit in the source code, as opposed to being explicit. There is a similar effect on the data flow and, thereby, the data coupling.

OOT also encourages hiding the details of the data representation (i.e., attributes) behind an abstract class interface. Suggested best practice is that attributes of an object should be private, and access to them only provided through the methods appropriate to the class of the object. Being able to access attributes only through methods makes the interaction between two or more objects implicit in the code.

Section 2.1 on classes provided an example where data and control coupling were impacted by constructors. If a class has multiple constructors that put a new object into different states, these different states may impact how the methods behave in subsequent calls.

Section 2.2 on inheritance provided examples where data and control coupling were impacted by specialization and overriding. An added method in the derived class changed the data dependencies between inherited methods. An overridden method changed the data and control dependencies between inherited methods (and attributes).

Section 2.4 on polymorphism and dynamic binding provided an example where data and control coupling were impacted by dynamic binding. The dynamic binding influences the control flow, and potentially the data flow, between methods with polymorphic references.

Section 2.5 on encapsulation and information hiding provided an example where data and control coupling were impacted by encapsulation and information hiding. Encapsulation and information hiding can obscure the coupling between objects when intermediary objects are involved.

Section 2.7 on implicit type conversions provided an example where data and control coupling were impacted by implicit type conversion. The constructors and destructors invoked during implicit type conversions may have side effects that complicate the data and control coupling analyses. Implicit type conversions may further obscure the true data and control coupling relationships by changing the resolutions of polymorphic references.

Section 2.9 on exceptions provided an example where data and control coupling were impacted by exception handling. The destructors invoked during exception handling may have side effects that complicate the data and control coupling analyses.

In summary, it appears that the major impact of OOT on structural coverage will be in the area of data and control coupling. The standard code coverage measures of statement coverage, decision coverage, and modified condition and decision coverage were not impacted by OOT within the three languages investigated during this study.

3.2 DEACTIVATED CODE.

Deactivated code is not unique to OOT. What is unique to OOT is that several variations of this can occur, some of which give the impression of being dead code, i.e., classes in a library not used, methods of a class not called in a particular application, methods of a class (abstract) overridden in all subclasses, or attributes of a class not accessed in a particular application.

For example, consider the situation presented in figure 13.

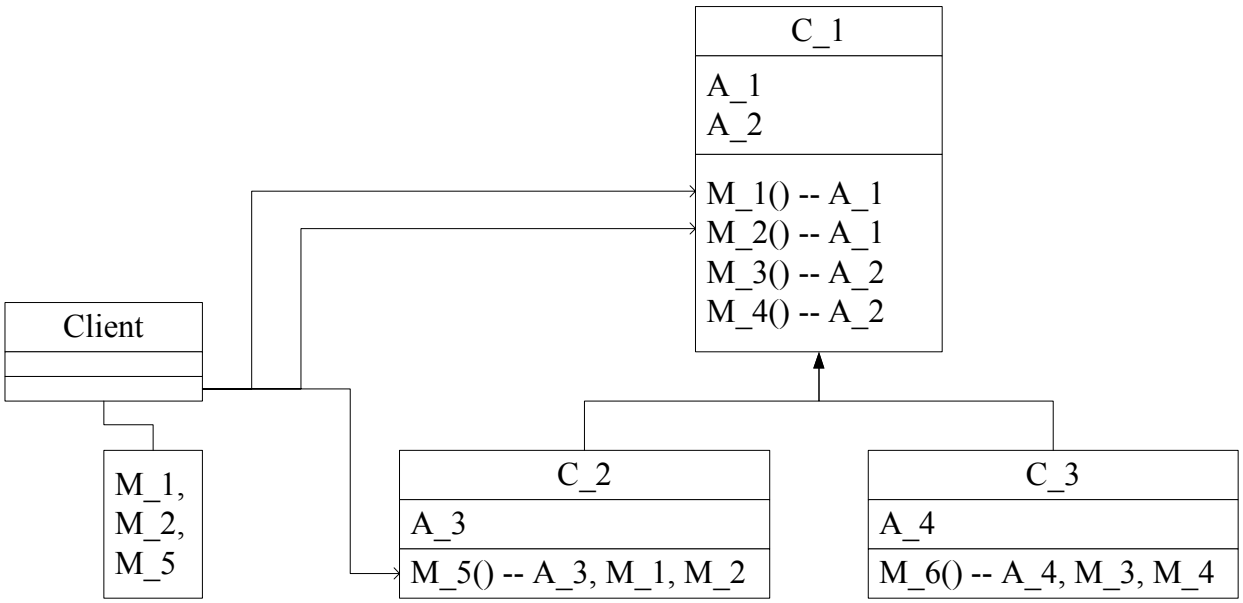

FIGURE 13. DEACTIVATED CODE EXAMPLE

Figure 13 presents a class being used by a client (the system). Within this diagram, as was done within figure 6, the methods have been annotated with the attributes they access, and the methods they call, as is the client. From the point of view of the client, class (C_3), methods (M_3, M_4, and M_6), and attributes (A_2 and A_4) appear to be dead code (i.e., not used by this system).

One approach would be to treat the unused classes, methods, and attributes as dead code and remove them for this system. However, the guidance given in DO-178B says that dead code is due to an error, but that does not appear to be the case in figure 13. The C_1 class has been intentionally designed and implemented with two subclasses (C_2 and C_3). At least, it is assumed that the design and implementation was intentional, as opposed to accidental or erroneous. The current client (system) is only making use of one of those subclasses (C_2), but there is potential use by other clients (systems) of the other subclass (C_3). Therefore, the unused classes, methods, and attributes should be treated as deactivated code. Note that it would be better if the intent behind the design and implementation was known, rather than assumed, as was done in the preceding analysis.

A complication can arise in the interaction between the use of a smart compiler or linker and a structural coverage analysis tool. For the uninstrumented source, the unused classes, methods, and attributes may not be loaded into the system, but they are in the instrumented source. This will lead the coverage analysis process to show that no tests cover the unused entities. However, this is not a problem as they are not loaded into the final system.

Smart compilers and linkers also present some issues of their own. Some smart compilers and linkers will remove only entire unused classes (e.g., C_3 in figure 13). Others will not only remove entire unused classes, but also unused attributes (e.g., A_2 of figure 13) or unused methods (e.g., M_3 and M_4 of figure 13) of used classes. The effects on the resulting executable code will be different and will need to be investigated.

3.3 OBJECT STATE TESTING.

Object state testing in OOT is an issue that was investigated because of the number of papers and technical reports published on both its necessity and the difficulties of properly achieving it.

State and state-based testing are not unique to OOT. What is unique to OOT is that potentially every object is its own little state machine. The state of an object is defined by the values held by its attributes. The issue of object state is related to the issue of context brought out in section 2.2 on inheritance.

For example, consider the stack class presented in figure 14.

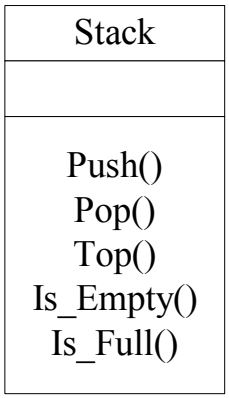

FIGURE 14. STACK CLASS

There are five methods applicable to any object of this class (Push, Pop, Top, Is_Empty, Is_Full). Each of these methods may have different behavior depending on the state of the stack object when they are called. These behaviors can be summarized by the stack state machine shown in figure 15. Notice that Is_Empty and Is_Full never change the state of the stack and never raise exceptions. Top never changes the state either, but if it is called when the Stack is in the Empty state an exception will be raised. Push and Pop can change the state, but not change the state and not raise an exception, or not change the state and raise an exception.

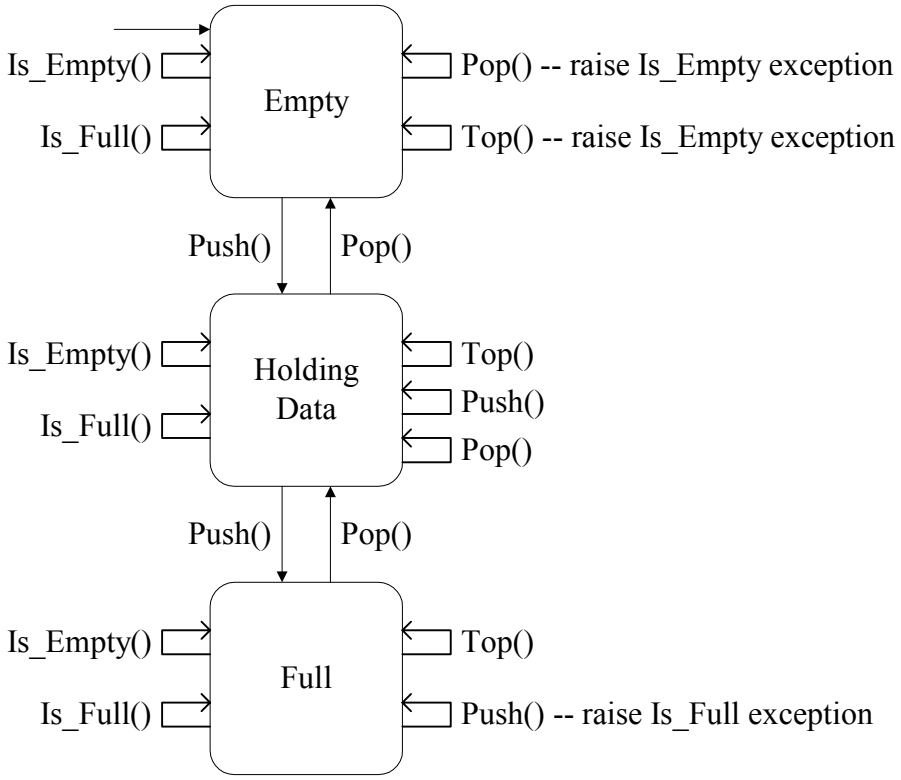

FIGURE 15. STACK STATE MACHINE

Though the proper verification and testing of object state appears to be an important issue with the use of OOT, and is crucial for certain kinds of classes (e.g., a stack), it is not an issue for the current measures used for structural coverage (statement coverage, decision coverage, modified condition and decision coverage), as given in DO-178B. The reason for this is that the semantics behind state are at the design and requirements level of the software, and are not available at the source code level. The current measures only concern themselves with the executions occurring at particular statements. Statement coverage and decision coverage do not concern themselves with the associations of the execution of statements and the values of the data alive at those statements. Modified condition decision coverage (MCDC) concerns itself only with the association at a single decision. This concern is not broad enough to encompass full state coverage. As an example, consider that to cover the methods of stack only requires the testing of each method in two of the three possible states.

However, object state testing appears to have implications for data and control coupling. Notice that the behavior of Push, Pop, and Top are all dependent on the current state and, thereby, dependent on previous executions of the methods. For example, consider that the execution of Push, which places the stack into the Full state, changes how Push will behave the next time it is called if the stack remains in the Full state.

4. RESULTS AND FURTHER WORK.

There is a desire and an emerging trend by suppliers of commercial airborne safety-critical systems towards the use of OOT. One area of concern, identified by Rierson [1], is the issues surrounding the proper application of structural coverage analysis to certain features of OOT. Structural coverage is used within DO-178B [2] as one of the adequacy measures for the requirements-based testing of software for commercial airborne computer-based systems. However, DO-178B does not specifically address OOT, possibly because the use of OOT in safety-critical systems was not contemplated in 1992 when DO-178B was released. This report identifies the issues concerning the structural coverage of certain features of OOT; the implementation of those features within the programming languages Ada95, C++, and Java; and the analysis of those features and implementations by current structural coverage analysis tools for object-oriented languages.

A summary of the OOT features examined, the role they play, and the results of this study are presented in table 1. In table 1, the issues column is supported by three types of issues: whether there was a coverage issue or not, whether there was a tool issue or not, and whether there is a data and control coupling issue or not. For the excluded features, all entries are not applicable (N/A).

TABLE 1. RESULTS SUMMARY

Feature	Role	Issues		
		Coverage	Tool	Coupling
Aggregation	Fundamental building block	No	No	No
Class	Fundamental concept	No	No	Yes
Dynamic Class Loading	Support mechanism	N/A	N/A	N/A
Dynamic Reclassification	Support mechanism	N/A	N/A	N/A
Encapsulation and Information Hiding	Fundamental principle	No	No	Yes
Exceptions	Support mechanism	No	No	Yes
Garbage Collection	Support mechanism	N/A	N/A	N/A
Implicit Type Conversions	Support mechanism	No	Yes	Yes
Inheritance	Fundamental building block	Yes	Yes	Yes
Just-In-Time Compiling	Support mechanism	N/A	N/A	N/A
Polymorphism and Dynamic Binding	Fundamental principle	Yes	Yes	Yes
Reflection	Support mechanism	N/A	N/A	N/A
Run Time Type Identification	Support mechanism	No	No	No
Templates	Support mechanism	No	No	No

Based on the results from the analyses of the OOT features performed during this study, it is concluded that there are two areas where the use of OOT impacts structural coverage. The first area of impact concerns whether measuring coverage against the source code only is sufficient. This area is important due to the widening of the abstraction level between source code and object code. Inheritance and polymorphism with dynamic binding are two OOT features where simple source code measurement may not be sufficient and are identified in table 1 as the only features with coverage issues. The second area of impact concerns the confirmation of data and

control coupling. Nearly every non-excluded OOT feature examined had data and control coupling concerns. Table 1 shows that only aggregation, Run Time Type Identification (RTTI), and templates were not concerns.

The general issue inheritance brings to structural coverage is: Against what should the coverage be measuredó concrete features only or concrete and inherited features? The analysis presented within this report shows that coverage should be measured against both concrete and inherited features. This goes beyond measuring coverage against the source code only, as there is source code for concrete features within the class where it is defined, not within the class(es) where it is inherited. This finding is in agreement with the general literature on the adequacy of testing inheritance [5]. Note that the measurement of coverage against concrete and inherited features does not necessarily require unique testing for concrete and inherited features. The same set of tests could be run with objects of the defining class and rerun with objects of the inheriting classes.

The general issue polymorphism with dynamic binding brings to structural coverage is: For compiler- and linker-generated data and control structures beyond normal source to object traceability (e.g., auto-code), should coverage be measured against the source code only, the object code only, or some combination? The analysis presented within this report shows that coverage should be measured against a combination of source code and object code for Level A-C software. This finding goes beyond DO-178B, where source-to-object traceability is only an issue for Level A software.

The OOT features examined in this report, both general OO features as well as language implementations for those features, were analyzed for structural coverage implications only. These features and implementations were identified through a literature search where there were questions about the proper requirements-based or implementation-based testing of the feature or implementation. There may be other features of OOT requiring investigation that were not identified in the literature search.

As mentioned previously, nearly every OOT feature examined had data and control coupling concerns. Unfortunately, data coupling, control coupling, and their confirmation are not well defined in either DO-178B or DO-248B. Within this report, if an OOT feature lead to either a data dependence or a control dependence, then a data coupling or a control coupling resulted, requiring verification. No attempt was made to specify what verification was needed for the coupling.

Both the general object-oriented software features and the issues associated with those features were examined to determine if there were any specific issues in the implementation of the object-oriented features in the programming languages: Ada95, C++, and Java. Surprisingly, there were no additional structural coverage issues with the implementation of the general OO features in the three target languages beyond those of the general OO feature itself.

How current structural coverage verification (analysis) tools deal with the issues uncovered during the study were identified. Table 1 shows that inheritance, polymorphism with dynamic

binding, and implicit type conversions were all general object-oriented features found to have tools issues. For inheritance, multiple approaches currently exist in different tools: recording coverage against concrete methods only versus recording coverage against the flattened class. This report recommends that the preferred approach is to have the structural coverage analysis tools for object-oriented languages record coverage against the flattened class. For polymorphism with dynamic binding multiple approaches also exist: source versus object code coverage. This report recommends that the preferred approach be a combination of the two: coverage of the polymorphic references at the source code level and coverage of the method tables at the object code level. For implicit type conversions the issue is whether analysis tools will properly understand what is happening in the code emitted by the compiler and linker. This report recommends that every implicit type conversion be analyzed.

The conclusions from this study are:

- OOT impacts structural coverage in two areas:
  - Measuring structural coverage against the source code only may not be sufficient, even for levels B and C software, due to the widening of the abstraction level between source code and object code; and
  - The confirmation of data and control coupling assumes greater importance with OOT, but is problematic.
- Structural coverage should be measured against both concrete and inherited features. Tools should record coverage against the flattened class.
- Structural coverage should be measured against a combination of source code and object code for polymorphism with dynamic binding. Tools should record coverage of the polymorphic reference at the source level and entries of the methods tables at the object code level.
- Each implicit type conversion needs to be analyzed.

The recommendations from this study are:

- Another study should be conducted to look into the widening of the abstraction level between source code and object code in OOT, and whether a combination of source and object code analysis is required.
- Another study should be conducted to look into the proper requirements-based verification of OOT features and language implementations per section 6.4.2 of DO-178B [2].
- Another study should be conducted to look into the proper verification (i.e., confirmation) of data and control coupling for OOT features and language implementations per section 6.4.4.2c of DO-178B [2].

5. REFERENCES.

1. Rierson, L.K., ìObject-Oriented Technology (OOT) in Civil Aviation Projects: Certification Concerns,î Federal Aviation Administration, Washington, D.C.

2. RTCA, DO-178B/ED-12B, ìSoftware Considerations in Airborne Systems and Equipment Certification,î December 1, 1992.

3. RTCA, DO-248B, ìFinal Report for Clarification of DO-178B ëSoftware Considerations in Airborne Systems and Equipment Certification,í î October 12, 2001.

4. The Open Group document, ìOMG Unified Modeling Language Specification,î Version 1.4, September 2001.

5. Binder, R.V., ìTesting Object-Oriented Systems: Models, Patterns, and Tools,î Addison Wesley, Massachusetts, 2000.

6. Page-Jones, M., ìThe Practical Guide to Structured Systems Design,î Yourdon Press, New York, 1980.