

**DOT/FAA/AR-07/19**

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

# **Object-Oriented Technology Verification Phase 2 Handbook— Data Coupling and Control Coupling**

August 2007

Final Report

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



U.S. Department of Transportation  
**Federal Aviation Administration**

## **NOTICE**

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

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

1. Report No. <b>DOT/FAA/AR-07/19</b>		2. Government Accession No.		3. Recipient's Catalog No.	
4. Title and Subtitle <b>OBJECT-ORIENTED TECHNOLOGY VERIFICATION PHASE 2 HANDBOOK—DATA COUPLING AND CONTROL COUPLING</b>				5. Report Date <b>August 2007</b>	
				6. Performing Organization Code	
7. Author(s) <b>John Joseph Chilenski and John L. Kurtz</b>				8. Performing Organization Report No.	
9. Performing Organization Name and Address <b>The Boeing Company P.O. Box 3707 Seattle, WA 98124-2207</b>				10. Work Unit No. (TRAVIS)	
				11. Contract or Grant No.	
12. Sponsoring Agency Name and Address <b>U.S. Department of Transportation Federal Aviation Administration Air Traffic Organization Operations Planning Office of Aviation Research and Development Washington, DC 20591</b>				13. Type of Report and Period Covered <b>Final Handbook – Phase 2</b>	
				14. Sponsoring Agency Code <b>AIR-120</b>	
15. Supplementary Notes <b>The Federal Aviation Administration Airport and Aircraft Safety R&amp;D Division COTR was Charles Kilgore.</b>					
16. Abstract <p>The purpose of this Handbook is to provide guidelines into issues and acceptance criteria for the verification (confirmation) of data coupling and control coupling (DCCC) within object-oriented technology (OOT) in commercial aviation. The intent of the structural coverage analyses (confirmation) of DCCC is to provide an objective assessment (measure) of the completeness of the requirements-based tests of the integrated components. Unfortunately, no measurable adequacy criterion is provided in RTCA DO-178B/EUROCAE ED-12B for Objective 8 of Table A-7. A review of the published literature concerning integration verification found that coverage of intercomponent dependencies as an acceptable adequacy criterion (measure) of integration testing in both non-OOT and OOT software was well motivated. This approach is known as coupling-based integration testing. This Handbook, therefore, employs the coverage of intercomponent dependencies as a measurable adequacy criterion to satisfy DO-178B/EUROCAE ED-12B Table A-7 Objective 8. One limitation of the coverage of intercomponent dependencies used in this Handbook is that any use of polymorphism in the OOT must conform to the Liskov Substitution Principle.</p>					
17. Key Words <b>Object-oriented technology, Data coupling, Control coupling, Verification, Coupling-based integration testing</b>			18. Distribution Statement <b>This document is available to the U.S. public through the National Technical Information Service (NTIS) Springfield, Virginia 22161.</b>		
19. Security Classif. (of this report) <b>Unclassified</b>		20. Security Classif. (of this page) <b>Unclassified</b>		21. No. of Pages <b>30</b>	22. Price

## TABLE OF CONTENTS

	Page
EXECUTIVE SUMMARY	vii
1. INTRODUCTION	1
1.1 Purpose	1
1.2 Background	1
1.3 Document Overview	2
1.4 Related Activities and Documents	2
2. STRUCTURAL COVERAGE OVERVIEW	2
3. COUPLING VERIFICATION	5
3.1 Sequencing	6
3.2 Timing	8
3.3 Control Dependency	9
3.4 Data Dependency	10
4. SUMMARY	12
5. REFERENCES	13
APPENDIX A—DEPENDENCY ANALYSIS OVERVIEW	

## LIST OF FIGURES

Figure		Page
1	Requirements/Implementation Overlap	3
2	Verification and Traceability	5

## LIST OF TABLES

Table		Page
1	Coverage Comparisons	4

## LIST OF ACRONYMS AND ABBREVIATIONS

CAST	Certification Authorities Software Team
CBIT	Coupling-based integration testing
CFG	Control flow graph
DC	Decision coverage
DCCC	Data coupling and control coupling
DUA	Definition-use association
Du-pair	Definition-use pair
FAA	Federal Aviation Administration
LSP	Liskov Substitution Principle
MCDC	Modified condition decision coverage
ms	Milliseconds
NASA	National Aeronautic and Space Administration
OOT	Object-oriented technology
OOTiA	Object-Oriented Technology in Aviation
PDG	Program dependency graph
SDG	System dependency graph
UML	Unified Modeling Language
WCET	Worst-case execution time

## 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 has started being used in the commercial airborne software and systems domain. However, as with any new technology, there are concerns and issues relating to its adoption within safety-critical systems. Previous Federal Aviation Administration (FAA) research and two Object-Oriented Technology 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 is the verification of data coupling and control coupling (DCCC).

This Handbook provides input to industry and the FAA into issues and acceptance criteria for the verification (confirmation) of DCCC within OOT in commercial aviation. This Handbook takes the position that the intent of the structural coverage analyses (confirmation) of DCCC is to provide an objective assessment (measure) of the completeness of the requirements-based tests of the integrated components. Unfortunately, no measurable adequacy criterion is provided in RTCA DO-178B/EUROCAE ED-12B for Objective 8 of Table A-7. A review of the published literature concerning integration verification found that coverage of intercomponent dependencies as an acceptable adequacy criterion (measure) of integration testing, known as coupling-based integration testing, was well motivated for both non-OOT and OOT software. This Handbook, therefore, uses the coverage of intercomponent dependencies as a measurable adequacy criterion to satisfy Objective 8 of DO-178B/EUROCAE ED-12B Table A-7. One limitation of the coverage of intercomponent dependencies used in this Handbook is that any use of polymorphism in the OOT must conform to the Liskov Substitution Principle.

Guidelines for developers, verifiers, and acceptors (generally regulators or their designees) for DCCC verification are provided for four types of dependencies.

- Sequencing dependencies, a part of control coupling, are requirements on the execution order of components.
- Timing dependencies, a part of control coupling, are requirements on the timing of individual components and sequences of multiple components.
- Control flow dependencies, part of control coupling, are represented by control dependencies between components. This is divided into sequencing dependencies and data dependencies within branch points.
- Information flow dependencies, part of data coupling, are represented by data flows between components, where one component defines the value of an object/data item that is used in another component (data dependencies).

One limitation of this Handbook is in the area of polymorphism with dynamic binding and dispatch. Defining adequate verification for this OOT feature is an active research area without a definitive answer. As such, the recommendation in this Handbook may only be considered an interim solution where polymorphism with dynamic binding and dispatch is concerned.

## 1. INTRODUCTION.

### 1.1 PURPOSE.

This Handbook is intended to provide input to industry and the Federal Aviation Administration (FAA) into issues and acceptance criteria for the verification (confirmation) of data coupling and control coupling (DCCC) within object-oriented technology (OOT) in commercial aviation. This Handbook takes the position that the intent of the structural coverage analyses (confirmation) of DCCC is to provide an objective assessment (measure) of the completeness of the requirements-based tests of the integrated components. A review of the published literature concerning integration verification found that coverage of intercomponent dependencies as an acceptable adequacy criterion (measure) of integration testing in both non-OOT and OOT software was well motivated. This approach is known as coupling-based integration testing (CBIT). This Handbook, therefore, uses the coverage of intercomponent dependencies as a measurable adequacy criterion to satisfy Objective 8 of RTCA DO-178B/EUROCAE ED-12B Table A-7 (DO-178B hereafter) [1]<sup>1</sup>.

Note that the analysis performed in this Handbook assumes subtype inheritance that conforms to the Liskov Substitution Principle (LSP) [2]. This means that a subclass must accept all messages that its superclass will accept, and it must produce appropriate results. As a result, “subclass objects can be substituted for superclass objects without causing failures or requiring special case code in clients” [3]. This ensures that “the objects of the subtype ought to behave the same as those of the supertype as far as anyone or any program using supertype objects can tell” [2]. Defining adequate verification for polymorphism with dynamic binding and dispatch is an active research area without a definitive answer. As such, the coverage of intercomponent dependencies used in this Handbook may only be considered an interim solution where polymorphism with dynamic binding and dispatch is concerned.

### 1.2 BACKGROUND.

DO-178B requires the confirmation of DCCC in Objective 8 of Table A-7 [1]. Unfortunately, no objective (measurable) adequacy criterion is given for this objective. This is in sharp contrast to the other structural coverage objectives (5-7) in the same table.

OOT has been used extensively throughout the non-safety-critical software and computer-based systems industry in safety-critical medical and automotive systems and has started being used in the commercial airborne software and systems domain [4 and 5]. Previous FAA research [4, 5, and 6] and two Object-Oriented Technology in Aviation (OOTiA) workshops with industry (see <http://shemesh.larc.nasa.gov/foot/> for more information) indicate that guidance for the application of DCCC 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 the confirmation of DCCC (satisfaction of Objective 8 of

---

<sup>1</sup> Note: Coverage of intercomponent dependencies (CBIT) will satisfy Objective 8 of DO-178B Table A-7 for both OOT and non-OOT software.

DO-178B Table A-7 [1]) within OOTiA. This Handbook is a companion document to the research report [7] on DCCC verification.

### 1.3 DOCUMENT OVERVIEW.

As stated, this Handbook is a companion document to the research report [7]. The research report contains the details behind the steps employed in this Handbook. This Handbook contains the practical how-to guidelines for performing DCCC verification.

- Section 1 provides the purpose, background, and general overview.
- Section 2 provides an overview of structural coverage within DO-178B and the role of DCCC verification [1].
- Section 3 provides the guidelines on how to perform the DCCC verification (structural coverage analysis).
- Section 4 summarizes the approach.
- Section 5 provides a list of references.
- Appendix A provides a brief overview of dependency analysis.

### 1.4 RELATED ACTIVITIES AND DOCUMENTS.

There are four related activities (one workshop, two previous studies, and one Certification Authorities Software Team (CAST) paper) and their associated documents and the companion DCCC research report that relate directly to the issues addressed herein:

- The joint FAA/National Aeronautics and Space Administration (NASA) OOTiA project workshops and the associated documentation at <http://shemesh.larc.nasa.gov/foot/>.
- “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/](http://faa.gov/aircraft/air_cert/design_approvals/air_software/oot/).
- Certification Authorities Software Team (CAST), Position Paper CAST-19, “Clarification of Structural Coverage Analyses of Data Coupling and Control Coupling,” Completed January 2004, (Rev 2), available at: [http://faa.gov/aircraft/air\\_cert/design\\_approvals/air\\_software/cast/cast\\_papers/](http://faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/).

## 2. STRUCTURAL COVERAGE OVERVIEW.

Software verification consists of a combination of reviews, analyses, and tests to ensure that the software satisfies its requirements and that errors, which could lead to unacceptable failure conditions have been removed [1]. Testing demonstrates the requirements compliance and provides high confidence that errors have been removed through execution of the software [1].

The adequacy of the testing is assessed by coverage analysis of the requirements and the code structure [1]. The need for both forms of coverage analysis is illustrated in the requirements versus implementation overlap depicted in figure 1.

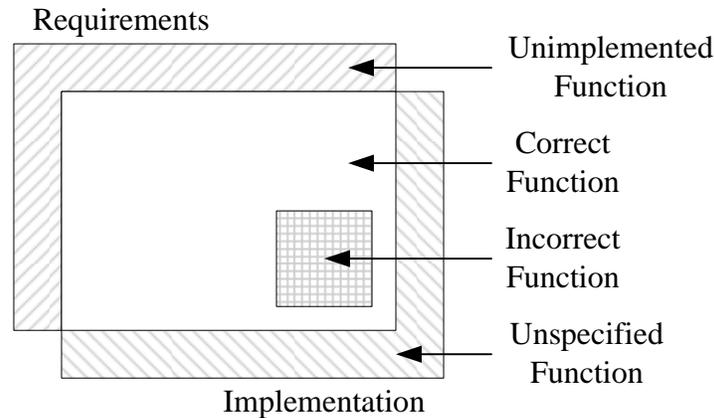


Figure 1. Requirements/Implementation Overlap

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 function) and parts where it is not (i.e., incorrect function). Requirements-based tests will generally operate within this overlap. Where the requirements do not have an overlap with the implementation is where the implementation fails to implement a requirement (i.e., unimplemented function). Where the implementation does not have an overlap with the requirements is where the implementation provides a capability beyond the requirements (i.e., unspecified function, possibly unintended).

Requirements coverage is used to ensure that the test cases satisfy the specified criteria [1]. This helps ensure that correct function is present, that incorrect function (i.e., error) is detected if present, and that no unimplemented function is present (i.e., no required function is missing). Requirements coverage will generally not catch unspecified function. Structural coverage is used to ensure that the requirements-based testing, as a whole, adequately exercised the code structure. This helps ensure that no unspecified function is present. Structural coverage will generally not catch unimplemented function.

The intent of the structural coverage analyses (confirmation) of DCCC is to provide an objective assessment (measure) of the completeness of the requirements-based tests of the integrated components [7]. This means that DCCC verification helps to ensure the demonstration of the presence of intended interactions (function) between those components and supports the demonstration of the absence of unintended interactions (function) between those components. Since DCCC is a structural coverage criterion, requirements-based tests covering these interactions should be present.

As defined in this Handbook, achieving DCCC coverage will have overlap with the other structural coverage criteria in Objectives 5 through 7 of Table A-7 of DO-178B [1 and 7]:

- Executing all statements for the unconditional control dependency edges and those conditional control dependency edges leading to statements (statement coverage, partial decision coverage (DC), and partial Modified Condition Decision Coverage (MCDC)).
- Executing those branch point outcomes that lead to statements for the conditional control dependency edges (partial DC, partial MCDC). Note that not all branches of a branch point need be represented in a program dependency graph (PDG) [7].
- Executing all executable definition-use pairs (du-pairs) for the data dependency edges. This leads to execution of all statements containing definitions and uses of objects (partial statement coverage, partial MCDC). This may lead to executing branch point outcomes that do not have a corresponding PDG control dependency edge, but form a definition-clear subpath (partial DC, partial MCDC).
- Executing all subprogram calls for both forms (control and data) of dependencies (a by-product of MCDC, therefore partial MCDC).

Table 1 compares the levels of coverage identified in Table A-7 of DO-178B [1] along with those of DCCC verification employed in this Handbook beyond that currently in DO-178B [1]. In table 1, the first column identifies the four levels of coverage. Note that only the branch point/branches portion of DC are being considered. The second through fifth columns show which of the coverage criteria are required by software Levels A, B, and C, and DCCC, respectively, where a particular coverage criterion that is satisfied by a software level or DCCC is indicated by an “X” in the row/column intersection. Note that covering DCCC will require coverage beyond that required for all software levels. Note also that covering DCCC does not satisfy all coverage required for Levels A and B software, because it only provides partial DC and MCDC.

Table 1. Coverage Comparisons

Type of Coverage	Level A	Level B	Level C	DCCC
Statement coverage	X	X	X	X
Decision (branch point) coverage	X	X		
MCDC	X			
Coverage of calls	X*			X
Coverage of du-pairs				X

\*Byproduct of MCDC

Structural coverage is most effective when it is obtained as a by-product of requirements-based testing. This Handbook assumes that all tests built to cover DCCC interactions will come from some combination of the system requirements, software high-level requirements, software low-level requirements, and software architecture because those sources define the need for the DCCC interaction.

### 3. COUPLING VERIFICATION.

Determining the adequacy of the DCCC verification consists of three steps.

Step 1 identifies the DCCC dependencies between components. In OOT, components are class methods. In this Handbook, there are four classes of dependencies identified. One class is subsumed by the analysis required by this Handbook for the other two, but is kept for consistency with the companion report [7] and the general literature. Each dependency is covered within one of the following four sections:

- Sequencing
- Timing
- Control dependencies
- Data dependencies

The requirement for these dependencies should be documented in the development artifacts: system requirements, software high-level requirements, software low-level requirements, and software architecture. The implementation of these requirements should be present in the implementation artifacts: source code and object code.

Step 2 identifies the verification methods for each dependency.

- Reviews
- Analyses
- Tests
- Traceability

Verification will consist of some combination of the above methods. The first three methods are the traditional verification methods. Traceability is needed between the development artifacts, the implementation artifacts, and the verification artifacts as depicted in figure 2 [1]. Because of the importance of traceability, it has been included as a necessary part of verification in the above list.

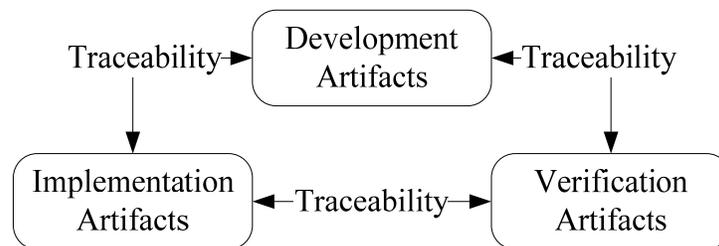


Figure 2. Verification and Traceability

Step 3 provides or identifies the justification for the absence of verification if there are no verification methods that cover a dependency. As DCCC is a structural coverage criterion, and therefore requires execution, the absence of testing is a special concern.

There are three categories of personnel that perform responsibilities within these steps.

- Developer—this group of individuals is collectively responsible for documenting the requirement for the DCCC dependency in the development artifacts, developing that requirement into an implementation, and establishing the traceability between the development artifacts and the implementation artifacts (generally the source code).
- Verifier—this group of individuals is collectively responsible for developing the verification of the DCCC dependency and for establishing the traceability between verification artifacts, development artifacts, and implementation artifacts. It is envisioned that, in general, no specific tests for DCCC structural coverage will need to be generated, because other requirements-based tests (operational scenarios) will exercise the DCCC dependencies as by-products. These tests merely need to be tagged as covering the DCCC dependencies (traceability). If it is determined that no testing covering a dependency is needed, then this group is collectively responsible for justifying why execution of a dependency by a requirements-based test (an operational scenario) is not necessary. Beizer points out that untested code is generally not to be trusted and should not be incorporated into an operational system [8].
- Reviewer/Acceptor—this group, generally a regulator or representative, is responsible for ensuring that each step has been adequately performed.

Discussion for each of the four DCCC dependencies follows in sections 3.1 through 3.4.

### 3.1 SEQUENCING.

Sequencing dependencies, which are a part of control coupling, are requirements on the execution order of components: for example, component  $C_1$  must execute before component  $C_2$ , component  $C_3$  calls component  $C_4$ , etc. Sequencing requirements are due to some form of dependency between the components. Earlier components perform functions necessary for later components. One concrete example in OOT is that the constructor for an object must be called before any other methods are called for that object.

A number of representations, both diagrammatic and textual, can be used to both express these sequencing requirements and to document the implementation. Commercial tools exist to generate these diagrams or text from user input during forward engineering (development), as well as reverse engineer these diagrams, or text, from existing implementations. Representations include:

- Call trees (also known as call graphs)
- Compiler tables (link directives)
- Data-flow diagrams
- Linker tables (link map)
- Structure charts

- Unified Modeling Language (UML) Activity (Interaction) Diagrams
- UML collaboration (interaction) diagrams
- UML communication diagrams
- UML sequence (interaction) diagrams
- Others

A number of mechanisms can be used to implement the transfers of control between components necessary to accomplish the ordering. Note that the mechanisms are dependent on the programming language, run time support, and hardware being used. Mechanisms include:

- Subprogram call/return
- Raising and handling of exceptions
- Jumps
- Task rendezvous
- Interrupts
- Others

The steps to be taken concerning the sequencing portion of DCCC verification are:

- Step 1: Identify all sequencing requirements and design in the development artifacts. It is anticipated that the majority of this information will reside in the software architecture. The developers are responsible for the initial production of this information. The verifiers are responsible for locating this information.
- Step 2: Identify all instances of sequencing/transfers of control between components in the implementation artifacts. It is anticipated that the majority of this information will reside in the source code. It is also anticipated that the majority of these instances will be due to subprogram call/return. The developers are responsible for the initial production of this information. The verifiers are responsible for locating this information. It is anticipated that the verifiers will use tools to help with the identification.

Component referencing can be direct, generally by name, or indirect, generally through an alias or a pointer. When indirection is used, part of the identification process will entail determining all references that can occur as a result of the instance (i.e., alias analysis).

- Step 3: Check the traceability between the development artifacts and the implementation artifacts. The developers are responsible for the initial production of this information. The verifiers are responsible for locating this information, checking it for completeness and correctness, and identifying errors.
- Step 4: Check the conformance of the implementation artifacts to the development artifacts. All instances of sequencing/transfers of control between components in the implementation artifacts (generally source code) will need to be checked for proper occurrence (i.e., do they occur as designed) and operation (i.e., do they behave as

designed, and do they behave as required). The verifiers are responsible for preparing the verification artifacts; establishing the traceability between the verification artifacts, development artifacts, and implementation artifacts per figure 2; and identifying errors.

This will generally entail a series of requirements-based tests executing each instance of sequencing/transfers of control between components. Note that an instance may only need a single execution (i.e., a series can consist of a single element), and a test may execute several instances.

If a transfer of control is due to a polymorphic reference or any other dynamic multiple-bound mechanism, multiple tests resolving to different bindings may be required [7]. Recall that if a polymorphic reference is present, it must conform to LSP. Also recall that verification beyond this may be required for polymorphism with dynamic binding.

- Step 5: If there are no verification methods that cover a sequencing dependency, generally a transfer of control between components will provide or identify the justification for the absence of verification.

### 3.2 TIMING.

Timing dependencies, which is a part of control coupling, are requirements on the timing of individual components and sequences of multiple components; e.g., component  $C_1$  must execute 37 milliseconds (ms) before component  $C_2$ , component  $C_4$  must complete execution within 25 ms after being called by component  $C_3$ , and so on. Timing dependencies include such things as component performance, throughput, and rates. Timing requirements are generally due to the necessary interactions between the system and the rest of the world.

Diagrammatic and textual representations can be used to both express these timing requirements and to document the implementation. Commercial tools exist to generate these diagrams, or text, from user input during forward engineering (development), as well as reverse engineer these diagrams, or text, from existing implementations. Representations include:

- Temporal logics (generally textual, but some have diagrammatic representations)
- Timing diagrams (both UML and others)
- Timing traces (from timing analyzers and other execution profilers)
- Others

The steps to be taken concerning the timing portion of DCCC verification are:

- Step 1: Identify all timing requirements and constraints in the development artifacts. It is anticipated that the majority of this information will reside in the software architecture. However, embedded real-time systems must generally interact with the real world, and the resulting requirements on system performance will come from the system requirements. The developers are responsible for the initial production of this information. The verifiers are responsible for locating this information.

- Step 2: Identify all instances of component and intercomponent timing constraints in the implementation artifacts. It is anticipated that the majority of this information will reside in the source code, generally as scheduling constructs. The developers are responsible for the initial production of this information. The verifiers are responsible for locating this information. It is anticipated that the verifiers will use tools to help with the identification.
- Step 3: Check the traceability between the development artifacts and the implementation artifacts. The developers are responsible for the initial production of this information. The verifiers are responsible for locating this information, checking it for completeness and correctness, and identifying errors.
- Step 4: Check the conformance of the implementation artifacts to the development artifacts. All instances of both component and intercomponent timing constraints in the implementation artifacts (generally executable object code) will need to be checked for proper operation (i.e., do they behave as designed, do they behave as required). The verifiers are responsible for preparing the verification artifacts; establishing the traceability between the verification artifacts, development artifacts, and implementation artifacts per figure 2; and identifying errors. It is anticipated that the verifiers will use tools to help with the identification, particularly timing analyzers and profilers. It is also anticipated that the majority of this effort will be common with the worst-case execution time (WCET) analysis.
- Step 5: If there are no verification methods that cover a timing dependency (generally a WCET), then provide or identify the justification for the absence of verification.

### 3.3 CONTROL DEPENDENCY.

Control flow dependencies, which are part of control coupling, are represented by control dependencies between components. These dependencies take two forms:

- Transfers of control between components, generally due to subprogram call/return. This form of control dependency is covered by the sequencing discussion in section 3.1.
- Definitions of objects/data items in one component used in the branch point selections of another component. This form of data dependency is covered by the data dependency discussion in section 3.4. Note that in the companion report to this Handbook it was pointed out that this form of data dependency could be handled as any other data dependency [7].

This form of dependency has been included for consistency with the companion report [7] and the published literature on the topic even though it is subsumed by the analyses required by other dependency forms in this Handbook.

### 3.4 DATA DEPENDENCY.

Information flow dependencies, which are part of data coupling, are represented by data flows between components where one component defines the value of an object/data item that is used in another component (data dependencies); e.g., the AirSpeed function calculates CurrentAirSpeed and then sends CurrentAirSpeed to the PrimaryFlightDisplay.

A number of representations, both diagrammatic and textual, can be used to both express these information flow requirements and to document the implementation. Commercial tools exist to generate these diagrams, or text, from user input during forward engineering (development), as well as reverse engineering these diagrams, or text, from existing implementations. Representations include:

- Call trees (also known as call graphs), especially when annotated with information flow
- Code analyzers (e.g., CodeSurfer<sup>®</sup> from GrammaTech)
- Compiler tables
- Cross-reference reports/tables (generated by compiler or commercial tool)
- Data dependency graph
- Data dictionaries
- Data-flow diagrams
- Definition-use graphs
- Definition-use matrices
- Interactive development environments
- Program-dependency graphs
- Set-use tables
- System-dependency graphs
- Structure charts
- UML activity (interaction) diagrams
- UML collaboration (interaction) diagrams
- UML communication diagrams
- UML class diagrams
- UML sequence (interaction) diagrams
- Others

Steps to be taken concerning the information flow (data dependency) portion of DCCC verification are:

- Step 1: Identify all information flow (data dependency) requirements and design in the development artifacts. It is anticipated that the majority of this information will reside in the software architecture. The developers are responsible for the initial production of this information. The verifiers are responsible for locating this information.

- Step 2: Identify all instances of information flow (data dependency) between components in the implementation artifacts. It is anticipated that the majority of this information will reside in the source code. The developers are responsible for the initial production of this information. The verifiers are responsible for locating this information. It is anticipated that the verifiers will use tools to help with the identification.

As identified in reference 7, the essence of the data dependency analysis is to identify all data items accessed by more than one component. In OOT, this means identifying all accesses to objects and attributes within methods. Object and attribute referencing can be direct (generally by name) or indirect (generally through an alias or a pointer). When indirection is used, part of the identification process will entail determining all references that can occur as a result of the instance (i.e., alias analysis).

The essence of this analysis is to identify all the realizable intercomponent du-pairs for each object and attribute in the software [7].

- Identify the objects and attributes present in the system. To simplify the analysis of composite data types, an array can be treated as a single entity, while records have each component (field) treated individually [9].
  - For each object and attribute, identify each method that defines a value for the object/attribute. This will form half of an intercomponent du-pair.
  - For each object and attribute, identify each method that uses the value of the object/attribute. This will form half of an intercomponent du-pair.
  - For each potential unique intercomponent du-pair (defining and using methods are different), determine if the intercomponent du-pair is realizable (i.e., executable under some operational scenario for the system). Two intercomponent du-pairs are considered to be duplicates if the corresponding definition-use associations (DUA) are the same [7].
- Step 3: Check the traceability between the development artifacts and the implementation artifacts. The developers are responsible for the initial production of this information. The verifiers are responsible for locating this information, checking it for completeness and correctness, and identifying errors.
  - Step 4: Check the conformance of the implementation artifacts to the development artifacts. All instances of information flow (data dependency) between components in the implementation artifacts (generally source code) will need to be checked for proper occurrence (i.e., do they occur as designed) and operation (i.e., do they behave as designed, do they behave as required). Proper operation also includes such things as:
    - Same size and structure in both the definition and the use? If not, are they compatible? If not, are they as required?

- Same meaning (semantics) behind the data? If not, are they compatible? If not, are they as required?
- Does the du-pair exist under the correct circumstances (i.e., is the DUA correct)? This analysis is especially important when the use has multiple definers [7].
- If the use appears in a branch point predicate, does it make the correct contribution to the determination of control flow? This is the special form of data dependency that is considered by some as a control dependency [7]. For Level A software, MCDC helps to ensure this. In general, ensuring the proper contribution of a use is a normal part of checking proper operation. As such, this form of dependency can be handled as any other data dependency [7].

The verifiers are responsible for preparing the verification artifacts; establishing the traceability between the verification artifacts, development artifacts and implementation artifacts per figure 2; and identifying errors.

- This will generally entail a series of existing requirements-based tests executing each instance of information flow (data dependency) between components. Note that an instance may only need a single execution (i.e., a series can consist of a single element), and a test may execute several instances. It is envisioned that this effort will mainly be a traceability exercise, since required information flow will have been covered by operational scenarios present in existing requirements-based tests (i.e., DCCC coverage is a by-product of requirements-based tests).
  - If an information flow (data dependency) is due to a polymorphic reference or any other dynamic multiple-bound mechanism, multiple tests resolving to different bindings may be required [7].
- Step 5: If there are no verification methods that cover an information flow (data dependency) dependency, provide or identify the justification for the absence of verification.

#### 4. SUMMARY.

This Handbook provides guidelines for developers, verifiers and acceptors (generally regulators or their designees) for the verification (confirmation) of DCCC within OOT in commercial aviation as required by Objective 8 of Table A-7 in DO-178B [1]. The intent of the structural coverage analyses (confirmation) of DCCC is to provide an objective assessment (measure) of the completeness of the requirements-based tests of the integrated components (i.e., objectively measure integration testing). Currently, DO-178B [1] does not impose an objective measure for the confirmation of DCCC between the code components called out in Section 6.4.4.3c of DO-178B [1]. This Handbook employs the coverage of intercomponent dependencies (du-pairs and calls) as that objective measure.

Dependency analysis is well-established within the computer science and software engineering disciplines. Current compilers perform this analysis in support of optimization, and commercial

tools performing this analysis in support of a number of activities (e.g., maintenance, change-impact analysis, reverse engineering) are currently available. The use of dependency analysis as an adequacy criterion for testing, particularly integration testing, is also well-motivated in both the non-OOT and OOT testing literature and is known as CBIT.

Guidelines for developers, verifiers, and acceptors (generally regulators or their designees) for DCCC verification are provided for four types of dependencies:

- Sequencing dependencies, which are a part of control coupling, are requirements on the execution order of components.
- Timing dependencies, which are a part of control coupling, are requirements on the timing of individual components and sequences of multiple components.
- Control flow dependencies, which are part of control coupling, are represented by control dependencies between components. This is divided into sequencing dependencies and data dependencies within branch points.
- Information flow dependencies, which are part of data coupling, are represented by data flows between components where one component defines the value of an object/data item that is used in another component (data dependencies).

One limitation of this Handbook is in the area of polymorphism with dynamic binding and dispatch. Defining adequate verification for this OOT feature is an active research area with no definitive answer yet. As such, the coverage of intercomponent dependencies to satisfy DCCC verification may only be considered an interim solution where polymorphism with dynamic binding and dispatch is concerned [7].

## 5. REFERENCES.

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

1. “Software Considerations in Airborne Systems and Equipment Certification,” Document No. RTCA/DO-178B, RTCA Inc., December 1, 1992.
2. Liskov, B.H. and Wing, J.M., “A Behavioral Notion of Subtyping,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 16, No. 6, November 1994, pp. 1811-1841.
3. Binder, R.V., “Testing Object-Oriented Systems: Models, Patterns, and Tools,” Addison-Wesley, 2000.
4. Chilenski, J.J., Heck, D., Hunt, R., and Philippon, D., “Object-Oriented Technology (OOT) Verification Phase 1 Report—Survey Results,” The Boeing Company.

5. Knickerbocker, J., “Object-Oriented Software—Object-Oriented Technology in Aviation (OOTiA) Survey,” a presentation to the 2005 FAA Software/CEH Conference, July 2005.
6. Chilenski, J.J., Timberlake, T.C., and Masalskis, J.M., “Issues Concerning the Structural Coverage of Object-Oriented Software,” FAA report DOT/FAA/AR-02/113, November 2002.
7. Chilenski, J.J. and Kurtz, J.L., “Object-Oriented Technology Verification Phase 2 Report—Data Coupling and Control Coupling,” FAA report DOT/FAA/AR-07/52, August 2007.
8. Beizer, B., “Software Testing Techniques,” 2<sup>nd</sup> ed., Van Nostrand Reinhold, 1990.
9. Barnes, J., “High Integrity Software: The SPARK Approach to Safety and Security,” Addison-Wesley, 2003.

## APPENDIX A—DEPENDENCY ANALYSIS OVERVIEW

This appendix contains a brief overview of dependency analysis. More detailed information may be found in the companion report [A-1].

A dependency exists between two components,  $C_1$  and  $C_2$ , when statements in component  $C_1$  influence the execution of statements in component  $C_2$ . In particular, for integration verification, faults in the statements of component  $C_1$  affect the execution of statements in component  $C_2$  [A-2].

Dependency relations require an interface between the components. This interface can be direct between call-pairs (e.g., component  $C_1$  calls component  $C_2$ , and both components access object A), or indirect (e.g., component  $C_1$  calls components  $C_2$  and  $C_3$  at different times, either directly or indirectly, and both  $C_2$  and  $C_3$  access object A). Dependency relations are also known as coupling [A-2]. Many types of dependency relations exist [A-3]:

- Syntactic—one statement affects the execution of another based on the syntax of the programming language used.
- Semantic—one statement affects the execution behavior of another during execution. A semantic dependency always implies a syntactic one (i.e., a semantic dependency is always allowed by the syntax of the programming language used). A semantic dependency can be executed dynamically (i.e., witnessed) during some operational scenario of the system containing the software.
- Control—one statement controls the execution of another statement. This may be either syntactic or semantic. In figure A-1, statements 4 and 5 are control dependent on statement 3, since the outcome of statement 3 determines which statement will be executed. All the statements in figure A-1 are control dependent on statements in other components that call the function `Maximum_Of`, since none of the statements will execute unless a call is made.
- Data—one statement defines the value of an object used in another statement. This may be either syntactic or semantic. In figure A-1, the statements labeled 3 and 4 are data dependent on the statement labeled 1 because X is defined in the statement labeled 1 and used in the statements labeled 3 and 4. Likewise, the statements labeled 3 and 5 are data dependent on the statement labeled 2 because of Y.

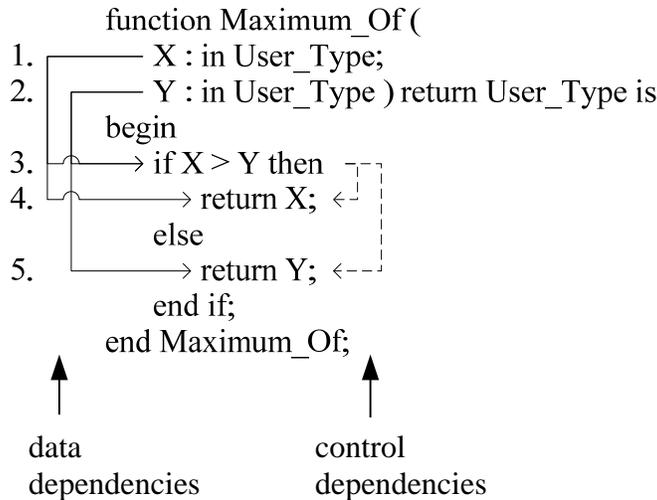


Figure A-1. Maximum\_Of and Dependencies

Dependencies are generally represented in a program dependency graph (PDG) [A-4 and A-5]. Figure A-2 shows the source code for function Maximum\_Of from figure A-1 on the left along with the corresponding PDG on the right. A PDG is a directed graph consisting of (1) a set of nodes (N) representing either the entry points or computations within the design or code and (2) a set of directed edges (E), expressed as ordered node pairs ( $n_{source}, n_{destination}$ ), representing either control dependencies or data dependencies. The directed edge flows from the source node to the destination node. In figure A-2, the entry node carries the name of the function.

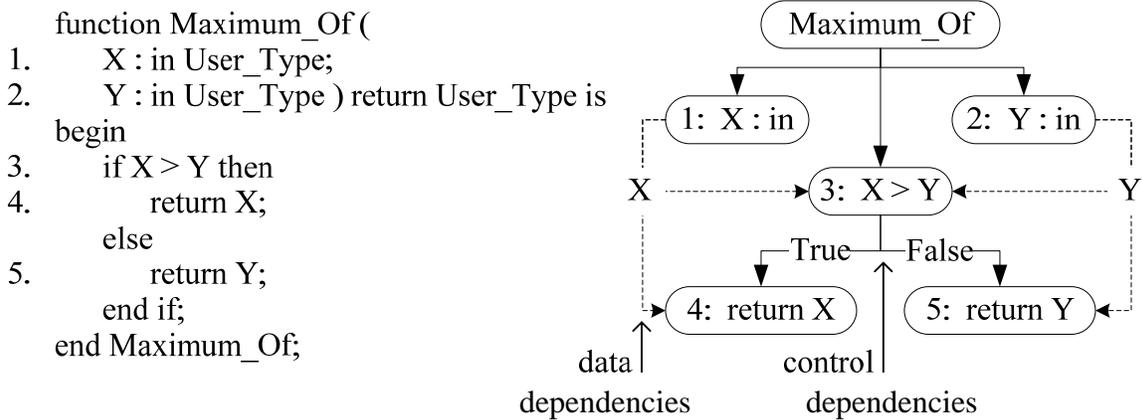


Figure A-2. Maximum\_Of Program Dependency Graph

A control dependency edge flows between two nodes and is labeled with the predicate outcome if traversal of the edge is conditional on some predicate in the source node. In figure A-2, there are unconditional control dependency edges between the entry and statements/nodes 1, 2, and 3. This means that if function Maximum\_Of is called and entered, these statements/nodes will be executed as long as no exceptions are raised. There are conditional control dependency edges between statement/node 3 and statements/nodes 4 and 5. If the predicate in statement/node 3 is True, then statement/node 4 will be executed, otherwise statement/node 5 will be executed. One

significant difference between a control flow graph (CFG), or flow chart, and a PDG is that in a PDG not all outcomes of a decision statement (branch point) need be represented with a control dependency edge [A-4 and A-5]. Figure A-3 depicts an example CFG on the left and the corresponding PDG with control dependency edges only on the right where not all outcomes of branch points in the CFG are represented with control dependency edges in the PDG. Note that statement/node 4 is control dependent on statement/node 1 being True, since statement/node 4 is reached when statement/node 2 is both True and False.

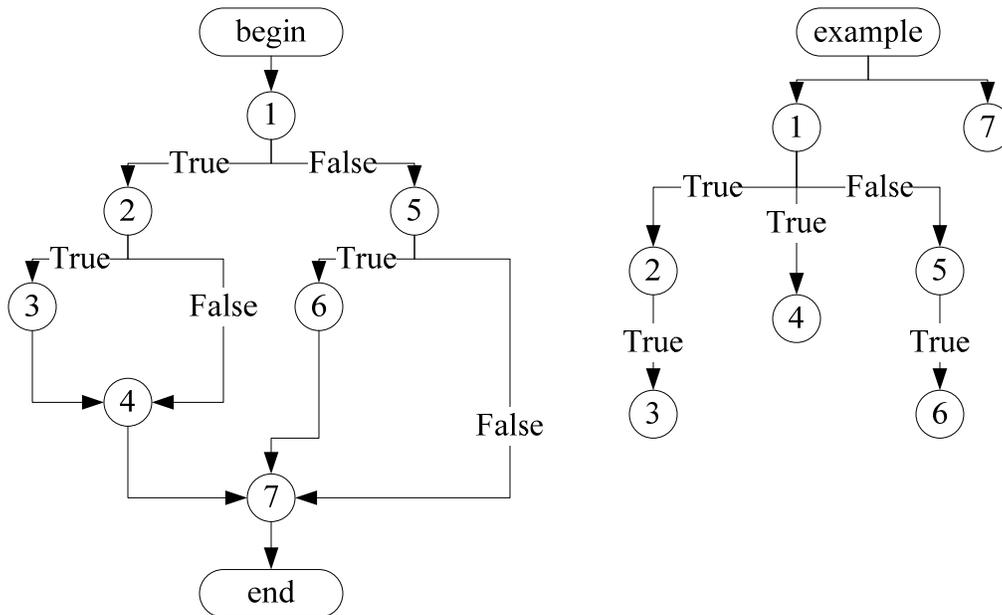


Figure A-3. The CFG PDG Comparison

A data dependency edge flows between two nodes where the source node defines the value of some object that is used in the destination node and is labeled with the name of the object. In figure A-2, there are data dependency edges between statement/node 1 and statements/nodes 3 and 4. These edges are labeled X since X receives a value in statement/node 1 that is used in statements/nodes 3 and 4. A similar data dependency for Y is shown for statement/node 2 and statements/nodes 3 and 5.

A data dependency edge that flows into a branching predicate is considered by some to be a part of the control dependency of the branched-to statements/nodes. In figure A-2, the data dependency edges between statements/nodes 1 and 3 for X, and statements/nodes 2 and 3 for Y, are examples of this form of special data dependency. This makes sense, because the execution of statements/nodes 4 and 5 is dependent on the values of X and Y in statement/node 3, which were defined in statements/nodes 1 and 3, respectively. This is also in conformance with the definition for control coupling in RTCA DO-178B [A-6] since the statement (component) defining the data is influencing the execution of the other statement (component).

Data dependencies are generally represented with a definition-use pair (du-pair) [A-7]. A du-pair is a pair of statement/node numbers that represent where an object is defined and where that

definition can be used. For the definition to be used, at least one subpath (sequence of statements) between the definition and use must be definition clear (i.e., not contain another definition of the object). The two intracomponent du-pairs for X in figure A-2 are (1,3) and (1,4), respectively, while those for Y are (2,3) and (2,5).

A du-pair can be extended into a definition-use association (DUA) by adding a predicate that defines when the du-pair exists. Recall that for a du-pair to exist, there must be at least one definition-clear subpath between the definition and the use. This means that the predicate in the DUA defines all the definition-clear subpaths for the du-pair. For the du-pair (1,3) for X in figure A-2, this predicate would be that the function Maximum\_Of is entered without an exception being raised. This would make the DUA (1,3,Enter(Maximum\_Of)), where the function “Enter” means that the identified subprogram must be entered without an exception being raised. For the du-pair (1,4) for X in figure A-2, the predicate would be that the function Maximum\_Of be entered without an exception and that the value of X at statement/node 3 be greater than the value of Y at statement/node 3. Since the values of X and Y at statement/node 3 are just the values on entry, the DUA would be (1,4, Enter(Maximum\_Of) and (X > Y)).

A syntactic dependency can be identified with a static analysis of the code. The analysis need only identify where objects are defined and where those same objects are used where a definition-clear control-flow path exists between the definition and the use. Figure A-4 on the left-hand side shows the syntactic dependencies for object A. Object A has definitions and uses on all lines. Note that there are definition-clear paths from both lines labeled 1 and 2 to both lines labeled 3 and 4, which leads to the following four syntactic intracomponent du-pairs for A: (1,3), (1,4), (2,3), and (2,4).

However, a syntactic dependency may not be realizable with execution data (i.e., there is no data that would cause that dependency to be executed). Semantic dependency overcomes this problem by requiring that the dependency be executable. Figure A-4 shows the difference between syntactic dependencies on the left and semantic dependencies on the right of the code.

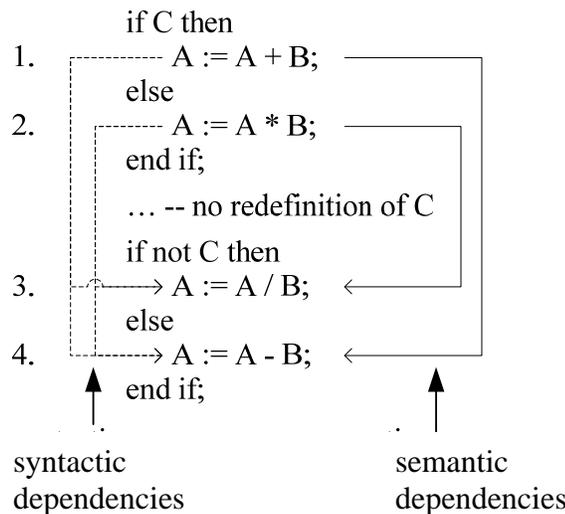


Figure A-4. Syntactic vs Semantic Dependencies

Since there is no redefinition of *C* between the two if-statements, when *C* is True, the statements labeled 1 and 4 will be executed, and when *C* is False, the statements labeled 2 and 3 will be executed. This means the only realizable dependencies and intracomponent du-pairs for *A* are: (1,4) and (2,3).

Recall that semantic dependency requires that one statement affects the execution behavior of another. This is demonstrated by either modifying one statement, or changing the value of one statement, to see if the execution of the other statement is changed [A-8 and A-9]. In figure A-4, if the statement labeled 1 is changed from a sum to a difference (i.e., changed from  $A := A + B$  to  $A := A - B$ ), then for all nonzero values of *B*, the result at the statement labeled 4 will differ between the two executions. This establishes that there is a semantic dependency between the statements labeled 1 and 4. A corresponding analysis can be done to establish the semantic dependency between the statements labeled 2 and 3.

Unfortunately, determining if a dependency is semantic is generally undecidable [A-3]. However, it has been shown that reachability over the PDG, that is, finding program data that will execute (witness) the dependency represented by the PDG edge (i.e., cover the PDG edge), is a conservative approximation of semantic dependency [A-3]. That is, semantic dependency implies PDG reachability, but PDG reachability does not imply semantic dependency.

For integration verification, there is a concern with relations between components. The definitions and examples given previously in this appendix are intracomponent relations in that each of the statements resides in the same component. To extend dependence relations from intracomponent to intercomponent usage, one of the statements must be in one component, while the other statement resides in a different component.

Note that reference to the object can be direct, generally by name, or indirect through an alias. An alias is an object name that can refer to other objects. The other objects do not need to have the same name, but generally need to have the same or compatible type. Examples of aliases are the formal parameter names for the actual parameters and pointers. Read-only formal parameters, also known as pass-by-copy or in-mode, will represent uses of the actual parameters. Write-only formal parameters, also known as out-mode, will represent definitions of the actual parameters. In some languages (e.g., Ada95), these parameters are allowed to be read after an initial assignment, in which case these reads will represent uses of the actual parameter. Read-write formal parameters, also known as pass-by reference or in-out-mode, will represent both definitions and uses of the actual parameters. Pointers are equivalent to read-write parameters.

The previously mentioned extension to intercomponent dependencies means that, for a particular object, there will be a definition of that object in one component and the usage of that object definition in another. The object will now have an intercomponent du-pair. As was mentioned previously, the two components can be involved in a direct call or invocation (e.g., tasking, interrupt) relationship (e.g., one calls the other, one rendezvous with the other) or an indirect one. Figure A-5 gives some examples of the cross component dependencies.

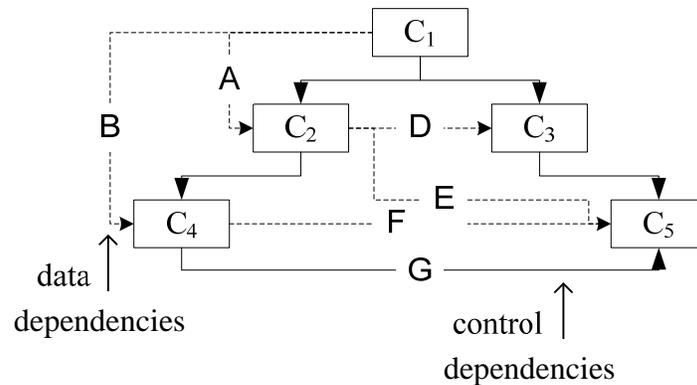


Figure A-5. Intercomponent Dependencies

Within figure A-5, components  $C_2$  and  $C_3$  are shown as control dependent on  $C_1$ , as  $C_1$  calls  $C_2$  and  $C_3$ . Similar control dependencies exist between components  $C_2/C_4$  and  $C_3/C_5$ . There is also a control dependency between components  $C_4$  and  $C_5$  due to a shared object named  $G$ . This is an example of the special form of data dependency mentioned previously where  $G$  receives a definition in component  $C_4$ , and that definition is then used in the predicate of a branch point in component  $C_5$ . During the study for the companion report [A-1], it was determined that handling this special form of data dependency, as any other (i.e., normal) data dependency, does not appear to compromise the verification in any way, and is illustrated for completeness.

Several forms of normal data dependency are also shown in figure A-5. Data dependency between direct call-pairs is shown by object  $A$  between components  $C_1/C_2$ , and between indirect call-pairs by object  $B$  between components  $C_1/C_4$  (indirect because  $C_1$  calls  $C_2$ , which in turn calls  $C_4$ ). Data dependencies between components that are not part of the same calling subtree are shown by object  $D$  between components  $C_2/C_3$ , object  $E$  between components  $C_2/C_5$  and object  $F$  between components  $C_4/C_5$ .

Figure A-5 represents a simplified view of the dependencies between components. PDGs are extended to system dependency graphs (SDGs) to model multicomponent systems (i.e., programs with subprograms, generally known as procedures and functions) [A-5]. An SDG consists of a collection of individual PDGs connected together with additional nodes representing calls/returns and additional dependency edges based on the relationships that cross calls/returns. Strictly speaking, an SDG is a PDG for a larger component. In figure A-5, the nodes and edges within the component PDGs have been suppressed, and only the dependency edges between the components have been shown. Note that this simplification results in a loss of information, specifically whether there are multiple accesses to the objects within the components and whether these accesses are conditional or not (i.e., nested within a conditional statement).

## REFERENCES.

- A-1. 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.

- A-2. Jin, Z. and Offutt, A.J., "Integration Testing Based on Software Couplings," *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, June 1995, pp. 13-23.
- A-3. Podgurski, A. and Clarke, L.A., "A Formal Model of Program Dependencies and its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transactions on Software Engineering*, Vol. 16, No. 9, September 1990, pp. 965-979.
- A-4. Ferrante, J., Ottenstein, K.J., and Warren, J.D., "The Program Dependency Graph and its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, July 1987, pp. 319-349.
- A-5. Horwitz, S. and Reps, T., "The Use of Program Dependency Graphs in Software Engineering," *Proceedings of the 14th International Conference on Software Engineering (ICSE '92)*, May 1992, pp. 392-411.
- A-6. "Software Considerations in Airborne Systems and Equipment Certification," Document No. RTCA/DO-178B, RTCA Inc., December 1, 1992.
- A-7. Harrold, M.J. and Soffa, M.L., "Interprocedural Data Flow Testing," *SIGSOFT Software Engineering Notes*, Vol. 14, No. 8, December 1989, pp. 158-167.
- A-8. Friedman, M.A. and Voas, J.M., "Software Assessment: Reliability, Safety, Testability," John Wiley & Sons, Inc., 1995.
- A-9. Voas, J.M. and McGraw, G.M., "Software Fault Injection: Inoculating Programs Against Errors," John Wiley & Sons, Inc., 1998.