RTCA, Inc.
1150 18th Street, NW, Suite 910
Washington, DC 20036

Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A

RTCA DO-332
December 13, 2011

Prepared by: SC-205

FOREWORD

This document was prepared by RTCA Special Committee 205 (SC-205) and EUROCAE Working Group 71 (WG-71). It was approved by the RTCA Program Management Committee (PMC) on December 13, 2011.

RTCA, Incorporated is a not-for-profit organization formed to advance the art and science of aviation and aviation electronic systems for the benefit of the public. The organization functions as a Federal Advisory Committee and develops consensus-based recommendations on contemporary aviation issues. RTCA's objectives include but are not limited to:

- coalescing aviation system user and provider technical requirements in a manner that helps government and industry meet their mutual objectives and responsibilities;
- analyzing and recommending solutions to the system technical issues that aviation faces as it continues to pursue increased safety, system capacity, and efficiency;
- developing consensus on the application of pertinent technology to fulfill user and provider requirements, including development of minimum operational performance standards for electronic systems and equipment that support aviation; and
- assisting in developing the appropriate technical material upon which positions for the International Civil Aviation Organization and the International Telecommunication Union and other interested international organizations can be based.

The organization's recommendations are often used as the basis for government and private sector decisions as well as the foundation for many Federal Aviation Administration Technical Standard Orders.

Since the RTCA is not an official agency of the United States Government, its recommendations may not be regarded as statements of official government policy unless so enunciated by the U.S. government organization or agency having statutory jurisdiction over any matters to which the recommendations relate.

TABLE OF CONTENTS

LIST OF TABLES

OO.1.0 INTRODUCTION

Object-oriented technology (OOT) has been widely adopted in non-critical software development projects. The use of this technology for critical software applications in avionics has increased, but there are a number of issues that need to be considered to ensure the safety and integrity goals are met. These issues are both directly related to language features and to complications encountered with meeting well-established safety objectives. In fact, there are a number of related techniques that are used with OOT that need to be considered as well. Clarifying each of these issues will ease the application of object-oriented technology and related techniques (OOT&RT); therefore, this supplement addresses both object-oriented technology and related techniques.

OO.1.1 Purpose

The purpose of this supplement is to provide guidance for the production of software using OOT&RT for systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements. This supplement contains modifications and additions to DO-178C objectives, activities, explanatory text, and software life cycle data that should be addressed when OOT&RT are used as part of the software life cycle.

Compliance with the objectives of DO-178C, Software Considerations in Airborne Systems and Equipment Certification, is the primary means of obtaining approval of software used in civil aviation products. Object-oriented technology is a software development methodology that uses objects and the connections between those objects to express the software design. There are a number of related techniques commonly associated with, but not limited to, object-oriented technology that are considered in this supplement. These related techniques include parametric polymorphism, overloading, type conversion, exception management, dynamic memory management, and virtualization. Since OOT&RT differs from the traditional functional approach to software development, satisfying some of the DO-178C objectives when using OOT&RT may be unclear and/or complicated.

This supplement identifies the additions, modifications, and deletions to DO-178C objectives when object-oriented technology or related techniques are used as part of the software development life cycle and additional guidance is required. This supplement, in conjunction with DO-178C, is intended to provide a common framework for the evaluation and acceptance of OOT&RT based systems.

This supplement provides the following data:

- Objectives for OOT&RT software life cycle processes.
- Descriptions of activities and design considerations for achieving those objectives.
- Descriptions of the evidence that indicate that the objectives have been satisfied.

OO.1.2 <u>Scope</u>

This supplement discusses the use of OOT&RT in the software life cycle for software that is produced in accordance with DO-178C. If the applicant is planning to use OOT or a related technique, then the applicant should comply with the relevant parts of this supplement. The related techniques described in section OO.1.6.2 for which this document applies are parametric polymorphism, overloading, type conversion, exception management, dynamic memory management, and virtualization.

The related techniques discussed in this supplement may be used outside of OOT. Although the primary purpose of their inclusion in this supplement is in their relation to OOT, the guidance for related techniques should be used even when OOT is not used.

OO.1.3 Relationship to Other Documents

This document supplements the guidance given in DO-178C. This supplement may be used in conjunction with other DO-178C supplements.

OO.1.4 How to Use This Supplement

This supplement should be used with, and in the same way as, DO-178C; however, the following should be noted:

a. Section OO.1 contains explanatory text to aid the reader in understanding OOT&RT, and therefore is not to be taken as guidance.

b. Annex OO.A of this supplement describes how the DO-178C objectives are revised in line with this OOT&RT guidance.

c. Annex OO.D of this supplement describes vulnerabilities associated with OOT&RT, as well as supporting information for activities of sections OO.4 through OO.12.

d. Guidance from DO-178C should be used for all aspects of the software life cycle, apart from where OOT&RT are used. All guidance specific to OOT&RT are contained within this supplement.

e. This supplement can be used with DO-278A in the same manner as DO-178C. Applicants complying with DO-278A should be aware that the text within this supplement is based on DO-178C; thus, the DO-278A applicant should ensure that the proper DO-278A guidance is being addressed. Annex OO.C of this supplement describes how the DO-278A, Annex A objectives are revised in line with the guidance in this supplement.

f. In addition to the guidance contained in this supplement, this supplement also provides clarification on the guidance provided in the form of answers to Frequently Asked Questions (FAQs) (see Appendix OO.B). FAQs contain no new or additional guidance.

g. Major sections are numbered as OO.X.0 throughout this supplement. It should be noted that references to an entire section are identified as “section X”; whereas, references to the content between section headers OO.X.0 and OO.X.1 are referenced as “section OO.X.0”.

OO.1.5 Document Overview

This supplement is intended to be used with DO-178C. Sections OO.4 through OO.12 and Annex OO.A provide additional and modified text to the referenced sections of DO-178C. The supplement minimizes any redundancy between DO-178C and the supplement itself. Each affected section of DO-178C is included in the supplement. Where guidance from DO-178C applies unchanged, this is stated in this supplement. Text taken directly from DO-178C is shown in italics. For those sections from DO-178C included in this supplement that are unchanged, except for the references, the DO-178C text is included with the updated OO reference.

In Annex OO.A, only those tables with changes to corresponding tables in DO-178C are included. This includes new objectives and changes to referenced text for existing objectives.

OO.1.6 Characteristics of Object-oriented Technology and Related Techniques

Object-oriented technology is a paradigm for system analysis, design, modeling, and programming centered on objects. There are concepts and techniques commonly found in object-oriented languages that need to be taken into account when considering software safety. Most do not require new guidance per se, but all require consistent interpretation of guidance to ensure safety. Many of these concepts and techniques are also found in other high-level languages; therefore, the discussion here is also germane to any development methodology that uses these concepts and techniques.

OO.1.6.1 Basic Concepts

When considering OOT&RT, there should be understanding of both the technology itself and the techniques with which it is frequently combined. A thorough treatment of OOT&RT features and the underlying concepts would exceed the scope of this supplement.

OO.1.6.1.1 Classes and Objects

The feature that distinguishes OOT is the use of classes to define objects along with the ability to create new classes via subclassing. In procedural programming, a program's behavior is defined by functions, and the state of a running program is defined by the contents of the data variables. In object-oriented programming, functions and data that are closely related are tightly coupled to form a coherent abstraction known as a class. A superclass is a class from which other classes are derived. A superclass is also called a parent class. The classes that are derived from a superclass are known as child classes, derived classes, or subclasses.

A class is a blueprint from which multiple concrete realizations can be created. These concrete realizations are known as objects. Although the terms object and class are often used interchangeably, the distinction should be clear; classes define the types of data within an object and the subprograms that may act upon that data, whereas objects are the concrete manifestation of the class definition within the system. This process of object creation is known as instantiation, and objects are synonymously referred to as class instances or more succinctly as instances.

The subprograms defined for a class are referred to as its methods, or member functions. Methods that operate on or use the data contained within an object instance are referred to as instance methods. This is in contrast to class methods that are associated only with a class, and do not require an associated instance to be invoked. Some languages provide for methods and procedures. Throughout this supplement, the term subprogram is used to refer generically to instance methods, class methods, member functions, and procedures.

Data variables associated with a class are referred to as attributes, fields, or data members. An attribute may be classified as an instance attribute, in which case a separate copy of the attribute exists for each object, or it may be a class attribute, in which case a single, shared copy of the attribute exists for all objects of the class.

OO.1.6.1.2 Types and Type Safety

Given that each object has an associated class, it is natural to consider the type of an object to be its associated class; where type is a designation for a set of objects with well-defined common semantics. A set of types and the rules for their use constitute a type system. These types may be primitive or composite. Primitive types are atomic with regard to the underlying implementation, and usually constitute a predefined, finite set. Composite types, on the other hand, are assembled from primitive types and are usually extendable through new definitions. In object-oriented systems, the set of types is open; that is, the programmer may define new types by creating new classes. Being able to create new types increases the power of object-oriented languages, but it also means that the programmer should ensure that new types do not introduce type inconsistencies.

If types were disjoint and conversion between types were not allowed, maintaining type consistency, and hence type safety, would be trivial; however, developing a system using only disjoint types and preventing type conversions would eliminate much of the advantage of OOT. In OOT, there can be definition of subclasses of already existing classes. Since a class is also a type, a subclass is also a subtype. This means that the subclass should be type consistent with all its superclasses.

For a type to be a proper subtype of some other type, it should exhibit the same behavior as each of its supertypes. In other words, any subtype of a given type should be usable wherever the given type is required. If a subclass is defined that does not adhere to this principle of substitutability, then the type system becomes unsafe. Type checking would fail to detect a misuse of that subtype, since type systems generally allow the substitution of a subtype for any of the type that it refines.

Type consistency can be considered globally or locally. Global type consistency is type consistency throughout the entire class hierarchy of the system. Local type consistency is a subset of global type consistency. Local type consistency is type consistency in a local context where substitution can occur. A local context may be the context of a variable, an attribute, or a parameter. It is generally sufficient to consider local type consistency for type safety.

OO.1.6.1.2.1 Liskov Substitution Principle

The Liskov Substitution Principle (LSP) formally defines what constitutes a proper, that is, type safe, subclass. The principle is, "*Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.*" The LSP limits how a subclass may behave. This means that each subprogram redefined in the subclass should meet the following requirements of the same subprogram in any of its superclasses:

- Preconditions may not be strengthened,
- Postconditions may not be weakened, and
- Invariants may not be weakened.

In addition, the principle implies that no new exceptions should be thrown by subprograms of the subclass, except where those exceptions are themselves subtypes of exceptions thrown by the subprograms of the superclass. In terms of requirements-based verification, LSP means that any subtype of a given type should fulfill all the requirements of the given type. LSP may be demonstrated through the application of formal methods or thorough testing.

OO.1.6.1.3 Hierarchical Encapsulation

There are many problems where a hierarchical system of classification of component parts can aid system understanding. For instance, in tracking helicopters, planes, dirigibles, and gliders, it is helpful to have a classification "aircraft" that has a common set of attributes and behaviors. On the other hand, there may also be interest in those aircraft that need to use a runway and the attributes and behavior that goes with horizontal takeoff and landing aircraft. Classes and subclasses are defined to organize the objects in a system into a hierarchy of object types or classes.

Classes become a much more powerful tool in managing complexity when they are related to each other through a class hierarchy formed through a set of superclass-to-subclass relationships. A class hierarchy is a directed acyclic graph of superclass-to-subclass relationships between classes. A class hierarchy enables the development and manipulation of a potentially large set of subclasses based on the abstraction of a single superclass. Class hierarchies can reduce verification effort and improve comprehension, maintainability, and reuse.

Class hierarchies are formed through both hierarchic composition and hierarchic decomposition. Hierarchic composition is a categorization process involving the identification of common behavior between disparate classes, and the formation of more general superclasses that standardize and generalize those behaviors. Hierarchic decomposition is a categorization process involving the definition of superclasses with behavior that is then supported by subclasses, which add behavior while maintaining substitutability with the superclass. When developing class hierarchies, it is important to recognize that a class hierarchy implies more than software reuse. A class, as a type, is more than a collection of interfaces, properties, and subprograms. The preconditions, postconditions, and invariants for a superclass should be upheld by all of its subclasses. This is particularly important when modifying the class hierarchy.

Class hierarchies are generally implemented via the use of language-supported features such as inheritance, polymorphism, overloading, and varying degrees of type checking. These language features are necessary, but not sufficient, to ensure type safety of a program. It is still necessary to ensure functional consistency.

OO.1.6.1.4 Polymorphism

Polymorphism, which literally means "many forms", is a way of applying a single abstraction to multiple situations, so that each abstraction can be more broadly applied. Polymorphism extends abstractions into implementation. As such, one of the primary goals of polymorphism is to maintain the simplicity of the underlying abstraction.

Traditionally, subprograms are monomorphic. A monomorphic subprogram has operands of unique types. Polymorphism generalizes this by enabling operands (actual parameters) to have more than one type. Polymorphic types may be defined as types whose operations are applicable to operands of more than one type.

Polymorphism has several variants. There are two broad categories: ad-hoc polymorphism and universal polymorphism. Ad-hoc polymorphism can be divided further into implicit coercion and overloading. Universal polymorphism can also be divided into parametric polymorphism and inclusion polymorphism. Each of these forms is provided by widely-used programming languages.

Implicit coercion is the automatic conversion of an operand of one type into another in order to match the required types of a function, in a situation that would otherwise result in a type error. A very common example of this is the transformation of integer values to floating-point before a floating-point arithmetic operation is performed. Another is the extending the range of an integral type; for example, converting a short into an integer value.

Overloading is the use of the same name or symbol for functions that take different arguments; for instance, using the plus operator for both adding numbers and concatenating strings. Another example is an add function with two arguments and an add function with three arguments.

A parametric polymorphic or generic function is a function that takes one or more types as arguments, treating the types uniformly. It provides the means by which the abstraction of common structural and behavioral aspects of a family of classes or operations can be encapsulated in a domain-independent manner. For example, the length function maps a list object of arbitrary type to an integer. It should not matter what type the list is or the types of its members: all lists have the generic abstraction of length. A generic function does the same kind of work independently of the types of its arguments.

Inclusion polymorphism is the ability to substitute any subtype for its supertype. This also means a subprogram of the supertype can be applied to any instances of its subtypes. Since a subclass is also a subtype, subclassing is an example of inclusion polymorphism.

A primary advantage of all forms of polymorphism is the sharing of operands of many types and raising the level of abstraction of the program.

OO.1.6.1.5 Function Passing and Closures

Function passing is a language feature that enables a function to be passed as an argument to another function. In essence, it allows for passing of behavior. A common function-passing example is sorting. Different sorting algorithms, such as quick sort, shell sort, and bubble sort, have been devised. Function passing provides an ability to specify the kind of sort at run-time.

In object-oriented languages, an object's method typically operates on, or uses, the associated object data. This tight binding between an object's method and data makes it impossible to execute a method without the companion object data. Consequently, using function passing to pass a method should also be accompanied by an additional parameter to include the associated object data.

A closure is a subprogram that is evaluated in an environment containing bound variables. During execution, the subprogram can access these variables. In some languages, a closure may also occur when a function is defined within another function, and the inner function refers to local variables of the outer function. At run-time, when the outer function executes, a closure is formed, consisting of the inner function's code and references to any variables of the outer function required by the closure. These local variables become a set of "private" variables, which persist over several invocations of the function. The scope of the variable encompasses only the closed-over function, so it cannot be accessed from other program code. As with the attributes of an object, the variables have indefinite extent, so a value established in one invocation remains available in the next. A closure is a means for implementing the method equivalent to function passing

Closures are useful for increasing the power of function passing. Languages that do not support closures can emulate function passing and closures through object passing. This usually involves creating some interface that includes a method to be called, and then passing an object that implements the given interface. The recipient can then call the predetermined method as a passed function would be called. Because of their flexibility and conciseness, closures are available in some object-oriented languages.

OO.1.6.1.6 Method Dispatch

Method dispatch is the mapping of a method call to its associated implementation. This mapping can be done statically based on the declared type of the object, or it can be done dynamically based on the actual type of the object. In addition, dynamic dispatch can be single, that is, on the implicit or a single argument, or multiple, that is, on multiple arguments to the method.

Static dispatch is normally implemented as a procedure call. It can be done safely only when the method in question is not redefined in a subclass of the static type. When a subtype of the static type redefines the method being called, the method of the static type (an ancestor type of the actual type) is still called. Since this can lead to unintended or erroneous behavior, some languages provide a means to ensure that statically dispatched methods cannot be overridden.

When a method call is dynamic dispatched, the mapping of a specific implementation is performed at run-time. Dynamic dispatching ensures that the method of the actual class is called even though the actual type cannot be determined statically (during compilation). Though this dispatching is analogous to a switch statement over all the subtypes of the static type, it is generally not implemented this way, since to do so would require recompilation every time a new subclass of the static type is added to the system. Rather, each method is associated with an offset in the method table for the class of the object, so dispatching is just a simple indirect procedure call. Dynamic dispatch is only used for code invocation and not for other binding processes such as accessing object attributes directly.

When the decision as to which method to call is made only on the first or implicit argument, for example, the x in x.write(stream), the technique is known as single dispatch. Single dispatch is the most common form of dynamic dispatch because it is both conceptually simpler and easier to implement efficiently. There are some languages that dispatch based on the types of all arguments, for example, both x and stream. This is known as multiple dispatching. There may be expanded verification activities as a result of the decision to use multiple dispatch.

OO.1.6.2 Key Features and Related Techniques

Besides these basic concepts, there are a number of features commonly provided in object-oriented languages. The division between concepts and features is somewhat arbitrary. In this supplement, features denote object-oriented or related technique elements that have direct safety considerations.

OO.1.6.2.1 Inheritance

Inheritance is a language feature in which a class shares properties defined in another class, where the defining class is considered to be the parent. The parent's properties that the inheriting class shares depend on the kind of inheritance used. With interface inheritance, the inheriting class shares only the complete signatures of the methods of its parent without the underlying implementation. On the other hand, with implementation inheritance, the inheriting class shares the implementation as well as the complete method signatures of its parent.

With both interface and implementation inheritance, an inheriting class may have either a single parent or more than one parent. The former case is referred to as single inheritance, the latter as multiple inheritance. Depending on the specific language, several combinations may be allowed, such as using single implementation inheritance while using multiple interface inheritance.

Inheritance enables substitution through subtyping, which means that where an object of a specific class is expected, an object of any of its subclasses can be provided. In the same way, polymorphism occurs when one object is used in the context of multiple declared classes. For example, a superclass may be substituted by one of several subclasses which all inherit the interface of the superclass.

OO.1.6.2.2 Parametric Polymorphism

Parametric polymorphism is a feature commonly used for enhancing the reusability of software components. For example, consider a stack with a generically parameterized type. Such a stack can be instantiated in any number of different forms such as a stack of integers, a stack of a user-defined type, or even a stack of stacks. Many languages support parametric polymorphism: as templates in C++ and Unified Modeling Language (UML), and as generics in Ada, Java, and Eiffel.

OO.1.6.2.3 Overloading (ad hoc polymorphism)

Languages that support overloading enable the reuse of subprogram names with different implementations. The intent behind overloading is to assign the same name or operator symbols to subprograms that are similar semantically. A compiler will select the appropriate subprogram by matching its signature with the types of arguments at the call site.

OO.1.6.2.4 Type Conversion

Type conversion is the act of changing the representation of some value of a source type to the representation of the same value of a target type. Type conversion can be made explicitly by the programmer or implicitly by the compiler according to some predefined type conversion rules. For checked types, the conversion is verified at compile time or run-time if type conversion is possible. Unchecked types are not verified.

In general, there are two categories of type conversion when considering object-oriented languages: a type conversion that changes the underlying structure of the data that represents the type, and a type conversion that changes the view.

A structural change in which the destination data element contains sufficient precision and magnitude to represent the source data is a widening conversion. Conversely, a structural change in which the destination data element does not contain sufficient precision and magnitude to represent the source data is a narrowing conversion.

Type conversions that require changing the view of the underlying representation are more prevalent in OOT. The change of view can either be from a subtype to a supertype (upcast), or from a supertype to a subtype (downcast). The change of view from a subtype to a supertype is allowed by the type system and usually implicit. The change of view from a supertype to a subtype should be performed with care and is generally unsafe without sufficient precautions.

Type conversions may also be categorized as implicit or explicit. An implicit type conversion is an automatic type conversion performed by the compiler, whereas an explicit type conversion is performed based on information supplied by the programmer. Although languages and their compiler implementations generally have strict implicit conversion rules, a lack of familiarity of these rules, or ignorance of the conversion taking place are often the cause of coding problems.

OO.1.6.2.5 Software Exceptions and Exception Handling

Exceptions signal abnormal (error) conditions which are detected within a method. The ability to throw exceptions within a method and to handle exceptions in the calling method is a common feature of most object-oriented languages.

Exceptions may be raised implicitly through compiler-generated checks, explicit checks such as failure of preconditions and postconditions, or explicitly via user actions. Exception handling is used for conditions that deviate from the normal flow of execution. When an exception is raised the execution will transfer to the nearest enclosing handler.

OO.1.6.2.6 Dynamic Memory Management

In many languages, objects can be created on demand during run-time. The deletion of such objects after their use is either performed explicitly using a program statement or automatically by the run-time system.

During the creation of an object, memory is allocated from a memory pool. This memory is freed when the object is deleted. The management of the memory that can be dynamically allocated and deallocated is called dynamic memory management.

Four main techniques for managing dynamic memory are described in the following sections.

OO.1.6.2.6.1 Object Pooling

An initial set of unused objects of a particular type is provided. Each time a new object of the given type is needed, one is taken from the pool. Once an object is no longer needed, it is returned to the pool.

OO.1.6.2.6.2 Activation Frame Management

For objects that are needed only in certain execution contexts (for example, only in a given method and its called methods) allocation within the activation frame can be used. This method may be implemented directly on the stack or indirectly through secondary structures such as scoped memory or a secondary stack.

OO.1.6.2.6.3 Manual Heap Management

Manual heap-based object management requires the user to deallocate each object when and only when the object is no longer referenced. The application developer has to ensure that fragmentation does not cause allocations to fail.

OO.1.6.2.6.4 Automatic Heap Management

Garbage collection is the standard form of automatic heap management. A precise garbage collector ensures that objects that are no longer referenced within the program are deleted.

OO.1.6.2.7 Virtualization Techniques

Virtualization is the general technique of using software (or hardware) to abstract a set of resources at some higher level. Examples include microcode, operating systems, math libraries (abstract access to a floating-point unit), device drivers (abstract access to a physical device), virtual machine managers, and interpreters.

Some virtualization techniques, such as interpreters, are programs that emulate the execution of instructions of a given language. Interpreters can be as simple as an Extensible Markup Language (XML) or a state machine interpreter within an application, or as complex as a programming language interpreter.

The term interpreter should not be taken to mean all software that processes data. For example, making decisions based on the contents of a data packet header or use of parameter data items are not programs processed by such interpreters.

OO.2.0 SYSTEM ASPECTS RELATING TO SOFTWARE DEVELOPMENT

Section 2 of DO-178C is unchanged.

OO.3.0 SOFTWARE LIFE CYCLE

Section 3 of DO-178C is unchanged.

OO.4.0 SOFTWARE PLANNING PROCESS

This section discusses the objectives and activities of the software planning process. This process produces the software plans and standards that direct the software development processes and the integral processes. Table OO.A-1 of Annex OO.A *is a summary of the objectives and outputs of the software planning process by software level.*

The applicant should ensure that the intended use of this supplement is acceptable to the appropriate certification authority. Several supplements may be applied to an applicant's software life cycle. The applicant should plan an approach that will comply with the objectives of DO-178C and those applicable objectives from the supplements. As such, all relevant supplements and the supplements' associated objectives should be considered. If the applicant's particular approach identifies a conflict between the objectives, then the applicant should propose a solution to the conflict in the Plan for Software Aspects of Certification (PSAC). The applicant may use a single set of software plans or multiple sets of software plans addressing the different supplements. Using a single set of software plans containing sections addressing each supplement, will provide an integrated approach with a clear and comprehensive plan.

As supplements only extend guidance from DO-178C for a specific technology, the applicant should first consider all of the DO-178C objectives, and then the supplement's objectives, and lastly any other additional considerations. One approach is to consolidate all objectives of DO-178C and supplements, and for each objective provide a statement of how compliance will be achieved, along with identifying any applicable life cycle data items. Because the objective numbering scheme is unique between the Annex A Tables of DO-178C and the Annex A Tables of the supplements, the identification of the objectives and their document sources will be clear and unambiguous.

For the purpose of this supplement, the term target computer can be interpreted to mean target environment. The target environment is either a target computer or a combination of virtualization software and a target computer. Virtualization software also should comply with DO-178C and applicable supplements.

OO.4.1 Software Planning Process Objectives

The purpose of the software planning process is to define the means of producing software that will satisfy its requirements and provide the level of confidence that is consistent with the software level. There are seven objectives of the software planning process.

a. *The activities of the software development processes and integral processes of the software life cycle that will address the system requirements and software level(s) are defined (*see OO.4.2*).*

b. *The software life cycle(s), including the inter-relationships between the processes, their sequencing, feedback mechanisms, and transition criteria are determined (see* DO-178C *3).*

c. *The software life cycle environment, including the methods and tools to be used for the activities of each software life cycle process has been selected and defined (*see OO.4.4*).*

d. *Additional considerations, such as those discussed in* DO-178C *section 12, have been addressed, if necessary.*

e. *Software development standards consistent with the system safety objectives for the software to be produced are defined (see* DO-178C *4.5).*

f. *Software plans that comply with sections* DO-178C *4.3 and* OO.11 *have been produced.*

g. *Development and revision of the software plans are coordinated (see* DO-178C *4.3).*

OO.4.2 Software Planning Process Activities

Effective planning is a determining factor in producing software that satisfies the guidance of this document. Activities for the software planning process include:

a. *The software plans should be developed that provide direction to the personnel performing the software life cycle processes. See also* DO-178C section *9.1.*

b. *The software development standards to be used for the project should be defined or selected.*

c. *Methods and tools should be chosen that aid error prevention and provide defect detection in the software development processes.*

d. *The software planning process should provide coordination between the software development and integral processes to provide consistency among strategies in the software plans.*

e. *The means should be specified to revise the software plans as a project progresses.*

f. *When multiple-version dissimilar software is used in a system, the software planning process should choose the methods and tools to achieve dissimilarity necessary to satisfy the system safety objectives.*

g. *For the software planning process to be complete, the software plans and software development standards should be under change control and reviews of them completed* (see OO.4.6).

h. *If deactivated code is planned, the software planning process should describe how the deactivation mechanism and deactivated code will be defined and verified to satisfy system safety objectives.*

i. *If user-modifiable software is planned, related processes, tools, environment, and data items substantiating the design (see* DO-178C *5.2.3) should be specified in the software plans and standards.*

j. When parameter data items are planned, the following should be addressed:

1. The way parameter data items are used.

2. The software level of the parameter data items.

3. The processes to develop, verify, and modify parameter data items, and any associated tool qualification.

4. Software load control and compatibility.

k. The software planning process should address any additional considerations that are applicable.

l. If software development activities will be performed by a supplier, planning should address supplier oversight.

m. Describe any planned use of virtualization and its associated executable program(s). Any time that data is interpreted as a set of instructions providing flow control, rather than as a set of values to be processed, virtualization is being used. This data should be treated as executable code and all applicable objectives should be satisfied. See Annex OO.D.1.7.1, for vulnerabilities associated with some virtualization techniques.

n. If the reuse of components is planned, describe how the component will be integrated with the new development. This includes the maintenance of type consistency, requirements mapping, and exception management strategy between the components and the using system. See Annex OO.D.1.1.1, OO.D.1.5.1, and OO.D.2.3.1 for inheritance, exception management, and component-based development vulnerabilities respectively.

Other software life cycle processes may begin before completion of the software planning process if transition criteria for the specific process activity are satisfied.

OO.4.3 Software Plans

The software plans define the means of satisfying the objectives of DO-178C and other applicable documents. *They specify the organizations that will perform those activities. The software plans are* as follows:

- *The Plan for Software Aspects of Certification* (section OO.11.1) *serves as the primary means for communicating the proposed development methods to the certification authority for agreement, and defines the means of compliance.* The Plan for Software Aspects of Certification gives an overview of how and where OOT&RT will be used and the means that will be used to address vulnerabilities and OOT&RT specific considerations.

- *The Software Development Plan* (DO-178C section 11.2) *defines the software life cycle(s) and software development environment, and the means by which the software development process objectives will be satisfied.* The Software Development Plan details the precise development steps that will include the use of OOT&RT. Software life cycle data of section OO.11.2 is unchanged from DO-178C section 11.2.

- *The Software Verification Plan* (DO-178C section 11.3) *defines the means by which the software verification process objectives will be satisfied.* The Software Verification Plan details the methods used to gather the verification evidence. Software life cycle data of section OO.11.3 is unchanged from DO-178C section 11.3.

The remainder of DO-178C section 4.3 is unchanged.

OO.4.4 Software Life Cycle Environment Planning

Planning for the software life cycle environment defines the methods, tools, procedures, programming languages, and target computer *that will be used to develop, verify, control, and produce the software life cycle data* (*see* OO.11*) and software product. Examples of how the software environment chosen can have a beneficial effect on the software include enforcing standards, detecting errors, and implementing error prevention and fault tolerance methods. The software life cycle environment is a potential error source that can contribute to failure conditions. Composition of this software life cycle environment may be influenced by the safety-related requirements determined by the system safety assessment process, for example, the use of dissimilar, redundant components.*

The goal of error prevention methods is to avoid errors during the software development processes that might contribute to a failure condition. The basic principle is to choose requirements development and design methods, tools, and programming languages that limit the opportunity for introducing errors, and verification methods that ensure that errors introduced are detected. The goal of fault tolerance methods is to include safety features in the software design or Source Code to ensure that the software will respond correctly to input data errors and prevent output and control errors. The need for error prevention or fault tolerance methods is determined by the system requirements and the system safety assessment process.

The considerations presented above may affect:

a. *The methods and notations used in the software requirements process and software design process.*

b. *The programming language(s) and methods used in the software coding process.*

c. *The software development environment tools.*

d. *The software verification and software configuration management tools.*

e. *The need for tool qualification (see* DO-178C *12.2).*

f. The assessment of software component reuse and the availability of associated life cycle data.

g. The inclusion of virtualization software within the target environment.

OO.4.4.1 Software Development Environment

Section 4.4.1 of DO-178C is unchanged.

OO.4.4.2 Language and Compiler Considerations

Upon successful completion of verification of the software product, the compiler is considered acceptable for that product. For this to be valid, the software verification process needs to consider particular features of the programming language and compiler. The software planning process considers these features when choosing a programming language and planning for verification. Activities include:

a. Some compilers have features intended to optimize performance of the object code. If the test cases give coverage consistent with the software level, the correctness of the optimization need not be verified. Otherwise, the impact of these features on structural coverage analysis should be determined. Additional information can be found in section 6.4.4.2 of DO-178C.

b. To implement certain features, compilers for some languages may produce object code that is not directly traceable to the Source Code, for example, initialization, built-in error detection, or exception handling (see DO-178C *6.4.4.2.b). The software planning process should provide a means to detect this object code and to ensure verification coverage, and should define the means in the appropriate plan.*

c. If a new compiler, linkage editor, or loader version is introduced, or compiler options are changed during the software life cycle, previous tests and coverage analyses may no longer be valid. The verification planning should provide a means of reverification that is consistent with sections OO.6 *and* DO-178C *12.1.3.*

<u>Note</u>: *Although the compiler is considered acceptable once all of the verification objectives are satisfied, the compiler is only considered acceptable for that product and not necessarily for other products.*

OO.4.4.3 Software Test Environment

Section 4.4.3 of DO-178C is unchanged.

OO.4.5 Software Development Standards

Software development standards define the rules and constraints for the software development processes. The software development standards include the Software Requirements Standards, the Software Design Standards, and the Software Code Standards. The software verification process uses these standards as a basis for evaluating the compliance of actual outputs of a process with intended outputs. Activities for development of the software standards include:

a. The software development standards should comply with section OO.11.

b. The software development standards should enable software components of a given software product or related set of products to be uniformly designed and implemented.

c. The software development standards should disallow the use of constructs or methods that produce outputs that cannot be verified or that are not compatible with safety-related requirements.

d. Robustness should be considered in the software development standards.

Note 1: In developing standards, consideration can be given to previous experience. Constraints and rules on development, design, and coding methods can be included to control complexity. Defensive programming practices may be considered to improve robustness.

Note 2: If allocated to software by system requirements, practices to detect and control errors in stored data, and refresh and monitor hardware status and configuration may be used to mitigate single event upsets.

OO.4.6 Review of the Software Planning Process

Reviews of the software planning process are conducted to ensure that the software plans and software development standards comply with the guidance of this document and means are provided to execute them. Activities include:

a. *Methods are chosen that enable the objectives of this document to be satisfied.*

b. *Software life cycle processes can be applied consistently.*

c. *Each process produces evidence that its outputs can be traced to their activity and inputs, showing the degree of independence of the activity, the environment, and the methods to be used.*

d. *The outputs of the software planning process are consistent and comply with section* OO.11.

OO.5.0 SOFTWARE DEVELOPMENT PROCESSES

This section discusses the objectives and activities of the software development processes. The software development processes are applied as defined by the software planning process (see OO.4) *and the Software Development Plan (see* DO-178C *11.2*). Table OO.A-2 of Annex OO.A *is a summary of the objectives and outputs of the software development processes by software level. The software development processes are:*

- *Software requirements process.*
- *Software design process.*
- *Software coding process.*
- *Integration process.*

Software development processes produce one or more levels of software requirements. High-level requirements are produced directly through analysis of system requirements and system architecture. Usually, these high-level requirements are further developed during the software design process, thus producing one or more successive, lower levels of requirements. However, if Source Code is generated directly from high-level requirements, then the high-level requirements are also considered low-level requirements and the guidance for low-level requirements also apply.

Note: *The applicant may be required to justify software development processes that produce a single level of requirements.*

The development of software architecture involves decisions made about the structure of the software. During the software design process, the software architecture is defined and low-level requirements are developed. Low-level requirements are software requirements from which Source Code can be directly implemented without further information.

Each software development process may produce derived requirements. Some examples of requirements that might be determined to be derived requirements are:

- *The need for interrupt handling software to be developed for the chosen target computer.*
- *The specification of a periodic monitor's iteration rate when not specified by the system requirements allocated to software.*
- *The addition of scaling limits when using fixed point arithmetic.*

High-level requirements may include derived requirements, and low-level requirements may include derived requirements. In order to determine the effects of derived requirements on the system safety assessment and system requirements, all derived requirements should be made available to the system processes including the system safety assessment process.

OO.5.1 Software Requirements Process

Section 5.1 of DO-178C is unchanged.

OO.5.1.1 Software Requirements Process Objectives

Section 5.1.1 of DO-178C is unchanged.

OO.5.1.2 Software Requirements Process Activities

Section 5.1.2 of DO-178C is unchanged.

OO.5.2 Software Design Process

Section 5.2 of DO-178C is unchanged.

OO.5.2.1 Software Design Process Objectives

Section 5.2.1 of DO-178C is unchanged.

OO.5.2.2 Software Design Process Activities

The software design process inputs are the Software Requirements Data, the Software Development Plan, and the Software Design Standards. When the planned transition criteria have been satisfied, the high-level requirements are used in the design process to develop software architecture and low-level requirements. This may involve one or more lower levels of requirements.

The primary output of the process is the Design Description (see DO-178C *11.10) which includes the software architecture and the low-level requirements.*

The software design process is complete when its objectives and the objectives of the integral processes associated with it are satisfied. Activities for this process include:

a. *Low-level requirements and software architecture developed during the software design process should conform to the Software Design Standards and be traceable, verifiable, and consistent.*

b. *Derived low-level requirements and the reason for their existence should be defined and analyzed to ensure that the higher level requirements are not compromised.*

c. *Software design process activities could introduce possible modes of failure into the software or, conversely, preclude others. The use of partitioning or other architectural means in the software design may alter the software level assignment for some components of the software. In such cases, additional data should be defined as derived requirements and provided to the system processes, including the system safety assessment process.*

d. *Interfaces between software components, in the form of data flow and control flow, should be defined to be consistent between the components.*

e. *Control flow and data flow should be monitored when safety-related requirements dictate, for example, watchdog timers, reasonableness-checks, and cross-channel comparisons.*

f. *Responses to failure conditions should be consistent with the safety-related requirements.*

g. *Inadequate or incorrect inputs detected during the software design process should be provided to the system life cycle processes, the software requirements process, or the software planning process as feedback for clarification or correction.*

h. Class hierarchy should be developed based on high-level requirements.

i. A locally type-consistent class hierarchy should be developed with associated low-level requirements wherever substitution is relied upon.

j. As part of the software architecture, strategy for memory management should be developed. See Annex OO.D.1.6.1 for vulnerabilities.

k. As part of the software architecture, strategy for exception management should be developed. See Annex OO.D.1.5.1 for vulnerabilities.

l. When reusing components, the resulting derived requirements and any functionality to be deactivated should be identified. See Annex OO.D.2.3.1 for component-based development vulnerabilities.

See Annex OO.D.1.1.1 for inheritance vulnerabilities.

Note: *The current state of software engineering does not permit a quantitative correlation between complexity and the attainment of system safety objectives. While no objective guidance can be provided, the software design process should avoid introducing complexity because as the complexity of software increases, it becomes more difficult to verify the design and to show that the safety-related requirements are satisfied.*

OO.5.2.3 Designing for User-Modifiable Software

Section 5.2.3 of DO-178C is unchanged.

OO.5.2.4 Designing for Deactivated Code

Section 5.2.4 of DO-178C is unchanged.

OO.5.3 Software Coding Process

Section 5.3 of DO-178C is unchanged.

OO.5.3.1 Software Coding Process Objectives

Section 5.3.1 of DO-178C is unchanged.

OO.5.3.2 Software Coding Process Activities

Section 5.3.2 of DO-178C is unchanged.

OO.5.4 Integration Process

Section 5.4 of DO-178C is unchanged.

OO.5.4.1 Integration Process Objectives

Section 5.4.1 of DO-178C is unchanged.

OO.5.4.2 Integration Process Activities

Section 5.4.2 of DO-178C is unchanged.

OO.5.5 Software Development Process Traceability

Software development process traceability activities include:

a. *Trace Data, showing the bi-directional association between system requirements allocated to software and high-level requirements, is developed. The purpose of this Trace Data is to:*

   1. *Enable verification of the complete implementation of the system requirements allocated to software.*

   2. *Give visibility to those derived high-level requirements that are not directly traceable to system requirements.*

b. *Trace Data, showing the bi-directional association between the high-level requirements and low-level requirements is developed. The purpose of this Trace Data is to:*

   1. *Enable verification of the complete implementation of the high-level requirements.*

   2. *Give visibility to those derived low-level requirements that are not directly traceable to high-level requirements and to the architectural design decisions made during the software design process.*

c. *Trace Data, showing the bi-directional association between low-level requirements and Source Code, is developed. The purpose of this Trace Data is to:*

   1. *Enable verification that no Source Code implements an undocumented function.*

   2. *Enable verification of the complete implementation of the low-level requirements*

d. In an object-oriented design, all functionality is implemented in methods; therefore, traceability is from requirements to the methods and attributes that implement the requirements. Classes are an artifact of the architecture for organizing the requirements. Due to subclassing, a requirement, which traces to a method implemented in a class, should also trace to the method in its subclasses when the method is overridden in a subclass. This is in addition to tracing requirements that are specific to the subclass. See Annex OO.D.2.1.1 for supporting information on traceability.

OO.6.0 SOFTWARE VERIFICATION PROCESS

This section discusses the objectives and activities of the software verification process. Verification is a technical assessment of the outputs of the software planning process, software development processes, and the software verification process. The software verification process is applied as defined by the software planning process (see OO.4) *and the Software Verification Plan (see* DO-178C *11.3). See* OO.4.6 *for the verification of the outputs of the planning process.*

Verification is not simply testing. Testing, in general, cannot show the absence of errors. As a result, the following sections use the term "verify" instead of "test" to discuss the software verification process activities, which are typically a combination of reviews, analyses, and tests.

Tables OO.A-3 through OO.A-7 of Annex OO.A *contain a summary of the objectives and outputs of the software verification process, by software level.*

Note: *For lower software levels, less emphasis is on:*

- *Verification of Source Code.*
- *Verification of low-level requirements.*
- *Verification of the software architecture.*
- *Degree of test coverage.*
- *Control of verification procedures.*
- *Independence of software verification process activities.*
- *Overlapping software verification process activities, that is, multiple verification activities, each of which may be capable of detecting the same class of error.*
- *Robustness testing.*
- *Verification activities with an indirect effect on error prevention or detection, for example, conformance to software development standards.*

OO.6.1 Purpose of Software Verification

Section 6.1 of DO-178C is unchanged.

OO.6.2 Overview of Software Verification Process Activities

Software verification process objectives are satisfied through a combination of reviews, analyses, the development of test cases and procedures, and the subsequent execution of those test procedures. Reviews and analyses provide an assessment of the accuracy, completeness, and verifiability of the software requirements, software architecture, and Source Code. The development of test cases and procedures may provide further assessment of the internal consistency and completeness of the requirements. The execution of the test procedures provides a demonstration of compliance with the requirements.

The inputs to the software verification process include the system requirements, the software requirements, software architecture, Trace Data, Source Code, Executable Object Code, and the Software Verification Plan.

The outputs of the software verification process are recorded in the Software Verification Cases and Procedures (see DO-178C *11.13), the Software Verification Results (*see OO.11.14*), and the associated Trace Data (see* DO-178C *11.21).*

The need for the requirements to be verifiable once they have been implemented in the software may itself impose additional requirements or constraints on the software development processes.

Software verification considerations include:

a. *If the code tested is not identical to the airborne software, those differences should be specified and justified.*

b. *When it is not possible to verify specific software requirements by exercising the software in a realistic test environment, other means should be provided and their justification for satisfying the software verification process objectives defined in the Software Verification Plan or Software Verification Results.*

c. *Deficiencies and errors discovered during the software verification process should be reported to other software life cycle processes for clarification and correction as applicable.*

d. *Reverification should be conducted following corrective actions and/or changes that could impact the previously verified functionality. Reverification should ensure that the modification has been correctly implemented.*

e. *Verification independence is achieved when the verification activity is performed by a person(s) other than the developer of the item being verified. A tool may be used to achieve equivalence to the human verification activity. For independence, the person who created a set of low-level requirements-based test cases should not be the same person who developed the associated Source Code from those low-level requirements.*

f. The class hierarchy should be verified for consistency with requirements.

g. Local type consistency should be verified.

h. The memory management system should be verified for consistency with the strategy defined in the software architecture (see OO.5.2.2.j).

i. The exception management system should be verified for consistency with the strategy defined in the software architecture (see OO.5.2.2.k and Annex OO.D.1.5.1 for vulnerabilities).

OO.6.3 Software Reviews and Analyses

Section 6.3 of DO-178C is unchanged.

OO.6.3.1 Reviews and Analyses of High-Level Requirements

Section 6.3.1 of DO-178C is unchanged.

OO.6.3.2 Reviews and Analyses of Low-Level Requirements

Section 6.3.2 of DO-178C is unchanged.

OO.6.3.3 Reviews and Analyses of Software Architecture

These review and analysis activities detect and report errors that may have been introduced during the development of the software architecture. These review and analysis activities confirm that the software architecture satisfies these objectives:

a. Compatibility with the high-level requirements: *The objective is to ensure that the software architecture does not conflict with the high-level requirements, especially functions that ensure system integrity, for example, partitioning schemes,* exception management, and memory management. See Annex OO.D.1.5.1 and OO.D.1.6.1 for exception and memory management vulnerabilities respectively.

b. Consistency: *The objective is to ensure that a correct relationship exists between the components of the software architecture. This relationship exists via data flow, control flow,* and inheritance within the class hierarchy of the software architecture. *If the interface is to a component of a lower software level, it should also be confirmed that the higher software level component has appropriate protection mechanisms in place to protect itself from potential erroneous inputs from the lower software level component.*

c. Compatibility with the target computer: *The objective is to ensure that no conflicts exist, especially initialization, asynchronous operation, synchronization, and interrupts, between the software architecture and the hardware/software features of the target computer.*

d. Verifiability: *The objective is to ensure that the software architecture can be verified, for example, there are no unbounded recursive algorithms.*

e. Conformance to standards: *The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations to the standards are justified, for example, deviations to complexity restriction and design construct rules.*

f. Partitioning integrity: *The objective is to ensure that partitioning breaches are prevented.*

OO.6.3.4 Reviews and Analyses of Source Code

These review and analysis activities detect and report errors that may have been introduced during the software coding process. Primary concerns include correctness of the code with respect to the software requirements and the software architecture, and conformance to the Software Code Standards. These review and analysis activities are usually confined to the Source Code and confirm that the Source Code satisfies these objectives:

a. Compliance with the low-level requirements: *The objective is to ensure that the Source Code is accurate and complete with respect to the low-level requirements and that no Source Code implements an undocumented function.*

b. Compliance with the software architecture: *The objective is to ensure that the Source Code matches the data flow, control flow,* and class hierarchy *defined in the software architecture.*

c. Verifiability: *The objective is to ensure the Source Code does not contain statements and structures that cannot be verified and that the code does not have to be altered to test it.*

d. Conformance to standards: *The objective is to ensure that the Software Code Standards were followed during the development of the code, for example, complexity restrictions and code constraints. Complexity includes the degree of coupling between software components, the nesting levels for control structures, and the complexity of logical or numeric expressions. This analysis also ensures that deviations to the standards are justified.*

e. Traceability: *The objective is to ensure that the low-level requirements were developed into Source Code.*

f. Accuracy and consistency: *The objective is to determine the correctness and consistency of the Source Code, including* type consistency, type conversion, *stack usage, memory usage, fixed point arithmetic overflow and resolution, floating-point arithmetic, resource contention and limitations, worst-case execution timing, exception handling, use of uninitialized variables, cache management, unused variables, and data corruption due to task or interrupt conflicts. The compiler (including its options), the linker (including its options), and some hardware features may have an impact on the worst-case execution timing and this impact should be assessed.*

See Annex OO.D.1.3.1 and OO.D.1.4.1 for overloading and type conversion vulnerabilities respectively.

OO.6.3.5 Reviews and Analyses of the Outputs of the Integration Process

Section 6.3.5 of DO-178C is unchanged.

OO.6.4 Software Testing

Section 6.4 of DO-178C is unchanged.

OO.6.4.1 Test Environment

Section 6.4.1 of DO-178C is unchanged.

OO.6.4.2 Requirements-Based Test Selection

Section 6.4.2 of DO-178C is unchanged.

OO.6.4.2.1 Normal Range Test Cases

Normal range test cases demonstrate the ability of the software to respond to normal inputs and conditions. Activities include:

a. Real and integer input variables should be exercised using valid equivalence classes and boundary values.

b. For time-related functions, such as filters, integrators, and delays, multiple iterations of the code should be performed to check the characteristics of the function in context.

c. For state transitions, test cases should be developed to exercise the transitions possible during normal operation.

d. For software requirements expressed by logic equations, the normal range test cases should verify the variable usage and the Boolean operators.

e. For object-oriented software, the normal range test cases should ensure that class constructors properly initialize the state of their objects and that the initial state is consistent with the requirements for the class.

OO.6.4.2.2 Robustness Test Cases

Section 6.4.2.2 of DO-178C is unchanged.

OO.6.4.3 Requirements-Based Testing Methods

Section 6.4.3 of DO-178C is unchanged.

See Annex OO.D.2.4 for resource analysis supporting information.

OO.6.4.4 Test Coverage Analysis

Section 6.4.4 of DO-178C is unchanged.

OO.6.4.4.1 Requirements-Based Test Coverage Analysis

Section 6.4.4.1 of DO-178C is unchanged.

OO.6.4.4.2 Structural Coverage Analysis

Section 6.4.4.2 of DO-178C is unchanged.

See Annex OO.D.2.2.3 for structural coverage supporting information.

OO.6.4.4.3 Structural Coverage Analysis Resolution

Section 6.4.4.3 of DO-178C is unchanged.

OO.6.4.5 Reviews and Analyses of Test Cases, Procedures, and Results

Section 6.4.5 of DO-178C is unchanged.

OO.6.5 Software Verification Process Traceability

Section 6.5 of DO-178C is unchanged.

OO.6.6 Verification of Parameter Data Items

Section 6.6 of DO-178C is unchanged.

OO.6.7 Local Type Consistency Verification

The use of inheritance with method overriding and dynamic dispatch requires additional verification activities that should be done either by testing or by analysis using formal methods.

OO.6.7.1 Local Type Consistency Verification Objective

Verify that all type substitutions are safe.

OO.6.7.2 Local Type Consistency Verification Activity

For each subtype where substitution is used, perform one of the following activities (see Annex OO.D.1.1.1 for inheritance vulnerabilities):

- Verify substitutability using formal methods.

- Ensure that each class passes all the tests of all its parent types which the class can replace.

- For each call point, test every method that can be invoked at that call point (pessimistic testing).

OO.6.8 Dynamic Memory Management Verification

Object-oriented programming tends to rely on dynamic memory management. There are several vulnerabilities, outlined in Annex OO.D.1.6.1, that should be considered whenever any dynamic memory management technique is used. This leads to the following guidance.

OO.6.8.1 Dynamic Memory Management Verification Objective

Verify the use of dynamic memory management is robust with regards to reference ambiguity, fragmentation starvation, deallocation starvation, memory exhaustion, premature deallocation, lost updates and stale references, and unbound allocation or deallocation time. See Annex OO.D.1.6.1 for vulnerabilities.

OO.6.8.2 Dynamic Memory Management Verification Activities

When using dynamic memory, the memory management system should be complete and consistent.

For each of the vulnerabilities, identified in Annex OO.D.1.6.1, there should be a corresponding activity to ensure that a given vulnerability does not occur in the final system, including the following:

a. Verify that the allocator returns a memory reference for which no other reference exists (exclusivity).

b. Verify that memory is organized such that any allocation request will succeed when there is sufficient free memory available.

c. Verify that allocated memory which is no longer referenced is reclaimed before that memory is needed for reuse.

d. Verify that there is sufficient memory to accommodate the maximum storage required at any time by the program.

e. Verify that reference consistency is maintained, that is, each object is unique, and is only viewed as that object.

f. Verify that object moves are atomic with respect to all references to the said object.

g. Verify that memory management operations (for example, allocation, deallocation, and the coalescence of free memory) complete within a specified time bound.

This Page Intentionally Left Blank

OO.7.0 SOFTWARE CONFIGURATION MANAGEMENT PROCESS

Section 7 of DO-178C is unchanged.

OO.8.0 SOFTWARE QUALITY ASSURANCE PROCESS

Section 8.0 of DO-178C is unchanged.

OO.8.1 Software Quality Assurance Process Objectives

Section 8.1 of DO-178C is unchanged.

OO.8.2 Software Quality Assurance Process Activities

Activities for satisfying the SQA process objectives include:

a. *The SQA process should take an active role in the activities of the software life cycle processes, and have those performing the SQA process enabled with the authority, responsibility, and independence to ensure that the SQA process objectives are satisfied.*

b. *The SQA process should provide assurance that software plans and standards are developed and reviewed for compliance with* DO-178C, this Supplement, *and for consistency.*

c. *The SQA process should provide assurance that the software life cycle processes comply with the approved software plans and standards.*

d. *The SQA process should include audits of the software life cycle processes during the software life cycle to obtain assurance that:*

    1. *Software plans are available as specified in section* OO.4.2.

    2. *Deviations from the software plans and standards are detected, recorded, evaluated, tracked, and resolved.*

    Note: *It is generally accepted that early detection of process deviations assists efficient achievement of software life cycle process objectives.*

    3. *Approved deviations are recorded.*

    4. *The software development environment has been provided as specified in the software plans.*

    5. *The problem reporting, tracking, and corrective action process activities comply with the Software Configuration Management Plan.*

    6. *Inputs provided to the software life cycle processes by the system processes, including the system safety assessment process, have been addressed.*

    Note: *Monitoring of the activities of software life cycle processes may be performed to provide assurance that the activities are under control.*

e. *The SQA process should provide assurance that the transition criteria for the software life cycle processes have been satisfied in compliance with the approved software plans.*

f. *The SQA process should provide assurance that software life cycle data is controlled in accordance with the control categories as defined in* DO-178C *section 7.3 and the tables of* Annex OO.A and DO-178C Annex A.

g. *Prior to the delivery of software products submitted as part of a certification application, a software conformity review should be conducted.*

h. *The SQA process should produce records of the SQA process activities (see* DO-178C *11.19), including audit results and evidence of completion of the software conformity review for each software product submitted as part of certification application.*

i. *The SQA process should provide assurance that supplier processes and outputs comply with approved software plans and standards.*

OO.8.3 Software Conformity Review

Section 8.3 of DO-178C is unchanged.

OO.9.0 CERTIFICATION LIAISON PROCESS

The objectives of the certification liaison process are to:

a. Establish communication and understanding between the applicant and the certification authority throughout the software life cycle to assist the certification process.

b. Gain agreement on the means of compliance through approval of the Plan for Software Aspects of Certification.

c. Provide compliance substantiation.

*The certification liaison process is applied as defined by the software planning process (*see OO.4*) and the Plan for Software Aspects of Certification* (see OO.11.1). Table OO.A-10 of Annex OO.A *is a summary of the objectives and outputs of this process.*

OO.9.1 Means of Compliance and Planning

The applicant proposes a means of compliance that defines how the development of the airborne system or equipment will satisfy the certification basis. The Plan for Software Aspects of Certification (see OO.11.1) *defines the software aspects of the airborne system or equipment within the context of the proposed means of compliance. This plan also states the software level(s) as determined by the system safety assessment process.*

Activities include:

a. Submitting the Plan for Software Aspects of Certification and other requested data to the certification authority for review.

b. Resolving issues identified by the certification authority concerning the planning for the software aspects of certification.

c. Obtaining agreement with the certification authority on the Plan for Software Aspects of Certification.

OO.9.2 Compliance Substantiation

The applicant provides evidence that the software life cycle processes satisfy the software plans, by making software life cycle data available to the certification authority for review. Certification authority reviews may take place at various facilities. For example, the reviews may take place at the applicant's facilities, the applicant's supplier's facilities, or at the certification authority's facilities. This may involve discussions with the applicant or its suppliers. The applicant arranges these reviews of the activities of the software life cycle processes and makes software life cycle data available as needed.

Activities include:

a. Resolving issues raised by the certification authority as a result of its reviews.

b. Submitting the Software Accomplishment Summary (see OO.11.20) *and Software Configuration Index (see* DO-178C *11.16) to the certification authority.*

c. *Submitting or making available other data or evidence of compliance requested by the certification authority.*

OO.9.3 Minimum Software Life Cycle Data Submitted to Certification Authority

Section 9.3 of DO-178C is unchanged.

OO.9.4 Software Life Cycle Data Related to Type Design

Section 9.4 of DO-178C is unchanged.

OO.10.0 OVERVIEW OF CERTIFICATION PROCESS

Section 10 of DO-178C is unchanged.

OO.11.0 SOFTWARE LIFE CYCLE DATA

Section 11.0 of DO-178C is unchanged.

OO.11.1 Plan for Software Aspects of Certification

The Plan for Software Aspects of Certification (PSAC) is the primary means used by the certification authority for determining whether an applicant is proposing a software life cycle that is commensurate with the rigor required for the level of software being developed. This plan should include:

a. *System overview: This section provides an overview of the system, including a description of its functions and their allocation to the hardware and software, the architecture, processor(s) used, hardware/software interfaces, and safety features.*

b. *Software overview: This section briefly describes the software functions with emphasis on the proposed safety and partitioning concepts. Examples include resource sharing, redundancy, fault tolerance, mitigation of single event upset, and timing and scheduling strategies.*

c. *Certification considerations: This section provides a summary of the certification basis, including the means of compliance, as relating to the software aspects of certification. This section also states the proposed software level(s) and summarizes the justification provided by the system safety assessment process, including potential software contributions to failure conditions.*

d. *Software life cycle: This section defines the software life cycle to be used and includes a summary of each of the software life cycle processes for which detailed information is defined in their respective software plans.* The software life cycle processes should accommodate the technology from the applicable supplements. *The summary explains how the objectives of each software life cycle process* (DO-178C objectives as well as the applicable supplement objectives) *will be satisfied, and specifies the organizations to be involved, the organizational responsibilities, and the system life cycle processes and certification liaison process responsibilities.* This summary should include processes and activities performed in accordance with any supplement. If there are any conflicts between objectives either of DO-178C and/or supplements, an approach to resolve the conflicts should be included.

e. *Software life cycle data: This section specifies the software life cycle data that will be produced and controlled by the software life cycle processes. This section also describes the relationship of the data to each other or to other data defining the system,* including data produced in accordance with any supplement used*, the software life cycle data to be submitted to the certification authority, the form of the data, and the means by which the data will be made available to the certification authority.*

f. *Schedule: This section describes the means the applicant will use to provide the certification authority with visibility of the activities of the software life cycle processes so reviews can be planned.*

g. *Additional considerations: This section describes specific considerations that may affect the certification process. Examples include alternative methods of compliance, tool qualification, previously developed software, option-selectable software, user-*

modifiable software, deactivated code, COTS software, field-loadable software, parameter data items, multiple-version dissimilar software, and product service history.

h. <u>Supplier oversight:</u> This section describes the means of ensuring that supplier processes and outputs will comply with approved software plans and standards.

i. <u>Consideration of OOT&RT features</u>: This section describes the means that will be used to address vulnerabilities and OOT&RT specific considerations identified in Annex OO.D.1 for key features and Annex OO.D.2 for general issues.

OO.11.2 Software Development Plan

Section 11.2 of DO-178C is unchanged.

OO.11.3 Software Verification Plan

Section 11.3 of DO-178C is unchanged.

OO.11.4 Software Configuration Management Plan

Section 11.4 of DO-178C is unchanged.

OO.11.5 Software Quality Assurance Plan

Section 11.5 of DO-178C is unchanged.

OO.11.6 Software Requirements Standards

Section 11.6 of DO-178C is unchanged.

OO.11.7 Software Design Standards

Software Design Standards define the methods, rules, and tools to be used to develop the software architecture and low-level requirements. These standards should include:

a. Design description method(s) to be used.

b. Naming conventions to be used.

c. Conditions imposed on permitted design methods, for example, scheduling, and the use of interrupts and event-driven architectures, dynamic tasking, re-entry, global data, and exception handling, and rationale for their use.

d. Constraints on the use of the design tools.

e. Constraints on design, for example, exclusion of recursion, dynamic objects, data aliases, and compacted expressions.

f. Complexity restrictions, for example, maximum level of nested calls or conditional structures, use of unconditional branches, and number of entry/exit points of code components.

g. Constraints on usage of OOT&RT features addressing the vulnerabilities described in Annex OO.D.1:

1. Inheritance.
2. Parametric polymorphism.
3. Overloading.
4. Type conversion.
5. Exception management.
6. Dynamic memory management.
7. Virtualization.

OO.11.8 Software Code Standards

Software Code Standards define the programming languages, methods, rules, and tools to be used to code the software. These standards should include

a. *Programming language(s) to be used and/or defined subset(s). For a programming language, reference the data that unambiguously defines the syntax, the control behavior, the data behavior, and side-effects of the language. This may require limiting the use of some features of a language.*

b. *Source Code presentation standards, for example, line length restriction, indentation, and blank line usage and Source Code documentation standards, for example, name of author, revision history, inputs and outputs, and affected global data.*

c. *Naming conventions for components, subprograms, variables, and constants.*

d. *Conditions and constraints imposed on permitted coding conventions, such as the degree of coupling between software components and the complexity of logical or numerical expressions and rationale for their use.*

e. *Constraints on the use of the coding tools.*

f. Constraints on usage of OOT&RT features addressing the vulnerabilities described in Annex OO.D.1:

   1. Inheritance.
   2. Parametric polymorphism.
   3. Overloading.
   4. Type conversion.
   5. Exception management.
   6. Dynamic memory management.
   7. Virtualization.

OO.11.9 Software Requirements Data

Section 11.9 of DO-178C is unchanged.

OO.11.10 Design Description

Section 11.10 of DO-178C is unchanged.

OO.11.11 Source Code

Section 11.11 of DO-178C is unchanged.

OO.11.12 Executable Object Code

Section 11.12 of DO-178C is unchanged.

OO.11.13 Software Verification Cases and Procedures

Section 11.13 of DO-178C is unchanged.

OO.11.14 Software Verification Results

The Software Verification Results are produced by the software verification process activities. Software Verification Results should:

a. For each review, analysis, and test, indicate each procedure that passed or failed during the activities and the final pass/fail results.

b. Identify the configuration item or software version reviewed, analyzed, or tested.

c. Include the results of tests, reviews, and analyses, including coverage analyses, local type consistency analyses, dynamic memory management verification*, and traceability analyses.*

Any discrepancies found should be recorded and tracked via problem reporting.

Additionally, evidence provided in support of the system processes' assessment of information provided by the software processes (see DO-178C *2.2.1.f and 2.2.1.g) should be considered to be Software Verification Results.*

OO.11.15 Software Life Cycle Environment Configuration Index

Section 11.15 of DO-178C is unchanged.

OO.11.16 Software Configuration Index

Section 11.16 of DO-178C is unchanged.

OO.11.17 Problem Reports

Section 11.17 of DO-178C is unchanged.

OO.11.18 Software Configuration Management Records

Section 11.18 of DO-178C is unchanged.

OO.11.19 Software Quality Assurance Records

Section 11.19 of DO-178C is unchanged.

OO.11.20 Software Accomplishment Summary

The Software Accomplishment Summary is the primary data item for showing compliance with the Plan for Software Aspects of Certification. This summary should include:

a. *<u>System overview</u>: This section provides an overview of the system, including a description of its functions and their allocation to hardware and software, the architecture, the processor(s) used, the hardware/software interfaces, and safety features. This section also describes any differences from the system overview in the Plan for Software Aspects of Certification.*

b. *<u>Software overview</u>: This section briefly describes the software functions with emphasis on the safety and partitioning concepts used, and explains differences from the software overview proposed in the Plan for Software Aspects of Certification.*

c. *<u>Certification considerations</u>: This section restates the certification considerations described in the Plan for Software Aspects of Certification and describes any differences.*

d. *<u>Software life cycle</u>: This section summarizes the actual software life cycle(s) and explains differences from the software life cycle and software life cycle processes proposed in the Plan for Software Aspects of Certification.*

e. *<u>Software life cycle data</u>: This section describes any differences from the proposals made in the Plan for Software Aspects of Certification for the software life cycle data produced, the relationship of the data to each other and to other data defining the system, and the means by which the data was made available to the certification authority. This section explicitly references, by configuration identifiers and version, the applicable Software Configuration Index and Software Life Cycle Environment Configuration Index. Detailed information regarding configuration identifiers and specific versions of software life cycle data is provided in the Software Configuration Index.*

f. *<u>Additional considerations</u>: This section summarizes any specific considerations that may warrant the attention of the certification authority. It explains any differences from the proposals contained in the Plan for Software Aspects of Certification regarding such considerations. Reference should be made to data items applicable to these matters, such as issue papers or special conditions.*

g. *<u>Supplier oversight</u>: This section describes how supplier processes and outputs comply with plans and standards.*

h. *<u>Software identification</u>: This section identifies the software configuration by part number and version.*

i. *<u>Software characteristics</u>: This section states the Executable Object Code size, timing margins including worst-case execution time, memory margins, resource limitations, and the means used for measuring each characteristic.*

j. *<u>Change history</u>: If applicable, this section includes a summary of software changes with attention to changes made due to failures affecting safety, and identifies any changes from and improvements to the software life cycle processes since the previous certification.*

k. *<u>Software status</u>: This section contains a summary of Problem Reports unresolved at the time of certification. The Problem Report summary includes a description of each problem and any associated errors, functional limitations, operational restrictions, potential adverse effect(s) on safety together with a justification for allowing the Problem Report to remain open, and details of any mitigating action that has been or needs to be carried out.*

l. *<u>Compliance statement</u>: This section includes a statement of compliance with this document and a summary of the methods used to demonstrate compliance with criteria specified in the software plans. This section also addresses additional rulings made by the certification authority and any deviations from the software plans, standards, and this document not covered elsewhere in the Software Accomplishment Summary.*

m. <u>Consideration of OOT&RT features</u>: This section describes how the vulnerabilities and OOT&RT specific considerations identified in Annex OO.D.1 (key features) and Annex OO.D.2 (general issues) were addressed.

OO.11.21 Trace Data

Trace Data establishes the associations between life cycle data items contents. Trace Data should be provided that demonstrates bi-directional associations between:

a. *System requirements allocated to software and high-level requirements.*

b. *High-level requirements and low-level requirements.*

c. *Low-level requirements and Source Code.*

d. *Software Requirements and test cases.*

e. *Test cases and test procedures.*

f. *Test procedures and test results.*

g. Requirements of a method implemented in a class and the requirements of the method in its subclasses when the method is overridden in the subclass.

OO.11.22 Parameter Data Item File

Section 11.22 of DO-178C is unchanged.

OO.12.0 ADDITIONAL CONSIDERATIONS

Section 12.0 of DO-178C is unchanged.

OO.12.1 Use of Previously Developed Software

Section 12.1 of DO-178C is unchanged.

OO.12.1.1 Modifications to Previously Developed Software

This guidance discusses modifications to previously developed software where the outputs of the previous software life cycle processes comply with DO-178C and this Supplement. *Modification may result from requirement changes, the detection of errors, and/or software enhancements.*

Activities include:

a. The revised outputs of the system safety assessment process should be reviewed considering the proposed modifications.

b. If the software level is revised, the guidance of section 12.1.4 of DO-178C *should be considered.*

c. Both the impact of the software requirements changes and the impact of software architecture changes should be analyzed, including the consequences of software requirement changes upon other requirements and the coupling between several software components that may result in reverification effort involving more than the modified area.

d. The area affected by a change should be determined. This may be done by data flow analysis, control flow analysis, timing analysis, traceability analysis, or a combination of these analyses.

e. Areas affected by the change should be reverified in accordance with section OO.6.

OO.12.1.2 Change of Aircraft Installation

Section 12.1.2 of DO-178C is unchanged.

OO.12.1.3 Change of Application or Development Environment

Section 12.1.3 of DO-178C is unchanged.

OO.12.1.4 Upgrading a Development Baseline

Section 12.1.4 of DO-178C is unchanged.

OO.12.1.5 Software Configuration Management Considerations

Section 12.1.5 of DO-178C is unchanged.

OO.12.1.6 Software Quality Assurance Considerations

Section 12.1.6 of DO-178C is unchanged.

OO.12.2 Tool Qualification

Section 12.2 of DO-178C is unchanged.

OO.12.3 Alternative Methods

Sections 12.3 through 12.3.4.1 and 12.3.4.3 through 12.3.4.4 of DO-178C are unchanged.

OO.12.3.4.2 Sufficiency of Accumulated Service History

The required amount of service history is determined by:

a. The system safety objectives of the software and the software level.

b. Any differences in service history environment and system operational environment.

c. The objectives from sections OO.4 to OO.9 *being addressed by service history.*

d. Evidence, in addition to service history, addressing those objectives.

ANNEX OO.A – PROCESS OBJECTIVES AND OUTPUTS BY SOFTWARE LEVEL IN DO-178C

This annex provides tables that describe the objectives, related activities, and outputs given in DO-178C, Annex A that are relevant when using OOT&RT in the production of airborne software.

The tables include guidance for:

a. *The process objectives applicable for each software level. For level E software, see* DO-178C section *2.3.3.*

b. *The independence by software level of the software life cycle process activities applicable to satisfy that process's objectives.*

c. *The control category by software level for the software life cycle data produced by the software life cycle process activities (see* DO-178C section *7.3).*

These tables include references as appropriate to DO-178C and to this supplement. For clear identification, references to DO-178C do not include a prefix, whereas references to this supplement include "OO." as a prefix.

These tables should not be used as a checklist. These tables do not reflect all aspects of compliance to this supplement*. In order to fully understand the guidance, the full body of* DO-178C and this supplement *should be considered.*

The following legend applies to "Applicability by Software Level" and "Control Category by Software Level" for all tables:

LEGEND:	●	*The objective should be satisfied with independence.*
	❍	*The objective should be satisfied.*
	Blank	*Satisfaction of objective is at applicant's discretion.*
	①	*Data satisfies the objectives of Control Category 1 (CC1).*
	②	*Data satisfies the objectives of Control Category 2 (CC2).*

Table OO.A-1 Software Planning Process

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	*The activities of the software life cycle processes are defined.*	OO.4.1.a	OO.4.2.a OO.4.2.c OO.4.2.d OO.4.2.e OO.4.2.g OO.4.2.i OO.4.2.l *4.3.c*	❍	❍	❍	❍	*PSAC*	OO.11.1	①	①	①	①
								SDP	*11.2*	①	①	②	②
								SVP	*11.3*	①	①	②	②
								SCM Plan	*11.4*	①	①	②	②
								SQA Plan	*11.5*	①	①	②	②
2	*The software life cycle(s), including the inter-relationships between the processes, their sequencing, feedback mechanisms, and transition criteria, is defined.*	OO.4.1.b	OO.4.2.i *4.3.b*	❍	❍	❍		*PSAC*	OO.11.1	①	①	①	
								SDP	*11.2*	①	①	②	
								SVP	*11.3*	①	①	②	
								SCM Plan	*11.4*	①	①	②	
								SQA Plan	*11.5*	①	①	②	
3	*Software life cycle environment is selected and defined.*	OO.4.1.c	*4.4.1* OO.4.4.2.a OO.4.4.2.b OO.4.4.2.c *4.4.3*	❍	❍	❍		*PSAC*	OO.11.1	①	①	①	
								SDP	*11.2*	①	①	②	
								SVP	*11.3*	①	①	②	
								SCM Plan	*11.4*	①	①	②	
								SQA Plan	*11.5*	①	①	②	
4	*Additional considerations are addressed.*	OO.4.1.d	OO.4.2.f OO.4.2.h OO.4.2.i OO.4.2.j OO.4.2.k OO.4.2.m OO.4.2.n	❍	❍	❍	❍	*PSAC*	OO.11.1	①	①	①	①
								SDP	*11.2*	①	①	②	②
								SVP	*11.3*	①	①	②	②
								SCM Plan	*11.4*	①	①	②	②
								SQA Plan	*11.5*	①	①	②	②
5	*Software development standards are defined.*	OO.4.1.e	OO.4.2.b OO.4.2.g OO.4.5	❍	❍	❍		*SW Requirements Standards*	*11.6*	①	①	②	
								SW Design Standards	OO.11.7	①	①	②	
								SW Code Standards	OO.11.8	①	①	②	
6	*Software plans comply with this document.*	OO.4.1.f	*4.3.a* OO.4.6	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
7	*Development and revision of software plans are coordinated.*	OO.4.1.g	OO.4.2.g OO.4.6.d	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	

Table OO.A-2 Software Development Processes

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	*High-level requirements are developed.*	*5.1.1.a*	*5.1.2.a* *5.1.2.b* *5.1.2.c* *5.1.2.d* *5.1.2.e* *5.1.2.f* *5.1.2.g* *5.1.2.j* OO.5.5.a	❍	❍	❍	❍	*Software Requirements Data* *Trace Data*	*11.9* OO.11.21	① ①	① ①	① ①	① ①
2	*Derived high-level requirements are defined and provided to the system processes, including the system safety assessment process.*	*5.1.1.b*	*5.1.2.h* *5.1.2.i*	❍	❍	❍	❍	*Software Requirements Data*	*11.9*	①	①	①	①
3	*Software architecture is developed.*	*5.2.1.a*	OO.5.2.2.a OO.5.2.2.d OO.5.2.2.h OO.5.2.2.i OO.5.2.2.j OO.5.2.2.k OO.5.2.2.l	❍	❍	❍	❍	*Design Description*	*11.10*	①	①	①	②
4	*Low-level requirements are developed.*	*5.2.1.a*	OO.5.2.2.a OO.5.2.2.e OO.5.2.2.f OO.5.2.2.g OO.5.2.2.i *5.2.3.a* *5.2.3.b* *5.2.4.a* *5.2.4.b* *5.2.4.c* OO.5.5.b OO.5.5.d	❍	❍	❍		*Design Description* *Trace Data*	*11.10* OO.11.21	① ①	① ①	① ①	
5	*Derived low-level requirements are defined and provided to the system processes, including the system safety assessment process.*	*5.2.1.b*	OO.5.2.2.b OO.5.2.2.c OO.5.2.2.l	❍	❍	❍		*Design Description*	*11.10*	①	①	①	
6	*Source Code is developed.*	*5.3.1.a*	*5.3.2.a* *5.3.2.b* *5.3.2.c* *5.3.2.d* OO.5.5.c	❍	❍	❍		*Source Code* *Trace Data*	*11.11* OO.11.21	① ①	① ①	① ①	

<u>Table OO.A-2 (Continued)</u> Software Development Processes

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
7	*Executable Object Code and Parameter Data Item Files, if any, are produced and loaded in the target computer.*	*5.4.1.a*	*5.4.2.a* *5.4.2.b* *5.4.2.c* *5.4.2.d* *5.4.2.e* *5.4.2.f*	❍	❍	❍	❍	*Executable Object Code* *Parameter Data Item File*	*11.12* *11.22*	① ①	① ①	① ①	① ①

Table OO.A-3 Verification of Outputs of Software Requirements Process

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	**Ref**	**Ref**	**A**	**B**	**C**	**D**	**Data Item**	**Ref**	**A**	**B**	**C**	**D**
1	*High-level requirements comply with system requirements.*	*6.3.1.a*	*6.3.1*	●	●	❍	❍	*Software Verification Results*	OO.11.14	②	②	②	②
2	*High-level requirements are accurate and consistent.*	*6.3.1.b*	*6.3.1*	●	●	❍	❍	*Software Verification Results*	OO.11.14	②	②	②	②
3	*High-level requirements are compatible with target computer.*	*6.3.1.c*	*6.3.1*	❍	❍			*Software Verification Results*	OO.11.14	②	②		
4	*High-level requirements are verifiable.*	*6.3.1.d*	*6.3.1*	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
5	*High-level requirements conform to standards.*	*6.3.1.e*	*6.3.1*	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
6	*High-level requirements are traceable to system requirements.*	*6.3.1.f*	*6.3.1*	❍	❍	❍	❍	*Software Verification Results*	OO.11.14	②	②	②	②
7	*Algorithms are accurate.*	*6.3.1.g*	*6.3.1*	●	●	❍		*Software Verification Results*	OO.11.14	②	②	②	

Table OO.A-4 Verification of Outputs of Software Design Process

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	*Low-level requirements comply with high-level requirements.*	*6.3.2.a*	*6.3.2*	●	●	❍		*Software Verification Results*	OO.11.14	②	②	②	
2	*Low-level requirements are accurate and consistent.*	*6.3.2.b*	*6.3.2*	●	●	❍		*Software Verification Results*	OO.11.14	②	②	②	
3	*Low-level requirements are compatible with target computer.*	*6.3.2.c*	*6.3.2*	❍	❍			*Software Verification Results*	OO.11.14	②	②		
4	*Low-level requirements are verifiable.*	*6.3.2.d*	*6.3.2*	❍	❍			*Software Verification Results*	OO.11.14	②	②		
5	*Low-level requirements conform to standards.*	*6.3.2.e*	*6.3.2*	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
6	*Low-level requirements are traceable to high-level requirements.*	*6.3.2.f*	*6.3.2*	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
7	*Algorithms are accurate.*	*6.3.2.g*	*6.3.2*	●	●	❍		*Software Verification Results*	OO.11.14	②	②	②	
8	*Software architecture is compatible with high-level requirements.*	OO.6.3.3.a	OO.6.3.3	●	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
9	*Software architecture is consistent.*	OO.6.3.3.b	OO.6.3.3	●	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
10	*Software architecture is compatible with target computer.*	OO.6.3.3.c	OO.6.3.3	❍	❍			*Software Verification Results*	OO.11.14	②	②		
11	*Software architecture is verifiable.*	OO.6.3.3.d	OO.6.3.3	❍	❍			*Software Verification Results*	OO.11.14	②	②		
12	*Software architecture conforms to standards.*	OO.6.3.3.e	OO.6.3.3	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
13	*Software partitioning integrity is confirmed.*	OO.6.3.3.f	OO.6.3.3	●	❍	❍	❍	*Software Verification Results*	OO.11.14	②	②	②	②

Table OO.A-5 Verification of Outputs of Software Coding & Integration Processes

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	**Ref**	**Ref**	**A**	**B**	**C**	**D**	**Data Item**	**Ref**	**A**	**B**	**C**	**D**
1	*Source Code complies with low-level requirements.*	OO.6.3.4.a	OO.6.3.4	●	●	❍		*Software Verification Results*	OO.11.14	②	②	②	
2	*Source Code complies with software architecture.*	OO.6.3.4.b	OO.6.3.4	●	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
3	*Source Code is verifiable.*	OO.6.3.4.c	OO.6.3.4	❍	❍			*Software Verification Results*	OO.11.14	②	②		
4	*Source Code conforms to standards.*	OO.6.3.4.d	OO.6.3.4	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
5	*Source Code is traceable to low-level requirements.*	OO.6.3.4.e	OO.6.3.4	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
6	*Source Code is accurate and consistent.*	OO.6.3.4.f	OO.6.3.4	●	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
7	*Output of software integration process is complete and correct.*	*6.3.5 a*	*6.3.5*	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
8	*Parameter Data Item File is correct and complete.*	*6.6.a*	*6.6*	●	●	○	○	Software Verification Cases and Procedures	*11.13*	①	①	②	②
								Software Verification Results	OO.11.14	②	②	②	②
9	*Verification of Parameter Data Item File is achieved.*	*6.6.b*	*6.6*	●	●	○		*Software Verification Results*	OO.11.14	②	②	②	

Table OO.A-6 Testing of Outputs of Integration Process

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	**Ref**	**Ref**	**A**	**B**	**C**	**D**	**Data Item**	**Ref**	**A**	**B**	**C**	**D**
1	*Executable Object Code complies with high-level requirements.*	*6.4.a*	*6.4.2* OO.6.4.2.1 *6.4.3* *6.5*	❍	❍	❍	❍	*Software Verification Cases and Procedures*	*11.13*	①	①	②	②
								Software Verification Results	OO.11.14	②	②	②	②
								Trace Data	OO.11.21	①	①	②	②
2	*Executable Object Code is robust with high-level requirements.*	*6.4.b*	*6.4.2* *6.4.2.2* *6.4.3* *6.5*	❍	❍	❍	❍	*Software Verification Cases and Procedures*	*11.13*	①	①	②	②
								Software Verification Results	OO.11.14	②	②	②	②
								Trace Data	OO.11.21	①	①	②	②
3	*Executable Object Code complies with low-level requirements.*	*6.4.c*	*6.4.2* OO.6.4.2.1 *6.4.3* *6.5*	●	●	❍		*Software Verification Cases and Procedures*	*11.13*	①	①	②	
								Software Verification Results	OO.11.14	②	②	②	
								Trace Data	OO.11.21	①	①	②	
4	*Executable Object Code is robust with low-level requirements.*	*6.4.d*	*6.4.2* *6.4.2.2* *6.4.3* *6.5*	●	❍	❍		*Software Verification Cases and Procedures*	*11.13*	①	①	②	
								Software Verification Results	OO.11.14	②	②	②	
								Trace Data	OO.11.21	①	①	②	
5	*Executable Object Code is compatible with target computer.*	*6.4.e*	*6.4.1 a* *6.4.3.a*	❍	❍	❍	❍	*Software Verification Cases and Procedures*	*11.13*	①	①	②	②
								Software Verification Results	OO.11.14	②	②	②	②

Table OO.A-7 Verification of Verification Process Results

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	**Ref**	**Ref**	**A**	**B**	**C**	**D**	**Data Item**	**Ref**	**A**	**B**	**C**	**D**
1	*Test procedures are correct.*	*6.4.5.b*	*6.4.5*	●	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
2	*Test results are correct and discrepancies explained.*	*6.4.5.c*	*6.4.5*	●	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
3	*Test coverage of high-level requirements is achieved.*	*6.4.4.a*	*6.4.4.1*	●	❍	❍	❍	*Software Verification Results*	OO.11.14	②	②	②	②
4	*Test coverage of low-level requirements is achieved.*	*6.4.4.b*	*6.4.4.1*	●	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	
5	*Test coverage of software structure (modified condition/decision) is achieved.*	*6.4.4.c*	*6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3*	●				*Software Verification Results*	OO.11.14	②			
6	*Test coverage of software structure (decision coverage) is achieved.*	*6.4.4.c*	*6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3*	●	●			*Software Verification Results*	OO.11.14	②	②		
7	*Test coverage of software structure (statement coverage) is achieved.*	*6.4.4.c*	*6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3*	●	●	❍		*Software Verification Results*	OO.11.14	②	②	②	
8	*Test coverage of software structure (data coupling and control coupling) is achieved.*	*6.4.4.d*	*6.4.4.2.c 6.4.4.2 d 6.4.4.3*	●	●	❍		*Software Verification Results*	OO.11.14	②	②	②	
9	*Verification of additional code, that cannot be traced to Source Code, is achieved.*	*6.4.4.c*	*6.4.4.2.b*	●				*Software Verification Results*	OO.11.14	②			
OO 10	Verify local type consistency.	OO.6.7.1	OO.6.7.2	●	●	❍		Software Verification Results	OO.11.14	②	②	②	
OO 11	Verify the use of dynamic memory management is robust.	OO.6.8.1	OO.6.8.2.a OO.6.8.2.b OO.6.8.2.c OO.6.8.2.d OO.6.8.2.e OO.6.8.2.f OO.6.8.2.g	●	●	❍		Software Verification Results	OO.11.14	②	②	②	

Table OO.A-8 Software Configuration Management Process

Table A-8 of DO-178C is unchanged.

<u>Table OO.A-9</u> Software Quality Assurance Process

	Objective		**Activity**	**Applicability by Software Level**				**Output**		**Control Category by Software Level**			
	Description	**Ref**	**Ref**	**A**	**B**	**C**	**D**	**Data Item**	**Ref**	**A**	**B**	**C**	**D**
1	*Assurance is obtained that software plans and standards are developed and reviewed for compliance with* DO-178C and this Supplement *and for consistency.*	*8.1.a*	OO.8.2.b OO.8.2.h OO.8.2.i	●	●	●		*SQA Records*	*11.19*	②	②	②	
2	*Assurance is obtained that software life cycle processes comply with approved software plans.*	*8.1.b*	OO.8.2.a OO.8.2.c OO.8.2.d OO.8.2.f OO.8.2.h OO.8.2.i	●	●	●	●	*SQA Records*	*11.19*	②	②	②	②
3	*Assurance is obtained that software life cycle processes comply with approved software standards.*	*8.1.b*	OO.8.2.a OO.8.2.c OO.8.2.d OO.8.2.f OO.8.2.h OO.8.2.i	●	●	●		*SQA Records*	*11.19*	②	②	②	
4	*Assurance is obtained that transition criteria for the software life cycle processes are satisfied.*	*8.1.c*	OO.8.2.e OO.8.2.h OO.8.2.i	●	●	●		*SQA Records*	*11.19*	②	②	②	
5	*Assurance is obtained that software conformity review is conducted.*	*8.1.d*	OO.8.2.g OO.8.2.h *8.3*	●	●	●	●	*SQA Records*	*11.19*	②	②	②	②

Table OO.A-10 Certification Liaison Process

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	**Ref**	**Ref**	**A**	**B**	**C**	**D**	**Data Item**	**Ref**	**A**	**B**	**C**	**D**
1	*Communication and understanding between the applicant and the certification authority is established.*	OO.9.a	OO.9.1.b OO.9.1.c	❍	❍	❍	❍	*Plan for Software Aspects of Certification*	OO.11.1	①	①	①	①
2	*The means of compliance is proposed and agreement with the Plan for Software Aspects of Certification is obtained.*	OO.9.b	OO.9.1.a OO.9.1.b OO.9.1.c	❍	❍	❍	❍	*Plan for Software Aspects of Certification*	OO.11.1	①	①	①	①
3	*Compliance substantiation is provided.*	OO.9.c	OO.9.2.a OO.9.2.b OO.9.2.c	❍	❍	❍	❍	*Software Accomplishment Summary* *Software Configuration Index*	OO.11.20 *11.16*	① ①	① ①	① ①	① ①

ANNEX OO.B – ACRONYMS AND GLOSSARY OF TERMS

The acronym and glossary definitions provided in this annex are for the terms used in this document. The terms defined in Annex B of DO-178C are not repeated here.

Acronym	**Meaning**
AL	Assurance Level
CC1	Control Category 1
CC2	Control Category 2
CNS/ATM	Communication, Navigation, Surveillance, and Air Traffic Management
COTS	Commercial Off-The-Shelf
DMM	Dynamic Memory Management
FAQ	Frequently Asked Question
LSP	Liskov Substitution Principle
MMI	Memory Management Infrastructure
N/A	Not Applicable
OO	Object-Oriented
OOT	Object-Oriented Technology
OOT&RT	Object-Oriented Technology and Related Techniques
OOTIA	Object-Oriented Technology in Aviation
PSAA	Plan for Software Aspects of Approval
PSAC	Plan for Software Aspects of Certification
SCM	Software Configuration Management
SDP	Software Development Plan
SEU	Single Event Upset
SQA	Software Quality Assurance
SVP	Software Verification Plan
SW	Software
UML	Unified Modeling Language
WCET	Worst-Case Execution Time
XML	Extensible Markup Language

GLOSSARY

Aggregation – A special form of association that specifies the whole-part relationship between the aggregate (whole) and a component (part).

Allocator – A mechanism to obtain a reference to an area of logically contiguous memory of a size specified, to which no other reference exists.

Annotation – Machine readable text in program Source Code that does not change the executional semantics of the code, but contains additional attributes, such as formal requirements.

Attribute – Data within a class that represents values or references to other objects.

Call point – A location in the code, where a subprogram is invoked.

Class – A description of a set of objects that share the same attributes, operations or methods, relationships, and semantics.

Class hierarchy – A collection of classes connected by generalization relationships. The root of the hierarchy represents the most general of these classes. The leaves represent the most specific of these classes.

Closure – A subprogram that is evaluated in an environment containing bound variables.

Conservative Garbage Collector – A garbage collector that cannot distinguish all references from other data. All data that might be a reference is treated as a reference. This ensures that referenced objects are not freed, but can result in some unreachable objects not being collected even though they are no longer usable by the program. This type of collector is used with languages that allow pointer arithmetic, such as C++.

Contravariance – Given a type T with a subtype S, the property of contravariance exists when an object of type T (a wider type) can be used when a more specific (a narrower type) is specified. An example would be a method in one class which takes an argument of type S could be overridden in another class with another method which takes an argument of type T. (Note: See covariance and invariance.)

Covariance – Given a type T with a subtype S, the property of covariance exists when an object of type S (a narrower type) can be used when a more generic (a wider type) is specified. An example would be a method which is specified to return a value of type T which in actuality always returns a value of type S. (Note: See contravariance and invariance.)

Deallocation starvation – The situation in which the process of deallocating memory is not being executed fast enough, to meet the memory allocation demands within the required time bounds.

Deallocation time – The amount of time needed to release a block of information from memory and allows that block to be reused.

Downcasting – The ability to use a reference of a base class to one of its derived classes.

Dynamic dispatch – The association of a method with a call based on the run-time type of the target object.

Dynamic memory management – Any technique for reusing memory at run-time. The essential requirement is to provide means of dynamically allocating portions of memory to programs and freeing memory for reuse when no longer needed. By definition, memory is no longer needed when it is no longer reachable from any part of the program using it.

Encapsulation – The process of compartmentalizing the elements of an abstraction that constitute its structure and behavior.

Escape analysis – A method for determining the dynamic scope of objects in a program.

Exception management – A technique for bypassing the normal control flow of a program. Exceptions can be thrown (or raised) implicitly (language checks) or explicitly in one part of a program and caught (or handled) in an outer nested call frame, that is, closer to the stack base. Uncaught (or unhandled) exceptions could cause premature program termination.

Fragmentation starvation – The situation in which there is sufficient total free memory available to meet the memory allocation request, but the memory is fragmented into many blocks too small to allocate the desired memory block size. Therefore, the memory cannot be allocated, even though sufficient memory is available to satisfy the request.

Function pointer – A pointer in a programming language that contains a reference to a subprogram. By contrast, an "ordinary" pointer references data.

Garbage collection – The process whereby a program reclaims memory that is no longer in use or no longer referenced by an active object.

Global type consistency – For each class in the hierarchy of classes used in the program, that class is type consistent with its parent class. Global type consistency subsumes local type consistency.

Immortal memory – Memory in which objects are only destroyed when the program ends.

Implicit coercion – The automatic conversion of an operand of one type into another in order to match the required types of a function, in a situation which otherwise would result in a type error.

Inheritance – A mechanism by which more specific elements incorporate (inherit) the structure and behavior of more general elements. Inheritance can be used to support generalization, or misused to support only code sharing, without attempting to follow behavioral subtyping rules.

Inline – A language command used to direct the compiler that expansion of a method body within the code of a calling method is to be preferred to the usual call implementation. The compiler can follow or ignore the recommendation to inline.

Instance – An entity to which a set of operations can be applied and which has a state that stores the effects of the operations.

Instrumentation – Code in an application to keep track of some attribute of an executing program.

Interpreter – A program that translates an instruction of one language into a different language, either simultaneously or consecutively.

Invariance – Given a type T with a subtype S, the property of invariance exists when substitutability of T for S is disallowed. (Note: See covariance and contravariance.)

Invariant – A condition associated with a class that is established when a new instance of the class is created and must be maintained by all its publicly accessible operations. As a result, the invariant is effectively a part of the precondition and the postcondition of every such operation. It may be violated in the intermediate states that represent the execution of a given method so long as the operations of the object are properly synchronized and such violations are not externally observable.

Liskov Substitution Principle – Constitutes a proper, that is, type safe, subclass. Liskov formulated the principle succinctly: "Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T."

Live memory – An allocated fragment of logically contiguous memory with at least one reference to it.

Live reference – A reference held in a global, active local, or live memory location.

Local type consistency – For each variable in a program, only values that are type compatible with the declared type of that variable may be assigned to that variable.

Logically contiguous memory – A fragment or set of fragments of memory that can be identified by a single reference.

Lost update – A state change made only to a no longer valid copy of an object.

Memory exhaustion – Inability to allocate memory for stack and heap usage.

Memory management – The act of providing ways to allocate portions of memory to programs at their request, and freeing it for reuse when no longer needed.

Method – The implementation of an operation. A method specifies the algorithm or procedure associated with an operation. A method corresponds to a subprogram with a body in Ada, to a function member with a body in C++, and to a concrete method in Java.

Multiple dispatch – The feature of some object-oriented programming languages in which a function or method can be dynamically dispatched based on the run-time (dynamic) type of more than one of its arguments.

Object – A logical entity contained in a live memory.

Operation – A service that can be requested of an object. An operation has a signature, which may restrict the actual parameters that are possible. An operation corresponds to a subprogram declaration in Ada, to a function member declaration in C++, and to an abstract method declaration in Java. It does not define an associated implementation. (Note: See method.)

Overloading – Use of the same name for different operators or behavioral features (operations or methods) visible within the same scope.

Overriding – The redefinition of an operation or method in a subclass.

Parametric polymorphism – A mechanism for sharing code by parameterizing the types to which the code can be applied. Implementations of parametric polymorphism include generics in Java and Ada and templates in C++ and Objective-C.

Pessimistic testing – A technique where at least one instance of each class that can occur at each call point is tested.

Polymorphism – A way of applying a single abstraction to multiple situations, so that each abstraction can be more broadly applied. Polymorphism has several variants. There are two broad categories: ad-hoc polymorphism and universal polymorphism. Ad-hoc polymorphism can be divided further into implicit coercion and overloading. Universal polymorphism can also be divided into parametric polymorphism and inclusion polymorphism. Each of these forms are provided by widely used programming languages.

Postcondition – A constraint that must be true at the completion of an operation.

Precise Garbage Collector – A garbage collector that can recognize and trace all references to objects on the heap, thus enabling the garbage collector to determine the reachability of all objects. Only a precise garbage collector can ensure that no memory is lost. (Note: See conservative garbage collector.)

Precondition – A constraint that must be true when an operation is invoked.

Premature deallocation – Returning memory that is still referenced back to the free pool.

Reachability analysis – The process of analyzing which data elements are currently able to be referenced by the program.

Reachable object – Object that can be reached by transitive closure of object references, starting from the root of live object references.

Reference – A unique identifier for an object.

Reference ambiguity – Occurs when there is more than one reference for an object or attribute in a symbol table.

Reference consistency – Any location holding a reference will always return the same live memory when dereferenced, so long as the program does not assign a new value to said location, and two locations containing the same reference intend to refer to the same object.

Reusable software – The software, its supporting software life cycle data, and additional supporting documentation being considered for reuse. The component designated for reuse may be any collection of software, such as, libraries, operating systems, or specific system software functions.

Scoped memory - Memory that provides a nesting of allocation domains similar to a stack frame in that scoped memory can be nested similar to stack frames on a stack, but not tied to a single thread. Objects can be allocated by any thread in the area, and all objects are deallocated when the last thread leaves the area. References to objects allocated in a given scoped memory may only be stored in that scope or a more deeply nested scope.

Stack usage analysis – A form of shared resource analysis that establishes the maximum possible size of the stack required by the system and whether there is sufficient physical memory to support this stack size. Some compilers use multiple stacks, and this form of analysis is required for each stack. Potential stack-heap allocation collisions, when these forms of storage compete for the same space, are also included.

Stale reference – A reference to memory that no longer contains the intended object.

State space – The association of a method with a call based on the run-time type of the target object determined during compile time.

Subclass – A class derived from another class.

Subtype – A subset of an existing type which may have additional common properties.

Tail recursion – A form of recursion where no code is executed after any recursive calls in a subprogram.

Type – A set of entities which share a common set of properties.

Type consistency – When a value of the subtype of a given type can be substituted for a value of the given type without changing the behavior of the system, that is, the subtype fulfills all the requirements of the given type.

Type conversion – The act of producing a representation of some value of a target type from a representation of some value of a source type. Type conversion is used to resolve mismatched types in assignments, expressions, or when passing parameters. Type conversions may be either implicit or explicit. With implicit type conversion the compiler is given the responsibility for determining that a conversion is required and how to perform the conversion. With explicit type conversion the programmer assumes the responsibility.

Type safety – A type system to prevent certain undesirable program behavior. A property of some programming languages.

Upcasting – The ability to use a variable of a base type to hold a reference to an object of a derived type.

Virtualization – General technique of using software (or hardware) to abstract a set of resources at some higher level.

Worst-case execution time – The maximum length of time a program segment could take to execute on a specific hardware platform.

ANNEX OO.C – PROCESS OBJECTIVES AND OUTPUTS BY ASSURANCE LEVEL IN DO-278A

This annex provides tables that describe the objectives, related activities, and outputs given in DO-278A, Annex A relevant when using object-oriented technology in the production of software for Communication, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) systems. The tables include guidance for:

a. The process objectives applicable for assurance levels (AL) 1-5, as defined in DO-278A, section 2.3.2.

b. The independence by assurance level of the software life cycle process activities applicable to satisfy that process's objectives.

c. The control category by assurance level for the software life cycle data produced by the software life cycle process activities (DO-278A, section 7.3).

The tables include references as appropriate to DO-278A and to this supplement. For clear identification, references to DO-278A do not include a prefix, whereas references to this supplement include "OO." as a prefix.

These tables should not be used as a checklist. These tables do not reflect all aspects of compliance to this supplement. In order to fully understand the guidance, the full body of DO-278A and this supplement should be considered. When reviewing this supplement for CNS/ATM systems, the following terms specific to the airborne community should be replaced as follows:

- "airborne" with "CNS/ATM system."
- "certification" with "approval."
- "airworthiness requirements" with "applicable approval requirements."
- "certification liaison process" with "approval process."
- "Plan for Software Aspects of Certification" (PSAC) with "Plan for Software Aspects of Approval" (PSAA).

This supplement does not provide guidance for the use of OOT&RT for Commercial Off-The-Shelf (COTS) software or adaptation parameter data (DO-278A, sections 12.4 and 12.5).

The following legend applies to "Applicability by Assurance Level" and "Control Category by Assurance Level" for all tables:

LEGEND:	●	The objective should be satisfied with independence.
	❍	The objective should be satisfied.
	Blank	Satisfaction of objective is at applicant's discretion.
	①	Data satisfies the objectives of Control Category 1 (CC1).
	②	Data satisfies the objectives of Control Category 2 (CC2).

Table OO.C-1 Software Planning Process

	Objective		Activity	Applicability by Assurance Level					Output		Control Category by Assurance Level				
	Description	**Ref**	**Ref**	**AL 1**	**AL 2**	**AL 3**	**AL 4**	**AL 5**	**Data Item**	**Ref**	**AL 1**	**AL 2**	**AL 3**	**AL 4**	**AL 5**
1	*The activities of the software life cycle processes are defined.*	OO.4.1.a	OO.4.2.a OO.4.2.c OO.4.2.d OO.4.2.e OO.4.2.g OO.4.2.i OO.4.2.l *4.3.c*	❍	❍	❍	❍	❍	*PSAA* *SDP* *SVP* *SCM Plan* *SQA Plan*	OO.11.1 *11.2* *11.3* *11.4* *11.5*	① ① ① ① ①	① ① ① ① ①	① ② ② ② ②	① ② ② ② ②	① ② ② ② ②
2	*The software life cycle(s), including the inter-relationships between the processes, their sequencing, feedback mechanisms, and transition criteria, is defined.*	OO.4.1.b	OO.4.2.i *4.3.b*	❍	❍	❍	❍		*PSAA* *SDP* *SVP* *SCM Plan* *SQA Plan*	OO.11.1 *11.2* *11.3* *11.4* *11.5*	① ① ① ① ①	① ① ① ① ①	① ② ② ② ②	① ② ② ② ②	
3	*Software life cycle environment is selected and defined.*	OO.4.1.c	*4.4.1* OO.4.4.2.a OO.4.4.2.b OO.4.4.2.c *4.4.3*	❍	❍	❍	❍		*PSAA* *SDP* *SVP* *SCM Plan* *SQA Plan*	OO.11.1 *11.2* *11.3* *11.4* *11.5*	① ① ① ① ①	① ① ① ① ①	① ② ② ② ②	① ② ② ② ②	
4	*Additional considerations are addressed.*	OO.4.1.d	OO.4.2.f OO.4.2.h OO.4.2.i OO.4.2.j OO.4.2.k OO.4.2.m OO.4.2.n	❍	❍	❍	❍	❍	*PSAA* *SDP* *SVP* *SCM Plan* *SQA Plan*	OO.11.1 *11.2* *11.3* *11.4* *11.5*	① ① ① ① ①	① ① ① ① ①	① ② ② ② ②	① ② ② ② ②	① ② ② ② ②
5	*Software development standards are defined.*	OO.4.1.e	OO.4.2.b OO.4.2.g OO.4.5	❍	❍	❍	❍		*SW Requirements Standards* *SW Design Standards* *SW Code Standards*	*11.6* OO.11.7 OO.11.8	① ① ①	① ① ①	② ② ②	② ② ②	
6	*Software plans comply with this document.*	OO.4.1.f	*4.3.a* OO.4.6	❍	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	
7	*Development and revision of software plans are coordinated.*	OO.4.1.g	OO.4.2.g OO.4.6.d	❍	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	

Table OO.C-2 Software Development Processes

	Objective		Activity	Applicability by Assurance Level					Output		Control Category by Assurance Level				
	Description	**Ref**	**Ref**	**AL 1**	**AL 2**	**AL 3**	**AL 4**	**AL 5**	**Data Item**	**Ref**	**AL 1**	**AL 2**	**AL 3**	**AL 4**	**AL 5**
1	*High-level requirements are developed.*	5.1.1.a	*5.1.2.a* *5.1.2.b* *5.1.2.c* *5.1.2.d* *5.1.2.e* *5.1.2.f* *5.1.2.g* *5.1.2.j* OO.5.5.a	❍	❍	❍	❍	❍	*Software Requirements Data* *Trace Data*	*11.9* OO.11.21	① ①	① ①	① ①	① ①	① ①
2	*Derived high-level requirements are defined and provided to the system processes, including the system safety assessment process.*	*5.1.1.b*	*5.1.2.h* *5.1.2.i*	❍	❍	❍	❍	❍	*Software Requirements Data*	*11.9*	①	①	①	①	①
3	*Software architecture is developed.*	*5.2.1.a*	OO.5.2.2.a OO.5.2.2.d OO.5.2.2.h OO.5.2.2.i OO.5.2.2.j OO.5.2.2.k OO.5.2.2.l	❍	❍	❍	❍	❍	*Design Description*	*11.10*	①	①	①	①	②
4	*Low-level requirements are developed.*	*5.2.1.a*	OO.5.2.2.a OO.5.2.2.e OO.5.2.2.f OO.5.2.2.g OO.5.2.2.i *5.2.3.a* *5.2.3.b* *5.2.4.a* *5.2.4.b* *5.2.4.c* OO.5.5.b OO.5.5.d	❍	❍	❍			*Design Description* *Trace Data*	*11.10* OO.11.21	① ①	① ①	① ①		
5	*Derived low-level requirements are defined and provided to the system processes, including the system safety assessment process.*	*5.2.1.b*	OO.5.2.2.b OO.5.2.2.c OO.5.2.2.l	❍	❍	❍			*Design Description*	*11.10*	①	①	①		
6	*Source Code is developed.*	*5.3.1.a*	*5.3.2.a* *5.3.2.b* *5.3.2.c* *5.3.2.d* OO.5.5.c	❍	❍	❍			*Source Code* *Trace Data*	*11.11* OO.11.21	① ①	① ①	① ①		

<u>Table OO.C-2 (Continued)</u> Software Development Processes

	Objective		Activity	Applicability by Assurance Level					Output		Control Category by Assurance Level				
	Description	Ref	Ref	AL 1	AL 2	AL 3	AL 4	AL 5	Data Item	Ref	AL 1	AL 2	AL 3	AL 4	AL 5
7	*Executable Object Code and Adaptation Data Item Files, if any, are produced and loaded in the target computer.*	*5.4.1.a*	*5.4.2.a 5.4.2.b 5.4.2.c 5.4.2.d 5.4.2.e 5.4.2.f*	❍	❍	❍	❍	❍	*Executable Object Code*	*11.12*	①	①	①	①	①
									Adaptation Data Item File	*11.22*	①	①	①	①	①

Table OO.C-3 Verification of Outputs of Software Requirements Process

	Objective		Activity	Applicability by Assurance Level					Output		Control Category by Assurance Level				
	Description	Ref	Ref	AL 1	AL 2	AL 3	AL 4	AL 5	Data Item	Ref	AL 1	AL 2	AL 3	AL 4	AL 5
1	*High-level requirements comply with system requirements.*	*6.3.1.a*	*6.3.1*	●	●	❍	❍	❍	*Software Verification Results*	OO.11.14	②	②	②	②	②
2	*High-level requirements are accurate and consistent.*	*6.3.1.b*	*6.3.1*	●	●	❍	❍	❍	*Software Verification Results*	OO.11.14	②	②	②	②	②
3	*High-level requirements are compatible with target computer.*	*6.3.1.c*	*6.3.1*	❍	❍				*Software Verification Results*	OO.11.14	②	②			
4	*High-level requirements are verifiable.*	*6.3.1.d*	*6.3.1*	❍	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	
5	*High-level requirements conform to standards.*	*6.3.1.e*	*6.3.1*	❍	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	
6	*High-level requirements are traceable to system requirements.*	*6.3.1.f*	*6.3.1*	❍	❍	❍	❍	❍	*Software Verification Results*	OO.11.14	②	②	②	②	②
7	*Algorithms are accurate.*	*6.3.1.g*	*6.3.1*	●	●	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	

Table OO.C-4 Verification of Outputs of Software Design Process

	Objective		Activity	Applicability by Assurance Level					Output		Control Category by Assurance Level				
	Description	Ref	Ref	AL 1	AL 2	AL 3	AL 4	AL 5	Data Item	Ref	AL 1	AL 2	AL 3	AL 4	AL 5
1	*Low-level requirements comply with high-level requirements.*	*6.3.2.a*	*6.3.2*	●	●	❍			*Software Verification Results*	OO.11.14	②	②	②		
2	*Low-level requirements are accurate and consistent.*	*6.3.2.b*	*6.3.2*	●	●	❍			*Software Verification Results*	OO.11.14	②	②	②		
3	*Low-level requirements are compatible with target computer.*	*6.3.2.c*	*6.3.2*	❍	❍				*Software Verification Results*	OO.11.14	②	②			
4	*Low-level requirements are verifiable.*	*6.3.2.d*	*6.3.2*	❍	❍				*Software Verification Results*	OO.11.14	②	②			
5	*Low-level requirements conform to standards.*	*6.3.2.e*	*6.3.2*	❍	❍	❍			*Software Verification Results*	OO.11.14	②	②	②		
6	*Low-level requirements are traceable to high-level requirements.*	*6.3.2.f*	*6.3.2*	❍	❍	❍			*Software Verification Results*	OO.11.14	②	②	②		
7	*Algorithms are accurate.*	*6.3.2.g*	*6.3.2*	●	●	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	
8	*Software architecture is compatible with high-level requirements.*	OO.6.3.3.a	OO.6.3.3	●	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	
9	*Software architecture is consistent.*	OO.6.3.3.b	OO.6.3.3	●	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	
10	*Software architecture is compatible with target computer.*	OO.6.3.3.c	OO.6.3.3	❍	❍				*Software Verification Results*	OO.11.14	②	②			
11	*Software architecture is verifiable.*	OO.6.3.3.d	OO.6.3.3	❍	❍				*Software Verification Results*	OO.11.14	②	②			
12	*Software architecture conforms to standards.*	OO.6.3.3.e	OO.6.3.3	❍	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	
13	*Software partitioning integrity is confirmed.*	OO.6.3.3.f	OO.6.3.3	●	❍	❍	❍	❍	*Software Verification Results*	OO.11.14	②	②	②	②	②

Table OO.C-5 Verification of Outputs of Software Coding & Integration Processes

	Objective		Activity	Applicability by Assurance Level					Output		Control Category by Assurance Level				
	Description	**Ref**	**Ref**	**AL 1**	**AL 2**	**AL 3**	**AL 4**	**AL 5**	**Data Item**	**Ref**	**AL 1**	**AL 2**	**AL 3**	**AL 4**	**AL 5**
1	*Source Code complies with low-level requirements.*	OO.6.3.4.a	OO.6.3.4	●	●	❍			*Software Verification Results*	OO.11.14	②	②	②		
2	*Source Code complies with software architecture.*	OO.6.3.4.b	OO.6.3.4	●	❍	❍			*Software Verification Results*	OO.11.14	②	②	②		
3	*Source Code is verifiable.*	OO.6.3.4.c	OO.6.3.4	❍	❍				*Software Verification Results*	OO.11.14	②	②			
4	*Source Code conforms to standards.*	OO.6.3.4.d	OO.6.3.4	❍	❍	❍			*Software Verification Results*	OO.11.14	②	②	②		
5	*Source Code is traceable to low-level requirements.*	OO.6.3.4.e	OO.6.3.4	❍	❍	❍			*Software Verification Results*	OO.11.14	②	②	②		
6	*Source Code is accurate and consistent.*	OO.6.3.4.f	OO.6.3.4	●	❍	❍			*Software Verification Results*	OO.11.14	②	②	②		
7	*Output of software integration process is complete and correct.*	*6.3.5.a*	*6.3.5*	❍	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	
8	*Adaptation Data Item File is correct and complete.*	*6.6.a*	*6.6*	●	●	❍	❍	❍	*Software Verification Cases and Procedures*	*11.13*	①	①	②	②	②
									Software Verification Results	OO.11.14	②	②	②	②	②
9	*Verification of Adaptation Data Item File is achieved*	*6.6.b*	*6.6*	●	●	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	

Table OO.C-6 Testing of Outputs of Integration Process

	Objective		Activity	Applicability by Assurance Level					Output		Control Category by Assurance Level				
	Description	**Ref**	**Ref**	**AL 1**	**AL 2**	**AL 3**	**AL 4**	**AL 5**	**Data Item**	**Ref**	**AL 1**	**AL 2**	**AL 3**	**AL 4**	**AL 5**
1	*Executable Object Code complies with high-level requirements.*	*6.4.a*	*6.4.2* OO.6.4.2.1 *6.4.3* *6.5*	❍	❍	❍	❍	❍	*Software Verification Cases and Procedures*	*11.13*	①	①	②	②	②
									Software Verification Results	OO.11.14	②	②	②	②	②
									Trace Data	OO.11.21	①	①	②	②	②
2	*Executable Object Code is robust with high-level requirements.*	*6.4.b*	*6.4.2* *6.4.2.2* *6.4.3* *6.5*	❍	❍	❍	❍	❍	*Software Verification Cases and Procedures*	*11.13*	①	①	②	②	②
									Software Verification Results	OO.11.14	②	②	②	②	②
									Trace Data	OO.11.21	①	①	②	②	②
3	*Executable Object Code complies with low-level requirements.*	*6.4.c*	*6.4.2* OO.6.4.2.1 *6.4.3* *6.5*	●	●	❍			*Software Verification Cases and Procedures*	*11.13*	①	①	②		
									Software Verification Results	OO.11.14	②	②	②		
									Trace Data	OO.11.21	①	①	②		
4	*Executable Object Code is robust with low-level requirements.*	*6.4.d*	*6.4.2* *6.4.2.2* *6.4.3* *6.5*	●	❍	❍			*Software Verification Cases and Procedures*	*11.13*	①	①	②		
									Software Verification Results	OO.11.14	②	②	②		
									Trace Data	OO.11.21	①	①	②		
5	*Executable Object Code is compatible with target computer.*	*6.4.e*	*6.4.1.a* *6.4.3.a*	❍	❍	❍	❍	❍	*Software Verification Cases and Procedures*	*11.13*	①	①	②	②	②
									Software Verification Results	OO.11.14	②	②	②	②	②

Table OO.C-7 Verification of Verification Process Results

	Objective		Activity	Applicability by Assurance Level					Output		Control Category by Assurance Level				
	Description	Ref	Ref	AL 1	AL 2	AL 3	AL 4	AL 5	Data Item	Ref	AL 1	AL 2	AL 3	AL 4	AL 5
1	*Test procedures are correct.*	*6.4.5.b*	*6.4.5*	●	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	
2	*Test results are correct and discrepancies explained.*	*6.4.5.c*	*6.4.5*	●	❍	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	
3	*Test coverage of high-level requirements is achieved.*	*6.4.4.a*	*6.4.4.1*	●	❍	❍	❍	❍	*Software Verification Results*	OO.11.14	②	②	②	②	②
4	*Test coverage of low-level requirements is achieved.*	*6.4.4.b*	*6.4.4.1*	●	❍	❍			*Software Verification Results*	OO.11.14	②	②	②		
5	*Test coverage of software structure (modified condition/decision) is achieved.*	*6.4.4.c*	*6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3*	●					*Software Verification Results*	OO.11.14	②				
6	*Test coverage of software structure (decision coverage) is achieved.*	*6.4.4.c*	*6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3*	●	●				*Software Verification Results*	OO.11.14	②	②			
7	*Test coverage of software structure (statement coverage) is achieved.*	*6.4.4.c*	*6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3*	●	●	❍			*Software Verification Results*	OO.11.14	②	②	②		
8	*Test coverage of software structure (data coupling and control coupling) is achieved.*	*6.4.4.d*	*6.4.4.2.c 6.4.4.2.d 6.4.4.3*	●	●	❍	❍		*Software Verification Results*	OO.11.14	②	②	②	②	
9	*Verification of additional code, that cannot be traced to Source Code, is achieved*	*6.4.4.c*	*6.4.4.2.b*	●					*Software Verification Results*	OO.11.14	②				
OO 10	Verify local type consistency	OO.6.7.1	OO.6.7.2	●	●	❍			*Software Verification Results*	OO.11.14	②	②	②		
OO 11	Verify the use of dynamic memory management is robust	OO.6.8.1	OO.6.8.2.a OO.6.8.2.b OO.6.8.2.c OO.6.8.2.d OO.6.8.2.e OO.6.8.2.f OO.6.8.2.g	●	●	❍			*Software Verification Results*	OO.11.14	②	②	②		

<u>Table OO.C-8</u> Software Configuration Management Process

DO-278A, Table A-8 is unchanged.

Table OO.C-9 Software Quality Assurance Process

	Objective		Activity	Applicability by Assurance Level					Output		Control Category by Assurance Level				
	Description	Ref	Ref	AL 1	AL 2	AL 3	AL 4	AL 5	Data Item	Ref	AL 1	AL 2	AL 3	AL 4	AL 5
1	*Assurance is obtained that software plans and standards are developed and reviewed for compliance to* DO-278A and this Supplement *and for consistency.*	*8.1.a*	OO.8.2.b OO.8.2.h OO.8.2.i	●	●	●	●		*SQA Records*	*11.19*	②	②	②	②	
2	*Assurance is obtained that software life cycle processes comply with approved software plans.*	*8.1.b*	OO.8.2.a OO.8.2.c OO.8.2.d OO.8.2.f OO.8.2.h OO.8.2.i	●	●	●	●	●	*SQA Records*	*11.19*	②	②	②	②	②
3	*Assurance is obtained that software life cycle processes comply with approved software standards.*	*8.1.b*	OO.8.2.a OO.8.2.c OO.8.2.d OO.8.2.f OO.8.2.h OO.8.2.i	●	●	●	●		*SQA Records*	*11.19*	②	②	②	②	
4	*Assurance is obtained that transition criteria for the software life cycle processes are satisfied.*	*8.1.c*	OO.8.2.e OO.8.2.h OO.8.2.i	●	●	●	●		*SQA Records*	*11.19*	②	②	②	②	
5	*Assurance is obtained that software conformity review is conducted.*	*8.1.d*	OO.8.2.g OO.8.2.h *8.3*	●	●	●	●	●	*SQA Records*	*11.19*	②	②	②	②	②

Table OO.C-10 Certification Liaison Process

	Objective		Activity	Applicability by Assurance Level					Output		Control Category by Assurance Level				
	Description	Ref	Ref	AL 1	AL 2	AL 3	AL 4	AL 5	Data Item	Ref	AL 1	AL 2	AL 3	AL 4	AL 5
1	*Communication and understanding between the applicant and the approval authority is established.*	OO.9.a	OO.9.1.b OO.9.1.c	❍	❍	❍	❍	❍	*Plan for Software Aspects of Approval*	OO.11.1	①	①	①	①	①
2	*The means of compliance is proposed and agreement with the Plan for Software Aspects of Approval is obtained.*	OO.9.b	OO.9.1.a OO.9.1.b OO.9.1.c	❍	❍	❍	❍	❍	*Plan for Software Aspects of Approval*	OO.11.1	①	①	①	①	①
3	*Compliance substantiation is provided.*	OO.9.c	OO.9.2.a OO.9.2.b OO.9.2.c	❍	❍	❍	❍	❍	*Software Accomplishment Summary* *Software Configuration Index*	OO.11.20 *11.16*	① ①	① ①	① ①	① ①	① ①

ANNEX OO.D –VULNERABILITY ANALYSIS

This annex provides the vulnerability analysis for the objectives, related activities, and outputs given in this supplement in sections OO.4 through OO.12, that are relevant when using OOT&RT in the production of avionics software.

Though using OOT&RT can increase the safety and robustness of software systems, there are a number of issues that should be considered. These issues are both directly related to new language features and to potential complications encountered with meeting well-established safety objectives. Clarifying each of these issues will ease the application of OOT&RT.

This annex includes supporting information for activities for the following:

a. The key features and related techniques of OOT&RT: inheritance, parametric polymorphism, overloading, type conversion, exception management, dynamic memory management, and virtualization (see OO.D.1).

b. The general issues of OOT&RT: traceability, structural coverage, component-based development, and resource analysis (see OO.D.2).

This annex refers to guidance from sections OO.4 through OO.12 of this supplement and sections from DO-178C. The sections titled "Supporting Information for Activities" provide additional clarification for the use of OOT&RT with DO-178C. As noted in section OO.1.4.e, this Annex is also applicable to DO-278A, even though only DO-178C references are identified throughout.

OO.D.1 Key Features and Related Techniques

This section provides information for issues associated with key features of OOT. Inheritance is a core feature of OOT, all other features in this section are related techniques.

OO.D.1.1 Inheritance

Inheritance is the central feature of object-oriented technology. It provides a mechanism for specialization with code reuse. This enables many different classes to implement the same functionality so that different objects can be handled in a uniform manner.

OO.D.1.1.1 Vulnerabilities

Inheritance may introduce several vulnerabilities that should be considered in the certification process. These vulnerabilities are associated with the following: substitutability, method implementation inheritance, unused code, dispatching, and multiple inheritance. Normally, any subclass is substitutable for its superclass. In terms of type theory, a non-parameterized subclass is also a subtype of its superclass. When such a subclass is not a proper subtype, class substitution can cause an application to behave incorrectly. This can occur when a method is overridden in the subclass such that it is functionally incompatible with the original definition.

Method implementation inheritance can be problematic when a subclass has an additional attribute and a method which needs to update the attribute is not overridden. When the value in that attribute is dependent on the results of executing the inherited method, it may not be updated. A call to the method could then result in an improper object state and unexpected behavior.

Another vulnerability is the case where a method of a superclass is overridden by the subclass and the superclass is not instantiated. This may result in deactivated code in the superclass. Removing the method at the superclass may break the integrity of the superclass. In this case, the deactivated method may be overlooked.

When static dispatch is used with method overriding, a different vulnerability can arise. Though it is clear what method is called at any given call point, the method called depends on the declared type of the object. With static dispatch, when the declared type is the superclass but the actual type is a subclass, then when an overridden method is called, the superclass method is called instead of the subclass method. The result can be incorrect behavior or state inconsistency.

Analogous to static dispatch, some programming languages allow a class's attributes to be overridden by a derived class. Depending on the actual language semantics, overriding attribute names can lead to a range of unexpected behaviors based on use of incorrect data.

Finally, when considering multiple inheritance, there are two kinds to consider: interface inheritance and implementation inheritance. Multiple interface inheritance enables a class to inherit method signatures from more than one parent class (sometimes referred to as an interface, that is, a class without any implementation code). This means that a given class can have more than one type. It allows a given method signature (name and argument list) to be inherited from two different classes. This signature could have incompatible intentions in the two parent classes. The result can be unexpected behavior. For languages that provide multiple implementation inheritance, this situation is more complex. Multiple implementation inheritance combines the problems of method inheritance with multiple interface inheritance. If the wrong implementation is inherited or an attribute is inherited through multiple paths, unexpected behavior can also result.

OO.D.1.1.2 Related Guidance

New activities were added to sections OO.4.2, OO.5.2.2, and OO.6.7.2 to address these vulnerabilities.

OO.D.1.1.3 Supporting Information for Activities

Type consistency can be demonstrated by ensuring that each subclass is substitutable for its superclass. Several methods exist, including the use of formal methods or requirements-based testing using the preconditions, postconditions, and invariants, to verify that the LSP properties are maintained. Whatever verification activities are performed on the superclass should also be performed on the subclass. A given class should be verified using both the verification procedures corresponding to its own requirements and the verification procedures that were used to verify all of its parents. If type consistency was not maintained, every member of the dispatch set at each call point would need to be verified.

Design standards should address complexity issues and their relation to design assurance level. Examples may include class size, control coupling, the depth of inheritance, and the number of immediate superclasses a class may have.

OO.D.1.2 Parametric Polymorphism

Parametric polymorphism enables reuse without subtyping, and is a means to maintain type consistency.

OO.D.1.2.1 Vulnerabilities

The parametric polymorphic operations that act on the generic data may not be consistent with the substituted data.

Source Code to object code traceability may be more difficult. There are multiple techniques available to compiler writers for implementing parametric polymorphism. The two most common are referred to here as “macro expansion” and “code sharing”. Macro expansion effectively generates the requisite code for the generic or template at the point of its instantiation. Code sharing generates a single block of code for all instances of the generic or template.

Note: In some languages, the instantiation may be implicit, making traceability less obvious than with explicitly instantiated templates. This may result in a one-to-many mapping of Source Code to object code when macro expansion is used.

Separate instantiations of the same generic can have different characteristics that may need separate verification.

OO.D.1.2.2 Related Guidance

These vulnerabilities are addressed by DO-178C sections 6.4.4.1, 6.4.4.2, 6.4.4.3, and 6.4.5 in general terms by ensuring that verification activities are complete.

OO.D.1.2.3 Supporting Information for Activities

Ensure that the parametric polymorphic operations acting on the substituted parameters implement the intended semantic behavior.

Each unique instantiation of a parameterized type or combination of types should be verified.

Note: Test cases may also be parameterized thereby reducing the burden of generating tests for each unique instance.

OO.D.1.3 Overloading

Overloading can aid in code development, readability, and maintenance, but care should be taken to avoid ambiguity or coincidental name collisions.

OO.D.1.3.1 Vulnerabilities

Overloading ambiguity occurs when a compiler performs implicit type conversions on parameters in order to select an acceptable match.

Implicit type conversion could lead to unintended subprogram selection. Although languages typically have specific rules to identify a ‘best’ match, the selection made by the compiler may not match the programmer’s intent. This problem is exacerbated in languages that treat overloaded operators differently from other subprograms.

Overloading potentially allows different developers to choose the same names coincidentally for subprograms that have semantically different behavior, leading to confusion.

OO.D.1.3.2 Related Guidance

These vulnerabilities are addressed by section OO.6.3.4.

OO.D.1.3.3 Supporting Information for Activities

Coding standards should address (this list is representative, not exhaustive):

1. The cases when overloading is allowed.

2. The use of explicit type conversions to avoid overloading ambiguity.

3. The use of the language's scoping mechanisms to reduce the likelihood of coincidental name collisions.

4. The use of semantically consistent naming for subprograms, including any predefined usages such as operators.

5. For languages that provide implicit type conversion, ensure the correct overloaded subprogram is executed.

OO.D.1.4 Type Conversion

Although type conversion may be convenient for the programmer, it can introduce unintended behavior. Lack of familiarity with the implicit conversion rules of a language is often a cause of coding problems.

OO.D.1.4.1 Vulnerabilities

The vulnerabilities are dependent on the nature of the type conversion. With a narrowing type conversion, data may be lost. With a downcast, the vulnerability depends on the implementation in a particular language. It may result in data corruption of the object itself or its neighbors in memory, incorrect behavior, or a run-time exception being thrown.

OO.D.1.4.2 Related Guidance

These vulnerabilities are addressed by section OO.6.3.4.

OO.D.1.4.3 Supporting Information for Activities

a. Ensure type conversions (including implicit type conversions) are safe, and implications are understood.

b. Ensure that upcasting and type widening retain type safety and can be performed implicitly.

c. When performing narrowing type conversions, explicit conversion should be used to convey to code reviewers that this result is desired by the programmer. The explicit conversion may also be accompanied by additional code (for example, a range check) to ensure that the narrowing conversion can be supported by the destination type.

d. When downcasting, ensure that the conversion is safe. Analysis, exception handling, or explicit guard code can be used to ensure safe conversion.

OO.D.1.5 Exception Management

The ability to throw (raise) exceptions within a method and to handle exceptions in the calling method is a common feature of most object-oriented languages. Some languages support both checked and unchecked exceptions. Checked exceptions should be included in the subprogram signature and therefore handled by the program. Unchecked exceptions are not part of the subprogram signature, and there is no assurance that they will be handled.

OO.D.1.5.1 Vulnerabilities

Potential vulnerabilities include that an exception may not be handled or that either no action or inappropriate actions may be performed when the exception is caught (handled). The program may be left in an inconsistent state as a result of an exception not being handled correctly.

OO.D.1.5.2 Related Guidance

These vulnerabilities are addressed by sections OO.4.2.n, OO.5.2.2.k, and OO.6.3.3.

OO.D.1.5.3 Supporting Information for Activities

The exception management strategy should define if exceptions are used, and if so, in which cases exceptions will be thrown (raised), where they should be caught (handled), and how they should be processed. If a language uses implicit checks where exceptions may be generated, such as range checks, bounds checks, divide-by-zero checks, and preconditions or postconditions checks, then the strategy should account for these exceptions. Exceptions should be considered as part of the class requirements and for verifying substitutability.

OO.D.1.6 Dynamic Memory Management (DMM)

Complex tasks may depend on the ability to allocate and deallocate objects dynamically. Many calculations result in the creation of objects whose lifespan is shorter than the duration of the program's execution. Since memory is limited, it may be necessary to reuse the memory consumed by such objects. Object-oriented programming tends to rely on dynamic memory management. This section identifies vulnerabilities and provides recommendations for dynamic memory management techniques.

OO.D.1.6.1 Vulnerabilities

There are several vulnerabilities that should be considered whenever any dynamic memory management technique is used. Such techniques include memory pooling, as well as stack, scope, and heap based memory management, both manual and automatic (garbage collection). For all techniques, one or more of the following vulnerabilities may exist:

a. Ambiguous references: An allocator returns a reference to live memory, for example, an object that is still reachable from program code. This could cause failure by allowing the program to use the given memory in an unintended manner.

b. Fragmentation starvation: An allocation request can fail due to insufficient logically contiguous free memory available.

c. Deallocation starvation: An allocation request can fail due to insufficient reclamation of unreferenced memory (for example, objects, or structures). This might also be caused by losing references.

d. Heap memory exhaustion: The simultaneous heap memory requirements of a program could exceed the memory available.

e. Premature deallocation: A memory fragment could be reclaimed while a live reference exists.

f. Lost update and stale reference: In a system where objects are moved to avoid fragmentation of memory, an update could be made to the old copy after the new copy has been created or to the new copy before it is initialized, or a read could be made from the old copy after the new copy has been created and changed or from the new copy before it has been initialized.

g. Time-bound allocation or deallocation: An application could encounter unexpected delay due to dynamic memory management.

OO.D.1.6.2 Related Guidance

These vulnerabilities are addressed by section OO.6.8.1 and section OO.6.8.2 items a through g.

OO.D.1.6.3 Supporting Information for Activities

There are three main techniques for managing dynamically allocated objects: object pooling, activation frame based object management, and heap based object management. Both activation frame based and heap based object management have important variants. Activation frame object management can either be directly linked to the call stack or use a more loosely scoped based mechanism. Heap based object management can be manual or automatic. Some techniques require a more substantial memory management infrastructure (MMI) than others, but the degree to which this infrastructure can fulfill the objectives varies accordingly as well. Each of these techniques has its own pros and cons, but needs to consider the issues stated above. Table OO.D.1.6.3 below illustrates to what extent the application developer needs to fulfill the objectives for each application as opposed to addressing them in the MMI once for all applications. The amount of memory an application uses can be adversely affected by retaining references to data that is no longer needed.

Table OO.D.1.6.3 Where Activities Are Addressed For DMM

Technique	Activities (OO.6.8.2)						
	a	**b**	**c**	**d**	**e**	**f**	**g**
Object pooling	AC	AC	AC	AC	AC	N/A	MMI
Stack allocation	AC	MMI	MMI	AC	AC	N/A	MMI
Scope allocation	MMI	MMI	MMI	AC	AC	MMI	MMI
Manual heap allocation	AC	AC*	AC	AC	AC	N/A	MMI
Automatic heap allocation	MMI	MMI	MMI	AC	MMI	MMI	MMI

AC = application, MMI = memory management infrastructure, N/A = not applicable, and * = difficult to ensure by either application or MMI.

OO.D.1.6.3.1 Object Pooling

Object pooling is a means of managing objects of the same type. An initial set of unused objects of a particular type is provided during startup. Each time a new object of the given type is needed, one is taken from the pool. Once an object is no longer needed, it is returned to the pool. Safety systems need not, in general, allow any given pool to grow or shrink unless it is combined with garbage collection.

Pooling has the advantage of providing fast allocation, but the disadvantage that the application developer needs to manage the pool manually. The application developer should ensure that at least the first five vulnerabilities listed above are addressed for each application. Usually, it is not necessary to provide for moving objects.

The application developer should make sure that each object that is no longer needed is in fact returned to its pool and that any object being returned to the pool no longer has references. In addition, the user needs to size the pools to prevent running out of memory prematurely, since pools are object specific. An application that ensures these properties addresses the applicable vulnerabilities.

OO.D.1.6.3.2 Activation Frame Management

For objects that are needed only in certain execution contexts, for example only in a given method and its called methods, a stack like allocation method could be used. The two variants of this approach vary in how tightly they are associated with a call's activation frame. Both need to ensure that no references are made to an object allocated in a frame or scope from outside that frame or scope, for example, parent frame of scope.

OO.D.1.6.3.3 Stack Frame

Stack frame object management uses the activation frame of a method to store local objects. This limits the extent to which frames can be shared between threads. It also reduces the danger of fragmentation. The application developer should consider the size of each stack required for object allocation and ensure that no references are made to an object stored on a stack other than from the same stack frame or a more deeply nested stack frame of the same stack.

OO.D.1.6.3.4 Scope Based

Scoped based object management is less strongly associated with activation frames. Several threads may enter a scope simultaneously. This increases the danger of fragmentation, since more care should be given to managing the backing store for storing scoped objects. The assignment rules of scoped memory ensure there are no dangling references.

OO.D.1.6.3.5 Heap Based Management

Heap based management is the most general approach. Memory can be reused more generally, but all vulnerabilities should be considered. Automatic management or garbage collection reduces the burden on the application developer, since the garbage collector is responsible for avoiding all vulnerabilities except item d in Table OO.D.1.6.3.

OO.D.1.6.3.6 Manual

Manual heap management requires the application to deallocate each object when and only when the object is no longer referenced. Usually objects are not moved, but this means that the application developer ensures that fragmentation does not cause allocations to fail. In general, the application developer should ensure that all vulnerabilities are avoided.

OO.D.1.6.3.7 Automatic (Garbage Collection)

Garbage collection enables the application developer to defer vulnerability analysis to the underlying execution environment. This means that a garbage collection can be validated once and reused, provided the execution environment is unchanged. When the execution environment is changed, the timing analysis should be redone. For a garbage collector, it should be verified that all objects no longer referenced by the application are returned to the free list, the free list is consolidated to ensure that any subsequent memory allocation will succeed, and no object is collected to which there is still a reference. The application developer should show that garbage collection activity does not interfere with the timeliness of the application and that enough memory is available for execution. In many garbage collectors, collection time increases as the memory in use approaches the available memory, so the heap size should include a safety margin to mitigate this effect.

OO.D.1.7 Virtualization

Virtualization refers to techniques for emulating one execution environment with another one. Some virtualization techniques, such as interpreters, are programs that provide this emulation. When verifying such programs, the data processed as instructions by the program should be verified as executable code (not data).

OO.D.1.7.1 Vulnerabilities

A vulnerability associated with some virtualization techniques is incorrectly categorizing programming instructions as data. Consequently, tracing may be neglected, requirements may be inadequate or missing, and verification may be insufficient.

OO.D.1.7.2 Related Guidance

These vulnerabilities are addressed by section OO.4.2.m.

OO.D.1.7.3 Supporting Information for Activities

The applicant should ensure that programming instructions processed as part of a virtualization technique, such as an interpreter, satisfy all the objectives of executable code. This verification should be in the context of the virtualization technique. A system may use layers of such virtualization techniques. When employed, each layer should be verified.

OO.D.2 General Issues

This section provides recommendations for issues that are not directly related to any OO-specific feature, and hence are integral issues that should always be considered regardless of the features used in the project.

OO.D.2.1 Traceability

DO-178C traceability objectives for object-oriented (OO) projects are the same as for traditional projects. However, there are additional considerations necessary in identifying and evaluating traceability artifacts for OO.

OO.D.2.1.1 OOT&RT Specific Considerations

Traceability may be complicated by a number of factors inherent in OO:

a. Requirements written from a functional viewpoint may not map well to an OO design, resulting in traceability that is not straightforward.

b. Inheritance introduces a relationship between the requirements of each subclass to the requirements for all of its super classes.

c. As discussed in DO-178C, for level A software, it is necessary to establish whether the object code is directly traceable to the Source Code. Items complicating source to object code traceability include constructors, destructors, inlining, run-time type checking, implicit type conversion, and other compiler-generated object code in support of the language.

d. The use of models (such as the UML) for capturing the behavior and structure of an OO software system is a common practice. When these models are used to capture the software system architecture and requirements, the traceability objectives associated with the software architecture and requirements apply. Typically, structural models are used to capture the software architecture and behavioral models are used to capture the software functional requirements. The types of models used should be specified in the design standards.

Note: One method for ensuring the correct development of models (such as UML) is the use of the Model-Based Development and Verification Supplement. However, this is not the only way to correctly develop these models.

OO.D.2.1.2 Related Guidance

Traceability activities are addressed by section OO.5.5. And, compiler inserted code is addressed by DO-178C section 6.4.4.2.b.

OO.D.2.1.3 Supporting Information for Activities

A step-wise refinement from system requirements to software requirements to classes, and the behavior they define, simplifies the traceability between requirements and implementation. To aid verification of type substitutability, traceability between the requirements of the subclass and the requirements of all of its superclasses should be established and verified.

OO.D.2.2 Structural Coverage

The intent of structural coverage is to provide evidence that the code structure was verified to the degree required for the applicable software level and provide a means to support demonstration of absence of unintended functions. Structural coverage analysis provides a means to confirm that the requirements-based tests exercised the code structure. As identified in DO-178C section 6.4.4.2, there are several aspects of structural coverage analysis to be considered:

a. Code coverage, including:

    1. Instruction or statement coverage.

    2. Decision coverage.

    3. Modified condition decision coverage.

b. Source Code to object code traceability

c. Coupling, including:

    1. Data coupling.

    2. Control coupling.

OOT uses control coupling to minimize data coupling. Dynamic dispatching is the method of control coupling introduced by OOT.

OO.D.2.2.1 OOT&RT Specific Considerations

Determining the degree of structural coverage may be complicated by the following language features:

a. Inheritance: The coupling between a method and the attributes of its class should be considered for the defining class and all subclasses. Inherited methods may result in inadequate behavior in the context of a subclass. For example, an inherited method may not adequately update the state of its subclass instance. Covering a method in the context of the superclass or subclass alone may not exercise the conditions dependent on the state space. Code coverage by itself does not take this data and control coupling into account.

b. Overriding: Overriding a method can violate the requirements of the superclass.

c. Dynamic dispatch: With dynamic dispatch, the specific method to be executed is determined at run-time from the set of classes of the static type available at the given dispatch point.

d. Subprogram overriding with static dispatch: Generally, this violates type consistency and will require pessimistic testing for the subprogram to ensure adequate coverage.

e. Parametric polymorphism: A compiler may generate unique code for each type instantiation.

OO.D.2.2.2 Related Guidance

Guidance for activities is provided in section OO.6.2 and DO-178C sections 6.4.4.1, 6.4.4.2, and 6.4.4.3.

OO.D.2.2.3 Supporting Information for Activities

The activities to be performed are determined by the ability to demonstrate if type consistency has been achieved. An acceptable means for demonstrating type consistency is by showing the software satisfies the LSP. This may be shown through test or formal methods.

It should be noted that some source-level code coverage techniques could merge structural coverage information coming from different instantiations inappropriately, yielding incorrect coverage results.

Structural coverage consists of the following unified steps:

- Execute the requirements-based tests to capture the data for structural coverage analysis.
- If type consistency is shown, then evaluate at least one instance of one of the classes that can occur at each call point.
- If the type consistency cannot be satisfied, then evaluate at least one instance of each class that can occur at each call point (pessimistic).
- Perform structural coverage analysis for the appropriate level, including data and control flow analysis.
- Consider all inherited, as well as explicit methods, for each class.

Recommendations for structural coverage analysis activities are described in the following sections.

OO.D.2.2.3.1 Code Coverage

Code coverage is performed for the applicable design assurance level. If type consistency was shown, then evaluate at least one instance of one of the classes that can occur at each call point. Otherwise, pessimistic testing is needed to ensure adequate coverage.

OO.D.2.2.3.2 Source Code to Object Code Traceability

The objective regarding traceability between object code and Source Code is unchanged. If multiple dynamic dispatches are possible through a call point, then the trace between object code to Source Code should be covered for all possible dispatches.

OO.D.2.2.3.3 Data Coupling

Data coupling describes the use of shared data by the software components via global data and parameter data. For OO projects, the use of data encapsulation as part of class definitions and the use of methods to manipulate the data reduce the use of global data.

Data coupling analysis should include an examination of the class structure and inheritance hierarchy.

OO.D.2.2.3.4 Control Coupling

Control coupling provides a measure of completeness of the integration process. The approach should be based on the requirements-based testing using the fully linked executable image. For OO projects, the control coupling may be less explicit due to the use of polymorphism and dynamic dispatch. Control coupling analysis should include an examination of the class structure and inheritance hierarchy (see OO.6.2.f).

OO.D.2.3 Component-Based Development

Component-based development is intended to provide developers with the ability to create new software systems using software artifacts designed or adapted for reuse. This reusable software may be comprised of procedures, methods, classes, packages and frameworks, and so on. Component-based development allows for combining new and existing components to fulfill system requirements. Inheritance, aggregation, subtyping, and the use of templates are common approaches to reusing components.

Reusable software often contains more functionality than required by the current system being certified. If this extra functionality results in extra code in the system itself, then there may be unused code to consider. Unused code will likely be present in any application that uses general-purpose software components and libraries, such as Commercial Off-The-Shelf (COTS) software libraries provided with a compiler, operating system or run-time environment, or OO development frameworks.

Components rely on proper abstractions that are intended to be kept intact whether or not they are fully used in a given system. Complete verification data for the component is leveraged to support system verification.

OO.D.2.3.1 Vulnerabilities

a. Requirements mismatch between the component and system. Component requirements are often developed outside the context of the system in which they will be used. Such requirements may be under-specified for the system in that they do not fully satisfy the intended system functionality or they may be over-specified and therefore provide functionality not required by the system.

b. The life cycle data for the component may be developed to different standards than the life cycle data for the system.

c. Error management mismatch. Error management might require a consistent policy on the overall system. Components coming from different sources might have been developed with different error management strategy in mind. For example, if the overall system error management strategy involves the handling of exceptions, a component which returns a status value rather than throwing an exception may need to have a wrapper around it which examines the status value and throws the exception if so indicated.

d. Resource management mismatch. Resource management (such as heap, stack, processor cycles, synchronization) might require consistent strategies over the entire system. Components coming from different sources might have conflicting resource needs or strategies incompatible with the system under construction.

e. Presence of deactivated code. Some parts of a reusable component (such as attributes, methods, subprograms, or complete classes) may not be needed in the context of a given system and thus will not be exercised during its normal execution. In particular, deactivated code resulting from the reuse of a component may make it difficult to identify loss of intended function and potential unintended behavior.

f. Verification of integrated system component testing, including data coupling analysis and control coupling analysis, may be difficult to achieve without internal knowledge of the component.

OO.D.2.3.2 Related Guidance

Guidance is provided in sections OO.4.2.n and OO.5.2.2.l.

OO.D.2.3.3 Supporting Information for Activities

Special attention should be paid to these areas:

- Requirements: Any component requirements that do not trace back to the system requirements should be identified (See Annex OO.A Table OO.A-2 Objectives 2 and 5).

- Reuse: When reusing a component, the original requirements for that component should be provided to the system safety assessment. The effects of these requirements on safety-related requirements are determined by the system safety assessment process. (DO-178C sections 5.1.2.j and 5.2.1.b)

- Architecture: The component architecture should be well defined, including its error management and resource management policies. Compatibility with the global system architecture should also be addressed.

- Deactivated code: Minimize the amount of deactivated code that is contained in the final executable code.

- Completeness: Ensure verification is complete for deactivated elements of a component.

A verification strategy for the integration of the components into the system should be established. Verification activities need to be identified to check the adequate integration of these components into the system, including the analysis of component and system data and control flow coupling.

The following recommendations may support the reuse of software components. These recommendations are intended to aid in determining if a component may be reused from another system. The recommendations identify attributes of a component that need to be reviewed in the context of the using system.

a. Planning

   1. Identify all reused components and a plan for deactivation of unused component parts within the Plan for Software Aspects of Certification (PSAC). The PSAC should describe the plan to use these components.

b. Requirements

   1. Review the reused component requirements against the requirements for the using system. This review should identify any mismatches between the component and the system using the component and the intended means to resolve the mismatch.

c. Life cycle data

   1. Identify available life cycle artifacts (for example, high-level requirements, architecture, low-level requirements, code, traceability, and verification data) and show compliance to DO-178C and this supplement objectives for each component.

   2. As different development standards (that is, requirement, design, or code standards) may be used between the reused component and the using system, an analysis of the standards should be performed to determine acceptability of the reused artifacts or if additional activities need to be performed.

d. Error Management

   1. Review the error handling and reporting behavior of the reused component to determine its compatibility with the error management scheme of the using system. Determine how mismatched error management schemes will be addressed in the using system.

   2. Specify the intended error handling behavior and define the steps necessary for the reused component to conform.

e. Resource Management

   1. Define the resource requirements for the reused component.

   2. Verify adequate resources are available in the using system.

f. Verification

   1. Analyze the using system to determine that the deactivated code within a component cannot be activated.

   2. Identify any reused component behavior that could adversely affect the system implementation (for example, vulnerabilities, partitioning requirements, hardware failure effects, and requirements for redundancy, data latency, and design constraints) for correct component operation.

   3. Ensure the integration test strategy is sufficient to verify the behavior of the reused component in the using system's context.

   4. Determine the global data defined or used by the component as part of the data coupling analysis.

   5. Analyze the control coupling between the system and reused component.

6. The verification data should include a list of test cases and procedures affected by any settable parameters of a component.

OO.D.2.4 Resource Analysis

As it is essential for safety-critical systems to have adequate memory, processor, and network resources available in order for them to accomplish their tasks in a timely manner, the allocation and consumption of these resources is of fundamental importance in achieving operational success for these systems.

Dynamic memory allocation, typically stack and heap usage, can lead to free memory exhaustion and improper behavior can lead to memory corruption. Throughput requirements can exceed the ability of the processor to complete all necessary tasks in a timely manner. Network contention or capacity restrictions can disrupt necessary communication between separate components. All of these situations need to be considered and allowance made for such events occurring outside of expected operating ranges. Of these resources, the use of OO programming is most likely to have an impact on memory usage and throughput and the necessary analysis to ensure the correctness of the usage of these resources.

Modern programming practices, including OO programming, have resulted in productivity improvements and potentially in safety improvements. The principal means by which these increases in productivity have been achieved is by increasing the abstraction level for the programmer; moving the act of programming away from a focus on the low-level activities of the processor and toward a higher-level view where the programming language and environment relieve the programmer of the responsibility of managing many of these low-level details.

This comes with some additional design and verification constraints, however, as the programmer no longer takes direct responsibility for fundamentally important aspects such as memory allocation and reuse and therefore has less control over this important resource. Additionally, the very nature of OO programming places a much greater emphasis on the use of dynamic memory allocation and its subsequent deallocation and return to the pool of available memory, thereby making memory allocation and reuse correspondingly more important. While OO programming can be done without the use of dynamic memory, such usage diminishes the very benefits for which OO programming has become so successful.

When memory management is done automatically, the programmer is no longer directly responsible for memory allocation and reuse. This eliminates a category of errors in the memory management process, but requires a more complex run-time system. Just as the use of higher-level languages has largely supplanted the use of assembly language in safety-critical systems, the use of automatic memory management systems is supplanting purely manual techniques. Use of these techniques has a direct impact on resource analysis.

OO.D.2.4.1 Annotations and Static Analysis

Before discussing resource analysis, a review of techniques often used to support such analysis is provided.

The use of annotations can improve the analysis of maximal resource usage. By expressing the programmer's intent and expectations explicitly in a manner which can be checked statically, the analysis can benefit from this information, achieving a more realistic result. In particular, expressing annotations in terms of attributes of the objects in the program rather than relying on static values makes them more generally applicable. For instance, an annotation providing the bound on a loop iterating through a collection is better expressed as a function of the size of the collection than as a hard-coded constant value.

Resource analysis of programs with OO code is more complex than those with purely procedural code. In particular, the increased control coupling of dynamic dispatch means that the control flow is more dependent on the data flow in a program. For this reason, more accurate analysis can be obtained using data flow analysis to supplement control flow analysis. Additional accuracy can be obtained by including pointer analysis. These techniques can also simplify resource analysis of programs using dynamic memory management.

OO.D.2.4.2 Memory Usage Analysis

Memory usage is typically viewed as being in one of four forms: program code, static, stack, and heap memory. As OO programming has not been shown to have a significant impact on program and static data memory usage, other than perhaps to diminish its overall use in an OO program, these memory usages will not be discussed further. Though there are other kinds of memory usage, such as scoped and immortal memory, they can be viewed as variants of stack and heap memory.

Stack memory is the term used for memory allocated to a particular subprogram for use during that subprogram's execution. It uses a first-in-last-out allocation strategy making analysis of stack memory much easier than the case for heap memory. The problem of the recovery of and reuse of the stack memory allocated to a particular subprogram once the subprogram has completed execution is well understood and widely accepted as being amenable to automatic handling by a high-level programming language and its run-time environment.

Heap memory is the term used for memory allocated for an entity, the existence of which is not bound by the activity or scope of the subprogram in which it was initially allocated. The impact of this is that the deallocation of such memory is, in general, not performed in the same context in which it was allocated, making its correct implementation a significantly greater problem than that of stack memory allocation and deallocation. Although garbage collection has been around for more than fifty years, it is only comparatively recently that heap memory has been viewed as being amenable to being automatically handled by a high-level programming language and its run-time environment, at least in the context of a real-time system.

OO.D.2.4.2.1 Stack Usage

Software for safety-critical systems often contains multiple threads of control. The most common technique for providing temporary storage for subprograms executing in a multiply threaded system is to allocate a separate stack for each thread. Such allocation can be manual or as part of a wider memory management system handled by the programming language and its run-time environment.

Analyzing stack usage for OO software is essentially the same as for procedural software, with a few extra considerations introduced. Such analysis consists of determining how much stack memory each subprogram can maximally allocate and which subprograms can be executing and therefore consuming stack space (for a particular thread's stack) at any given time. The use of OO programming adds the complication of having to consider the ramifications of dynamic dispatch which makes determining which methods are (or may be) invoked at any particular call point more complex than that of procedural software. Recursion also complicates the analysis of stack usage but not in a manner which is any different for OO software from that found in procedural software. Some compilers transform tail recursion instances into loops. Aside from this case, all uses of recursion should be shown to be resource bounded.

Manual calculation of a thread's stack usage, although theoretically possible, is widely considered to be too error prone to be considered practical for any program of significant size. Realistically, such analysis requires tools to automate the determination of program flow and its associated call sequences in order to calculate maximal stack usage for the thread. For OO programs, this analysis will be complicated by the need to determine at each dispatch point the set of methods inherited from a superclass or redefined by a subclass which might be invoked at that dispatch point. Additionally, techniques such as escape analysis used to minimize heap memory allocations and often found in sophisticated OO language run-time systems will further impact the analysis of stack usage.

Such analysis may result in an overly conservative determination of maximal stack usage since often only a subset of available subclasses can actually be used at any given dispatch point. In this case global data flow analysis can be used to provide a more accurate limit on stack size. Global data flow analysis can also be useful in determining the limits of recursion in a particular thread.

OO.D.2.4.2.1.1 Related Guidance

Guidance is provided for stack usage in section OO.6.3.4 and DO-178C section 6.4.3.a.

OO.D.2.4.2.1.2 Supporting Information for Activities

Potential anomalies associated with stack usage such as data corruption, stack overflow, and underflow are unchanged by the use of OO techniques so all recommendations pertaining to procedural language stack usage still apply. Care should be taken to address the additional difficulties of OO stack usage discussed above, such as dispatching and object allocation on the stack coming from escape analysis. As providing sufficiently accurate stack usage analysis by manual methods has shown itself to be generally unreliable, automated techniques including global data flow and pointer analysis should be considered. Stack instrumentation, for example, run-time stack monitoring, should be used to detect stack underflow and overflow conditions in order to minimize the probability of these conditions arising during flight operations. Stack overflow and underflow should be detectable. If they occur, appropriate action, such as reducing operations to a degraded mode, should be taken.

OO.D.2.4.2.2 Heap Usage

In a system that does not dynamically allocate objects after the system has been initialized, it is sufficient to demonstrate that the initialization code does not require more heap memory than is available; however, heap analysis for dynamically allocated memory is more complex. It is not sufficient to track all allocations; the lifespan of all objects allocated on the heap should be understood. Ideally, there would be a method to determine precisely the lifespan of each object. In practice, this is intractable, since exact lifespans depend on data the program receives from its environment; therefore, a safe approximation method or extensive testing is needed.

Ideally, formal methods would be used to provide such an approximation. Techniques such as escape analysis and data flow with pointer analysis can help. Escape analysis can determine which objects are used only in a stack-like manner and whose lifespan is related to the call stack behavior of a program. Data flow and pointer analysis can provide information about when references are discarded, since it is difficult to have sufficient precision to model the heap accurately.

Alternatively, system integration testing can be used for ensuring that an application has sufficient memory. The heap infrastructure should have a means of determining the current amount of memory in use. This can be used to track memory usage over time under expected load and data situations. It is important in such tests to dimension the input data larger than what could be reasonably expected in actual operation.

OO.D.2.4.2.2.1 Manual Reclamation of Memory

For as long as there has been heap memory there has been manual allocation and reclamation of heap memory. There are several variations of this technique, the most common being heap management using an allocation and a free routine. Others include object pooling and scoped memory.

For the C community, these allocation and free routines are the familiar: malloc() and free(). Manual allocation and reclamation of memory has been shown to be chronically difficult to manage properly as the allocation and deallocation are typically separate and therefore inherently difficult to ensure that accurate pairing has been made of all allocations with a sufficient set of deallocations. This frequently leads to memory corruption through dangling pointers (pointers to memory that has already been freed) and memory leaks (allocated memory which, even after no longer of any value, still remains allocated). While extremely thorough testing and code review can address these problems they are, in general, error prone; therefore, the use of malloc() and free() for dynamic memory management is strongly discouraged.

Object pooling is a variation on heap management using manual allocation and deallocation. As this is a manual process, errors can still occur with objects being returned to a pool prematurely or failing to return an object to its pool before all references to it are lost. Fragmentation within the heap is exchanged for fragmentation between heaps. If this technique is used, all the vulnerabilities of manual heap management should still be addressed and care should be taken to ensure that all pools are properly sized, so one pool is not exhausted while memory is available in other pools.

Scoped memory tries to use a stack-like allocation paradigm to manage memory usage. This has the advantage, that even lost pointers can be free and there is generally a mechanism to ensure that objects are not deallocated to which references still exist. Still the management of the memory blocks themselves can become an issue and additional run-time checks are necessary to prevent premature deallocation, by not allowing references into a more deeply nested scope.

The correctness of these methods cannot be determined solely by their implementation. They rely too heavily on how they are used. Correctness can only be determined in the scope of a particular application.

OO.D.2.4.2.2.2 Automatic Reclamation of Memory

Automatic reclamation of memory, commonly referred to as garbage collection, is just one technique of managing heaped memory, but since it could improve system integrity at the expense of additional run-time system code, it deserves special attention. This improvement in system integrity is provided by ensuring that reference integrity is maintained on the heap. In other words, only objects that are no longer referenced are returned to the free list and no memory is lost by no longer having a reference to it. In order to accomplish this, a garbage collector needs to be precise; that is it knows what memory it is responsible for and can find all references to objects within this memory.

There are some garbage collectors for languages, such as C and C++, which allow pointer manipulation, but these collectors cannot be precise, since a program can break reference integrity by, for example, casting a pointer to a void pointer. These collectors need to also use heuristics to decide what is a pointer and what is not a pointer on the program stacks, since casting can be used to convert between pointers and other data types. Such heuristics can be misled by, for example, finding a double on a stack and within its mantissa bit pattern finding a value that otherwise appears to be a pointer. As a result, these collectors can guarantee neither that objects with references are not collected nor that objects are not lost.

It should be clear, that only a precise garbage collector is usable in safety-critical applications, and only a real-time garbage collector is usable for real-time applications. A precise garbage collector is one that never loses objects. A real-time collector is one that can provide response guarantees for critical tasks. This usually means that the collector works incrementally instead of all at once.

The analysis of any memory management technique should cover the objective of section OO.6.8 and is discussed in Annex OO.D.1.6. A correctly implemented precise garbage collector can ensure that most of the objective of section OO.6.8 is fulfilled. Only objects that are no longer reachable and all objects that are no longer reachable are returned to the free list for reallocation. For a garbage collector to be precise, the programmer may not have any ability to manipulate pointers directly. For this reason, it is impossible to produce a precise garbage collector for languages such as C and C++. Furthermore, a garbage collector should know the sizes and reference addresses for all objects, so that all references from an object can be followed and freed objects can be made available for reuse. Thus, through the correct implementation of reachability analysis, activities OO.6.8.2.a, OO.6.8.2.c, and OO.6.8.2.e. are completed. Most precise garbage collectors have some mechanism to deal with fragmentation which should suffice for activities OO.6.8.2.b and OO.6.8.2.d, whereas OO.6.8.2.f is only an issue in systems that actually move objects.

Performing activity OO.6.8.2.g, which is only a prerequisite for real-time tasks, requires a more rigorous approach than used in standard garbage collectors, since correctness depends not only on the function of the garbage collector, but also when such operations occur. Care should be taken to ensure that the garbage collector does not inhibit the timely execution of critical tasks. This topic is discussed in detail in sections OO.D.2 and OO.D.2.3.2 of this Annex.

Heap memory exhaustion is the core issue for heap usage. In order to ensure that a program that uses garbage collection does not run out of memory prematurely, the garbage collector needs to be able to run two full cycles before the program can allocate the remaining free memory, where a cycle indicates a full traversal of the heap. The reason that two cycles need to run is to account for objects allocated while the garbage collector is running. This is dependent both on the amount of free memory and on the allocation rate.

As an example, there are three common approaches to real-time garbage collection: periodic, slack-based, and work-based. Certainly, others will appear, but these three can be used to illustrate the main points that need to be considered. In any case, garbage collection either has its own thread context or operates in the context of the allocating threads.

Mark and sweep is the oldest and simplest of the basic techniques. This technique uses separate phases to determine which objects are still reachable and returns objects that are no longer reachable to the pool of free memory. Typically, the algorithm is divided into four phases: root scan, mark, sweep, and compaction. In the root scan phase, all pointers into the heap are found. This typically involves tracing the thread stacks and static data for pointers. The mark phase identifies all objects that are reachable from these entry points. The sweep phase returns all unreachable objects to the free list. Finally, the compaction phase rearranges objects to reduce fragmentation so that the largest possible object can be allocated so long as enough memory is available. Other garbage collection algorithms are essentially optimizations on these steps, that is, all reachable objects remain, and all unreachable objects are made available for reallocation. It is important to note that this process can be accomplished incrementally; that is, the marking of live objects on the heap can be conceptually concurrent with the allocation of new objects on the heap. The use of read and/or write barrier algorithms makes this possible.

In periodic and slack-based collectors, the garbage collector runs in its own thread of control. In the first case, the garbage collector runs at a set time for a set duration. Real-time tasks are scheduled around the garbage collector. In the second case, the collector runs at a low priority and the system needs to allow enough slack time (time when no other task is running) for the garbage collector to do its work. The analysis should have information about both the amount of free memory and the allocation rate to ensure that the garbage collector thread runs often and long enough to ensure objects are reclaimed quickly enough to prevent the system from failing to allocate memory when there is still unused memory in the system. Slack-based garbage collectors generally require a full schedulability analysis to ensure that there will always be enough slack time for the collector to recycle memory in a timely manner.

A work-based collector does not have a separate thread of control. Instead, it runs some number of steps whenever memory is allocated. In other words, an application task pays for memory allocation by doing a little bit of garbage collection. This means that the collector automatically tracks the allocation rate since when the allocation rate rises, so does the amount of garbage collection work being done, and when the allocation rate falls, so does the amount of garbage collection work. The analysis needs the amount of free memory, but not the allocation rate, to ensure that two cycles complete before memory is exhausted.

The main point is to demonstrate not only the maximal memory use, but also some means of obtaining or tracking the allocation rate. In a system that has no means of automatically tracking the allocation rate, the allocator will prematurely fail, that is, signal out of memory before the memory is fully exhausted. All activities under OO.6.8.2 (see OO.D.2.4.2.2.4 for additional information), except item d and to some extent item g, can be done once for all programs using a given garbage collector. Timeliness (item g) will be addressed in Annex OO.D.2.4.3.

OO.D.2.4.2.2.3 Related Guidance

Guidance is provided for heap usage in section OO.6.8.2.

OO.D.2.4.2.2.4 Supporting Information for Activities

All dynamic memory management techniques need to fulfill the objective of section OO.6.8. This means that any dynamic memory management technique used in avionics systems need to ensure that:

a. The allocator returns a memory reference for which no other reference exists (exclusivity).

b. Memory is organized such that any necessary allocation request will succeed when there is sufficient free memory available.

c. Allocated memory which is no longer referenced is reclaimed before that memory is needed for reuse.

d. There is sufficient memory to accommodate all simultaneously live objects.

e. Reference consistency is maintained, that is, each object is unique, and is only viewed as that object.

f. When moving objects, the move of an object is atomic with respect to all references to the said object.

g. That all memory operations are deterministic, that is, have specified time bounds.

For memory analysis, once the technique chosen is demonstrated to work correctly for the application, show that the system has sufficient memory for executing the application. Ideally, use formal methods to demonstrate that the system does not need more memory than what is available. Barring this, instrumentation, for example, run-time heap monitoring, should be used to show that heap memory does not exceed a peak value. In any case, the system should recognize when memory is about to be exhausted and take appropriate action, such as reducing operations to a degraded mode, should this condition occur.

OO.D.2.4.3 Timing Analysis

Almost all safety-critical avionics applications are also time-critical; that is, they have real-time deadlines that need to be met. That means that a particular computation needs to finish within a specified amount of time. It is incumbent on the applicant to show that the time-critical code will always complete within its deadline. To do this, ensure that both the time-critical subprogram never requires more time to execute than it has available and that this subprogram receives sufficient time on the CPU to finish before the deadline is reached. Typically, the analysis necessary to demonstrate that time-critical subprograms meet their deadlines is to break down the problem into determining the worst-case execution time (WCET) of the subprogram and determining the scheduling of the subprogram.

OO.D.2.4.3.1 Dynamic Memory Management

n general, OO languages are more dependent on dynamic memory management than procedural languages. Dynamic memory management techniques have an impact on the execution time and some even have an impact on the scheduling of time-critical subprograms. In general, the time needed to allocate and free objects should be considered. There must be understanding of this behavior as part of a complete timing analysis of an application.

In most cases, the timing requirements of dynamic memory management can be subsumed by WCET analysis.

When allocation and free operations (or garbage collection work) is done directly in an application thread, the developer can know exactly where these operations take place. When analyzing the WCET, these operations are simply part of that calculation. There is no direct impact on scheduling.

On the other hand, some real-time garbage collection and compaction techniques require a separate task (also known as thread of execution) to do their work. In this case, both timing and scheduling analysis should be done to ensure proper system responsiveness. There needs to be an understanding of not only when this task is scheduled, but also how much time it takes, and at what granularity the task can be interrupted.

As opposed to a standard garbage collector, which cannot be used for time-critical applications, a real-time garbage collector algorithm should be incremental so that its work can be interleaved with the rest of the application. It should complete two complete cycles in the time it takes to use up free memory via allocation. Two cycles are needed to account for objects that are allocated in parallel within one cycle of the garbage collector.

OO.D.2.4.3.2 Worst-Case Execution Time Analysis

WCET analysis is only marginally impacted by the use of OOT&RT. Both method dispatch and dynamic memory management can have an effect on the timing of a system. The dynamic method dispatch mechanism determines the call overhead that is needed to consider in WCET analysis. In the case of single inheritance, or multiple inheritance using sparse matrices, the dispatch time can be constant, usually a single pointer dereferenced. Otherwise, a table lookup may be necessary. Then this lookup time should be included in the WCET analysis. The defined time bounds on relevant memory management routines should also be considered. These include, but are not limited to, memory allocation and memory deallocation.

OO.D.2.4.3.3 Scheduling

Scheduling analysis can also be impacted by some OOT and related techniques. If the memory management subsystem has tasks that run in dedicated threads, then this may need to be considered during scheduling analysis. Virtualization techniques may also have impact on scheduling, particularly when the virtualization layer incorporates its own scheduler. Features which require scheduling need to be documented and included in the scheduling analysis.

OO.D.2.4.3.4 Related Guidance

Guidance is provided for timing analysis in section OO.6.8.2 and DO-178C section 6.4.2.2.

OO.D.2.4.3.5 Supporting Information for Activities

WCET and scheduling verification are basically the same as for procedural languages, but additional aspects of OO technology and dynamic memory management should be considered. At each call point, the run-time of the method of the static type in the WCET, and the method from each of the possible dynamic types that can be substituted for the static type should be considered in the WCET. For memory management, all in-code memory operations should be considered as part of WCET and, if there is a separate thread for memory management, such as with a garbage collector thread, that thread should be considered as part of the task load for scheduling as well.

APPENDIX OO.A – COMMITTEE MEMBERSHIP

EXECUTIVE COMMITTEE MEMBERS

Jim Krodel, Pratt & Whitney	SC-205 Chair
Gérard Ladier, Airbus/Aerospace Valley	WG-71 Chair
Mike DeWalt, Certification Services, Inc./FAA	SC-205 Secretary (until March 2008)
Leslie Alford, Boeing Company	SC-205 Secretary (from March 2008)
Ross Hannan, Sigma Associates (Aerospace)	WG-71 Secretary
Barbara Lingberg, FAA	FAA Representative/CAST Chair
Jean-Luc Delamaide, EASA	EASA Representative
John Coleman, Dawson Consulting	Sub-group Liaison
Matt Jaffe, Embry-Riddle Aeronautical University	Web Site Liaison
Todd R. White, L-3 Communications/Qualtech	Collaborative Technology Software Liaison

SUB-GROUP LEADERSHIP

SG-1 – Document Integration

Ron Ashpole, SILVER ATENA	SG-1 Co-chair
Tom Ferrell, Ferrell and Associates Consulting	SG-1 Co-chair (until March 2008)
Marty Gasiorowski, Worldwide Certification Services	SG-1 Co-chair (from March 2008)
Tom Roth, Airborne Software Certification Consulting	SG-1 Secretary

SG-2 – Issues and Rationale

Ross Hannan, Sigma Associates (Aerospace)	SG-2 Co-chair
Mike DeWalt, Certification Services, Inc./FAA	SG-2 Co-chair (until March 2008)
Will Struck, FAA	SG-2 Co-chair (until March 2009)
Fred Moyer, Rockwell Collins	SG-2 Co-chair (from April 2009)
John Angermayer, Mitre	SG-2 Secretary

SG-3 – Tool Qualification

Frédéric Pothon, ACG Solutions	SG-3 Co-chair
Leanna Rierson, Digital Safety Consulting	SG-3 Co-chair
Bernard Dion, Esterel Technologies	SG-3 Co-secretary
Gene Kelly, CertTech	SG-3 Co-secretary (until May 2009)
Mo Piper, Boeing Company	SG-3 Co-secretary (from May 2009)

SG-4 – Model-Based Development and Verification

Pierre Lionne, EADS APSYS	SG-4 Co-chair
Mark Lillis, Goodrich GPECS	SG-4 Co-chair
Hervé Delseny, Airbus	SG-4 Co-chair
Martha Blankenberger, Rolls-Royce	SG-4 Secretary

SG-5 – Object-Oriented Technology

Peter Heller, Airbus Operations GmbH	SG-5 Co-chair (until February 2009)
Jan-Hendrik Boelens, Eurocopter	SG-5 Co-chair (Feb 2009 to August 2010)
James Hunt, aicas	SG-5 Co-chair (from August 2010)
Jim Chelini, Verocel	SG-5 Co-chair (until Oct 2009)
Greg Millican, Honeywell	SG-5 Co-chair (Oct 2009 to August 2011)
Jim Chelini, Verocel	SG-5 Co-chair (from August 2011)

SG-6 – Formal Methods

Duncan Brown, Aero Engine Controls (Rolls-Royce)	SG-6 Co-chair
Kelly Hayhurst, NASA	SG-6 Co-chair

SG-7 – Special Considerations and CNS/ATM

David Hawken, NATS	SG-7 Co-chair (until June 2010)
Jim Stewart, NATS	SG-7 Co-chair (from June 2010)
Don Heck, Boeing Company	SG-7 Co-chair
Leslie Alford, Boeing Company	SG-7 Secretary (until March 2008)
Marguerite Baier, Honeywell	SG-7 Secretary (from March 2008)

RTCA Representative:

Rudy Ruana	RTCA Inc. (until September 2009)
Ray Glennon	RTCA Inc. (until March 2010)
Hal Moses	RTCA Inc. (until August 2010)
Cyndy Brown	RTCA Inc. (until August 2011)
Hal Moses	RTCA Inc. (from August 2011)

EUROCAE Representative:

Gilbert Amato	EUROCAE (until September 2009)
Roland Mallwitz	EUROCAE (from October 2009)

EDITORIAL COMMITTEE

Leanna Rierson, Digital Safety Consulting	Editorial Committee Chair
Ron Ashpole, SILVER ATENA	Editorial Committee
Alex Ayzenberg, Boeing Company	Editorial Committee
Patty (Bartels) Bath, Esterline AVISTA	Editorial Committee
Dewi Daniels, Verocel	Editorial Committee
Hervé Delseny, Airbus	Editorial Committee
Andrew Elliott, Design Assurance	Editorial Committee
Kelly Hayhurst, NASA	Editorial Committee
Barbara Lingberg, FAA	Editorial Committee
Steven C. Martz, Garmin	Editorial Committee
Steve Morton, TBV Associates	Editorial Committee
Marge Sonnek, Honeywell	Editorial Committee

COMMITTEE MEMBERSHIP

Name	Organization
Kyle Achenbach	Rolls-Royce
Dana E. Adkins	Kidde Aerospace
Leslie Alford	Boeing Company
Carlo Amalfitano	Certon Software, Inc
Gilbert Amato	EUROCAE
Peter Amey	Praxis High Integrity Systems
Allan Gilmour Anderson	Embraer
Håkan Anderwall	Saab AB
Joseph Angelo	NovAtel Inc, Canada
John Charles Angermayer	Mitre Corp
Robert Annis	GE Aviation
Ron Ashpole	SILVER ATENA
Alex Ayzenberg	Boeing Company
Marguerite Baier	Honeywell
Fred Barber	Avidyne
Clay Barber	Garmin International
Gerald F. Barofsky	L-3 Communications
Patty (Bartels) Bath	Esterline AVISTA
Brigitte Bauer	Thales
Phillipe Baufreton	SAGEM DS Safran Group
Connie Beane	ENEA Embedded Technology Inc
Bernard Beaudouin	EADS APSYS
Germain Beaulieu	Independent Consultant
Martin Beeby	Seaweed Systems
Scott Beecher	Pratt & Whitney
Haik Biglari	Fairchild Controls
Peter Billing	Aviya Technologies Inc
Denise Black	Embedded Plus Engineering
Brad Blackhurst	Independent Consultant
Craig Bladow	Woodward
Martha Blankenberger	Rolls-Royce
Holger Blasum	SYSGO
Thomas Bleichner	Rohde & Schwarz
Don Bockenfeld	CMC Electronics
Jan-Hendrik Boelens	Eurocopter
Eric Bonnafous	CommunicationSys
Jean-Christophe Bonnet	CEAT
Hugues Bonnin	Cap Gemini
Matteo Bordin	AdaCore
Feliks Bortkiewicz	Boeing
Julien Bourdeau	DND (Canada)
Paul Bousquet	Volpe National Transportation Systems Center
David Bowen	EUROCAE
Elizabeth Brandli	FAA
Andrew Bridge	EASA
Paul Brook	Thales
Daryl Brooke	Universal Avionics Systems Corporation
Duncan Brown	Aero Engine Controls (Rolls-Royce)
Thomas Buchberger	Siemens AG

Name	Organization
Brett Burgeles	Consultant
Bernard Buscail	Airbus
Bob Busser	Systems and Software Consortium
Christopher Caines	QinetiQ
Cristiano Campos Almeida De Freitas	Embraer
Jean-Louis Camus	Esterel Technologies
Richard Canis	EASA
Yann Carlier	DGAC
Luc Casagrande	EADS Apsys
Mark Chapman	Hamilton Sundstrand
Scott Chapman	FAA
Jim Chelini	Verocel
Daniel Chevallier	Thales
John Chilenski	Boeing Company
Subbiah Chockalingam	HCL Technologies
Chris Clark	Sysgo
Darren Cofer	Rockwell Collins
Keith Coffman	Goodrich
John Coleman	Dawson Consulting
Cyrille Comar	AdaCore
Ray Conrad	Lockheed Martin
Mirko Conrad	The MathWorks, Inc.
Nathalie Corbovianu	DGAC
Ana Costanti	Embraer
Dewi Daniels	Verocel
Eric Danielson	Rockwell Collins
Henri De La Vallée Poussin	SABCA
Michael Deitz	Gentex Corporation
Jean-Luc Delamaide	EASA
Hervé Delseny	Airbus
Patrick Desbiens	Transport Canada
Mike DeWalt	Certification Services, Inc./FAA
Mansur Dewshi	Ultra Electronics Controls
Bernard Dion	Esterel Technologies
Antonio Jose Vitorio Domiciano	Embraer
Kurt Doppelbauer	TTTech
Cheryl Dorsey	Digital Flight
Rick Dorsey	Digital Flight
John Doughty	Garmin International
Vincent Dovydaitis III	Foliage Software Systems, Inc.
Georges Duchein	DGA
Branimir Dulic	Transport Canada
Gilles Dulon	SAGEM DS Safran Group
Paul Dunn	Northrop Grumman Corporation
Andrew Eaton	UK CAA
Brian Eckmann	Universal Avionics Systems Corporation
Vladimir Eliseev	Sukhoi Civil Aircraft Company (SCAC)
Andrew Elliott	Design Assurance
Mike Elliott	Boeing Company
Joao Esteves	Critical Software
Rowland Evans	Pratt & Whitney Canada
Louis Fabre	Eurocopter

Name	Organization
Martin Fassl	Siemens AG
Michael Fee	Aero Engine Controls (Rolls-Royce)
Tom Ferrell	Ferrell and Associates Consulting
Uma Ferrell	Ferrell and Associates Consulting
Lou Fisk	GE Aviation
Ade Fountain	Penny and Giles
Claude Fournier	Liebherr
Pierre Francine	Thales
Timothy Frey	Honeywell
Stephen J. Fridrick	GE Aviation
Leonard Fulcher	TTTech
Randall Fulton	Seaweed Systems
Francoise Gachet	Dassault-Aviation
Victor Galushkin	GosNIIAS
Marty Gasiorowski	Worldwide Certification Services
Stephanie Gaudan	Thales
Jean-Louis Gebel	Airbus
Dries Geldof	BARCO
Dimitri Gianesini	Airbus
Jim Gibbons	Boeing Company
Dara Gibson	FAA
Greg Gicca	AdaCore
Steven Gitelis	Lumina Engineering
Ian Glazebrook	WS Atkins
Santiago Golmayo	GMV SA
Ben Gorry	British Aerospace Systems
Florian Gouleau	DGA Techniques Aéronautiques
Olivier Graff	Intertechnique - Zodiac
Russell DeLoy Graham	Garmin International
Robert Green	BAE Systems
Mark Grindle	Systems Enginuity
Peter Grossinger	Pilatus Aircraft
Mark Gulick	Solers, Inc.
Pierre Guyot	Dassault Aviation
Ibrahim Habli	University of York
Ross Hannan	Sigma Associates (Aerospace) Limited
Christopher H. Hansen	Rockwell Collins
Wue Hao Wen	Civil Aviation Administration of China (CAAC)
Keith Harrison	HVR Consulting Services Ltd
Bjorn Hasselqvist	Saab AB
Kevin Hathaway	Aero Engine Controls (Goodrich)
David Hawken	NATS
Kelly Hayhurst	NASA
Peter Heath	Securaplane Technologies
Myron Hecht	Aerospace Corporation
Don Heck	Boeing Company
Peter Heller	Airbus Operations GmbH
Barry Hendrix	Lockheed Martin
Michael Hennell	LDRA
Michael Herring	Rockwell Collins
Ruth Hirt	FAA

Name	**Organization**
Kent Hollinger	Mitre Corp
C. Michael Holloway	NASA
Ian Hopkins	Aero Engine Controls (Rolls-Royce)
Gary Horan	FAA
Chris Hote	PolySpace Inc.
Susan Houston	FAA
James Hummell	Embedded Plus
Dr. James J. Hunt	aicas
Rebecca L. Hunt	Boeing Company
Stuart Hutchesson	Aero Engine Controls (Rolls-Royce)
Rex Hyde	Moog Inc. Aircraft Group
Mario Iacobelli	Mannarino Systems
Melissa Isaacs	FAA
Vladimir Istomin	Sukhoi Civil Aircraft Company (SCAC)
Stephen A. Jacklin	NASA
Matt Jaffe	Embry-Riddle Aeronautical University
Marek Jaglarz	Pilatus Aircraft
Myles Jalalian	FAA
Merlin James	Garmin International
Tomas Jansson	Saab AB
Eric Jenn	Thales
Lars Johannknecht	EADS
Rikard Johansson	Saab AB
John Jorgensen	Universal Avionics Systems
Jeffrey Joyce	Critical Systems Labs
Chris Karis	Ensco
Gene Kelly	CertTech
Anne-Cécile Kerbrat	Aeroconseil
Randy Key	FAA
Charles W. Kilgore II	FAA
Wayne King	Honeywell
Daniel Kinney	Boeing Company
Judith Klein	Lockheed Martin
Joachim Klichert	Diehl Avionik Systeme
Jeff Knickerbocker	Sunrise Certification & Consulting, Inc.
John Knight	University of Virginia
Rainer Kollner	Verocel
Andrew Kornecki	Embry-Riddle Aeronautical University
Igor Koverninskiy	Gos NIIAS
Jim Krodel	Pratt & Whitney
Paramesh Kunda	Pratt & Whitney Canada
Sylvie Lacabanne	AIRBUS
Gérard Ladier	Airbus/Aerospace Valley
Ron Lambalot	Boeing Company
Boris Langer	Diehl Aerospace
Susanne Lanzerstorfer	APAC GesmbH
Gilles Laplane	SAGEM DS Safran Group
Jeanne Larsen	Hamilton Sundstrand
Emmanuel Ledinot	Dassault Aviation
Stephane Leriche	Thales
Hong Leung	Bell Helicopter Textron
John Lewis	FAA

Name	**Organization**
John Li	Thales
Mark Lillis	Goodrich GPECS
Barbara Lingberg	FAA
Pierre Lionne	EADS APSYS
Hoyt Lougee	Foliage Software Systems
Howard Lowe	GE Aviation
Hauke Luethje	NewTec GmbH
Jonathan Lynch	Honeywell
Françoise Magliozzi	Atos Origin
Veronique Magnier	EASA
Kristine Maine	Aerospace Corporation
Didier Malescot	DSNA/DTI
Varun Malik	Hamilton Sundstrand
Patrick Mana	EUROCONTROL
Joseph Mangan	Coanda Aerospace Software
Ghilaine Martinez	DGA Techniques Aéronautiques
Steven C. Martz	Garmin International
Peter Matthews	Independent Consultant
Frank McCormick	Certification Services Inc
Scott McCoy	Harris Corporation
Thomas McHugh	FAA
William McMinn	Lockheed Martin
Josh McNeil	US Army AMCOM SED
Kevin Meier	Cessna Aircraft Company
Amanda Melles	Bombardier
Marc Meltzer	Belcan Engineering
Steven Miller	Rockwell Collins
Gregory Millican	Honeywell
John Minihan	Resource Group
Martin Momberg	Cassidian Air Systems
Pippa Moore	UK CAA
Emilio Mora-Castro	EASA
Endrich Moritz	Technical University
Robert Morris	CDL Systems Ltd.
Allan Terry Morris	NASA
Steve Morton	TBV Associates
Nadir Mostefat	Mannarino Systems
Fred B. Moyer	Rockwell Collins
Robert D. Mumme	Embedded Plus Engineering
Arun Murthi	AERO&SPACE USA
Armen Nahapetian	Teledyne Controls
Gerry Ngu	EASA
Elisabeth Nguyen	Aerospace Corporation
Robert Noel	Mitre Corp
Sven Nordhoff	SQS AG
Paula Obeid	Embedded Plus Engineering
Eric Oberle	Becker Avionics
Brenda Ocker	FAA
Torsten Ostermeier	Bundeswehr
Frederic Painchaud	Defence Research and Development Canada
Sean Parkinson	Resource Group

Name	Organization
Dennis Patrick Penza	AVISTA
Jean-Phillipe Perrot	Turbomeca
Robin Perry	GE Aviation
David Petesch	Hamilton Sundstrand
John Philbin	Northrop Grumman Integrated Systems
Christophe Piala	Thales Avionics
Cyril Picard	EADS APSYS
Francine Pierre	Thales Avionics
Patrick Pierre	Thales Avionics
Gerald Pilj	FAA
Benoit Pinta	Intertechnique - Zodiac
Mo Piper	Boeing Company
Andreas Pistek	ITK Engineering AG
Laurent Plateaux	DGA
Laurent Pomies	Independent Consultant
Jennifer Popovich	Jeppesen Inc.
Clifford Porter	Aircell LLC
Frédéric Pothon	ACG Solutions
Bill Potter	The MathWorks Inc
Sunil Prasad	HCL Technologies, Chennai, India
Paul J. Prisaznuk	ARINC-AEEC
Naim Rahmani	Transport Canada
Angela Rapaccini	ENAC
Lucas Redding	Silver-Atena
David Redman	Aerospace Vehicle Systems Institute (AVSI)
Tammy Reeve	Patmos Engineering Services, Inc.
Guy Renault	SAGEM DS Safran Group
Leanna Rierson	Digital Safety Consulting
George Romanski	Verocel
Cyrille Rosay	EASA
Edward Rosenbloom	Kollsman, Inc
Tom Roth	Airborne Software Certification Consulting
Jamel Rouahi	CEAT
Marielle Roux	Rockwell Collins France
Rudy Ruana	RTCA, Inc.
Benedito Massayuki Sakugawa	ANAC Brazil
Almudena Sanchez	GMV SA
Vdot Santhanam	Boeing Company
Laurence Scales	Thales
Deidre Schilling	Hamilton Sundstrand
Ernst Schmidt	Bundeswehr
Peter Schmitt	Universität Karlsruhe
Dr. Achim Schoenhoff	EADS Military Aircraft
Martin Schwarz	TT Technologies
Gabriel Scolan	SAGEM DS Safran Group
Christel Seguin	ONERA
Beatrice Sereno	Teuchos SAFRAN
Phillip L. Shaffer	GE Aviation
Jagdish Shah	Parker
Vadim Shapiro	TetraTech/AMT
Jean François Sicard	DGA Techniques Aéronautiques
Marten Sjoestedt	Saab AB

Name	Organization
Peter Skaves	FAA
Greg Slater	Rockwell Collins
Claudine Sokoloff	Atos Origin
Marge Sonnek	Honeywell
Guillaume Soudain	EASA
Roger Souter	FAA
Robin L. Sova	FAA
Richard Spencer	FAA
Thomas Sperling	The Mathworks
William StClair	LDRA
Roland Stalford	Galileo Industries Spa
Jerry Stamatopoulous	Aircell LLC
Tom Starnes	Cessna Aircraft Company
Jim Stewart	NATS
Tim Stockton	Certon
Victor Strachan	Northrop-Grumman
John Strasburger	FAA
Margarita Strelnikova	Sukhoi Civil Aircraft Company (SCAC)
Ronald Stroup	FAA
Will Struck	FAA
Wladimir Terzic	SAGEM DS Safran Group
Wolfgang Theurer	C-S SI
Joel Thornton	Tier5 Inc
Mikael Thorvaldsson	KnowIT Technowledge
Bozena Brygida Thrower	Hamilton Sundstrand
Christophe Travers	Dassault Aviation
Fay Trowbridge	Honeywell
Nick Tudor	Tudor Associates
Silpa Uppalapati	FAA
Marie-Line Valentin	Airbus
Jozef Van Baal	Civil Aviation Authorities Netherlands
John Van Leeuwen	Sikorsky Aircraft
Aulis Viik	NAV Canada
Bertrand Voisin	Dassault Aviation
Katherine Volk	L-3 Communications
Dennis Wallace	FAA
Andy Wallington	Bell Helicopter
Yunming Wang	Esterel Technologies
Don Ward	AVSI
Steve Ward	Rockwell Collins
Patricia Warner	Software Engineering
Michael Warren	Rockwell Collins
Rob Weaver	NATS
Yu Wei	CAA China
Terri Weinstein	Parker Hannifin
Marcus Weiskirchner	EADS Military Aircraft
Daniel Weisz	Sandel Avionics, Inc.
Rich Wendlandt	Quantum3D
Michael Whalen	Rockwell Collins
Paul Whiston	High Integrity Solutions Ltd
Todd R. White	L-3 Communications/Qualtech

Name	**Organization**
Virginie Wiels	ONERA
ElRoy Wiens	Cessna Aircraft Company
Terrance Williamson	Jeppesen Inc.
Graham Wisdom	BAE Systems
Patricia Wojnarowski	Boeing Commercial Airplanes
Joerg Wolfrum	Diehl Aerospace
Kurt Woodham	NASA
Cai Yong	CAAC (Civil Aviation Administration of China)
Edward Yoon	Curtiss-Wright Controls, Inc
Robert Young	Rolls-Royce
William Yu	CAAC China
Erhan Yuceer	Savunma Teknolojileri Muhendislik ve Ticaret
Uli Zanker	Liebherr

APPENDIX OO.B –FREQUENTLY ASKED QUESTIONS

This section contains frequently asked questions (FAQs) about using OOT&RT with DO-178C, as well as DO-278A. The purpose of a FAQ is to provide short and concise responses to questions that are frequently asked by industry concerning the material in this supplement. A FAQ contains no new or additional guidance material.

The FAQs are organized into five categories:

- General questions (see OO.B.1).
- Requirements considerations (see OO.B.2).
- Design considerations (see OO.B.3).
- Programming language considerations (see OO.B.4).
- Verification considerations (see OO.B.5).

The responses to these FAQs have been prepared and approved by RTCA Special Committee #205 (SC-205) and EUROCAE Working Group #71 (WG-71). These FAQs have no recognition by the certification authorities and are provided for information purposes only.

OO.B.1 General Questions

FAQ #1: What is covered by this supplement?

Reference: Annex OO.D sections OO.D.1.2, OO.D.1.3, OO.D.1.4, OO.D.1.5, OO.D.1.6, OO.D.1.7, OO.D.2.1, OO.D.2.2, OO.D.2.3, and OO.D.2.4

Keywords: component-based development; dynamic memory management; exception management; overloading; parametric polymorphism; resource analysis; structural coverage; traceability; type conversion; virtualization;

Answer:

The OOT&RT supplement covers object-oriented technology, as well as the following related techniques:

- Parametric polymorphism (see Annex OO.D.1.2)
- Overloading (see Annex OO.D.1.3)
- Type conversion (see Annex OO.D.1.4)
- Exception management (see Annex OO.D.1.5)
- Dynamic memory management (see Annex OO.D.1.6)
- Virtualization (see Annex OO.D.1.7)

These techniques were included in the supplement because they are often used in combination with object-oriented technology. The following topics are also discussed in the context of OOT&RT:

- Traceability (see Annex OO.D.2.1)
- Structural Coverage (see Annex OO.D.2.2)
- Component-based development (see Annex OO.D.2.3)
- Resource Analysis (see Annex OO.D.2.4)

<u>FAQ #2</u>: What is the relationship between this supplement and the OOTiA Handbook?

Reference: OOTiA Handbook

Keywords: OOTiA

Answer:

The Object-Oriented Technology in Aviation (OOTiA) Handbook was an input to the process of writing this supplement. The handbook raises many potential issues for OOT&RT development. This supplement provides guidance for these and other issues; therefore, the OOTiA Handbook is superseded by this supplement.

<u>FAQ #3</u>: What additional information is required for the software planning process in an OO related system?

Reference: Section OO.4.3

Keywords: planning; Plan for Software Aspects of Certification; PSAC

Answer:

The Plan for Software Aspects of Certification (PSAC) may identify that the product will be utilizing this supplement when creating the life cycle data for the project. The PSAC may describe the plan to use the reusable components and a plan for deactivation of unused components may be clearly defined. Use of virtualization and its associated executable program(s) may be defined.

<u>FAQ #4</u>: What additional measures may be taken when reusing OO components?

Reference: Sections OO.4.2.n, OO.5.2.2.l, and Annex OO.D.2.3

Keywords: reusable software

Answer:

Reusable software may be comprised of procedures, methods, classes, packages, and frameworks. Reusable software often contains more functionality than required by the current system being certified. If this extra functionality results in extra code in the system itself, then there may be unused code with which to deal. When reusing components, identify the resulting derived requirements and any functionality to be deactivated (see OO.4.2.n, OO.5.2.2.l, and Annex OO.D.2.3).

FAQ #5: What additional objectives and activities have been added to support OOT&RT in the software development processes?

Reference: Section OO.5.2.2, Table OO.A-2, and Annex OO.D

Keywords: control category

Answer:

There are no additional objectives or activities for the software requirements process.

There are no additional objectives for the software design process, but additional activities have been added to satisfy the OOT&RT software design process (see OO.5.2.2).

Table OO.A-2 identifies the control category, based on software level, that may be met for these life cycle data.

There are no additional objectives or activities for the coding process.

There are no additional objectives or activities for the integration process.

It should be noted that, even though no additional objectives were added for OOT&RT, additional recommendations are given in Annex OO.D.

FAQ #6: What additional objectives may be met and activities may be performed when reviewing and analyzing software using OO components?

Reference: Sections OO.6.3.3.a, OO.6.3.3.b, OO.6.3.4.b, OO.6.3.4.f, OO.6.7.1, OO.6.7.2, Table OO.A-4, Table OO.A-5, and Table OO.A-6

Keywords: type consistency

Answer:

There are no additional objectives when reviewing high-level and low-level requirements.

The following modified objectives apply to the review and analysis of the software architecture in OO components:

- Reference section OO.6.3.3.a.
- Reference section OO.6.3.3.b.

Table OO.A-4 provides leveling information (applicability, independence, and control category based on software level) for these objectives.

The following modified objectives apply to the review and analysis of the software code in OO components:

- Reference section OO.6.3.4.b.

- Reference section OO.6.3.4.f.

Additional verification objectives that support type consistency in OO components:

- Reference section OO.6.7.1 for new objective.

For local type consistency, the following additional activities may be performed to show compliance with the new objective:

- Reference section OO.6.7.2 for new activity.

Table OO.A-5 and Table OO.A-6 provide leveling information (applicability, independence, and control category based on software level) for these objectives and activities.

FAQ #7: What are the vulnerabilities required to be addressed in section OO.11.1?

Reference: Annex OO.D.1

Keywords: PSAC; vulnerabilities

Answer:

The vulnerabilities that should be addressed are those listed in Annex OO.D.1. These refer to the technology and techniques used in the software to be approved: inheritance, parametric polymorphism, overloading, type conversion, exception management, dynamic memory management, and virtualization. Annex OO.D.1 lists, for each key feature of this supplement, the vulnerabilities to be addressed and recommendations related to those vulnerabilities. For each of the features used, the PSAC should indicate that the given feature is used, and provide details on how the associated vulnerabilities will be handled.

FAQ #8: Are there any special concerns for using closures?

Reference: Section OO.1.6.1.5

Keywords: closure

Answer:

A closure is similar to an object in that it binds state to a function. Since a closure binds state, creating a closure creates a new data object with indefinite extent, which typically requires dynamic memory management. The state may change after each invocation, which may cause the behavior of the closure to change from one invocation to the next. When using closures, one should be aware that memory is allocated and verification takes the internal state into account, but this is the same as for objects with state and methods.

OO.B.2 Requirements Considerations

<u>FAQ #9</u>: What is the relationship between low-level requirements and classes?

Reference: Sections OO.5.2.2.i, OO.5.5.d, and OO.6.4.2.1.e

Keywords: exceptions; invariants; preconditions; postconditions

Answer:

Section OO.5.2.2.i states "Develop a locally type consistent class hierarchy with associated low-level requirements whenever substitution is relied upon." Section OO.6.4.2.1.e states, "For object-oriented software, the normal range test cases may ensure that class constructors properly initialize the state of their objects and the initial state is consistent with the requirements for the class." Section OO.5.5.d states "In an object-oriented design, all functionality is implemented in methods; therefore, traceability is from requirements to the methods and attributes that implement the requirements. Classes are an artifact of the architecture for organizing requirements. Due to subclassing, a requirement, which traces to a method implemented in a class, should also trace to the method in its subclasses when the method is overridden in a subclass." Low-level requirements may include the following:

- All invariants of each class.
- The normal and exceptional behavior of methods of each class, for example, with preconditions (the states in which the method may be called) and postconditions (the result of the method when called).
- The initial state of the class as established by the constructors, including any initialization order constraints of the objects referenced by the constructor.

<u>FAQ #10</u>: What role do classes play in the design process?

Reference: Sections OO.5.2.2.h, and OO.6.3.4.b

Keywords: architecture; class; design; low-level requirements

Answer:

Section OO.5.2.2.h states: "Develop the class hierarchy based on high-level requirements." Section OO.6.3.4.b states: "The objective is to ensure that the Source Code matches the data flow, control flow, and class hierarchy defined in the software architecture." The class hierarchy, relationships between classes, and the methods contained in each class are part of the architecture. Low-level requirements specify the behavior of the methods of each class and any constraints associated with each class. Low-level requirements complement the architectural decisions made during design.

FAQ #11: When should a requirement be expressed as invariants as opposed to preconditions and postconditions?

Reference: Section OO.1.6.1.2.1

Keywords: invariants; preconditions; postconditions

Answer:

Preconditions indicate when a given subprogram can be called. Postconditions indicate the result of a method after its completion. An invariant is stronger than the same constraint formulated by a postcondition because it constrains all visible states of the execution of all subprograms, not just the initial and final states of a single subprogram. Without this concept, the constraint on objects would have to be restated as a postcondition on every subprogram. Invariants are also preconditions for each subprogram except constructors. This duplication in preconditions and postconditions would complicate maintenance and be more error prone.

FAQ #12: Do all methods, including constructor, accessor, mutator, and destructor, need requirements?

Reference: Section OO.1.6.1.2.1

Keywords: accessor; constructor; destructor; method; mutator; requirement

Answer:

Yes, but simple methods need only be specified by simple requirements. For example, a constructor is responsible for establishing the initial state of an object, therefore it is sufficient to have a requirement that states what that state is; a requirement for an accessor need only state that a particular field be accessible from outside the object; a requirement for a mutator need only state that a particular field be mutable from outside the class; and a requirement for a destructor need only state the resources held by a class that need to be released.

OO.B.3 Design Considerations

FAQ #13: What is typing?

Reference: Section OO.4.2.n

Keywords: type consistency; typing

Answer:

Typing is a categorization technique with the primary goal of ensuring language type consistency by ensuring consistent use of variables. It is intended to ensure proper behavior by distinguishing between programs that are properly typed and those that violate the type system. There is a range of values a variable can contain during the execution of a program. The type of that variable specifies the limits of its range of values.

FAQ #14: Why is the Liskov Substitution Principle (LSP) important?

Reference: Section OO.1.6.1.2.1

Keywords: dynamic dispatch; inheritance; Liskov Substitution Principle; LSP; overriding; pessimistic testing; substitutability

Answer:

The main concern with OOT is the safe use of inheritance, method override, and dynamic dispatch. When using these features, it is not obvious from local code review which method is actually executed at any call point in a program. Dynamic dispatch ensures that the correct code is executed for manipulating each data type.

The question then is, what constitutes the safe use of inheritance, method overriding, and dynamic dispatch? If this question could not be answered, then each possible dispatch at every call point in a program would need to be tested (pessimistic testing). Insisting on this might cause a combinatorial explosion of test cases that would have to be run. Fortunately, a disciplined use of subclassing, based on type theory, provides a safe environment for dispatching.

For a type to be a proper subtype of some other type, it needs to exhibit the same behavior as each of its supertypes. In other words, any subtype of a given type needs to be usable wherever the given type is required. If a subclass is defined that does not adhere to this principle of substitutability, then dynamic dispatching may become unsafe. The LSP formally defines what constitutes a proper subtype.

The LSP defines the limits on how a subclass may behave. Each method redefined in a subclass needs to meet the requirements of the same method in any of its superclasses:

- Preconditions may not be strengthened, because when a subclass's method has a stronger precondition than the method it overrides, the precondition may not be met in all the contexts where the original method was called.
- Postconditions may not be weakened, otherwise the context in which the method is called may rely on properties that are not provided by the new method.

In addition, invariants defined on the state of a class may not be weakened.

Adherence to LSP ensures that inheritance, method overriding, and dynamic dispatch are used safely. Demonstrating that subclasses of a given class are proper subtypes of the type of the class may be a means to demonstrate that the dynamic dispatching used in a program is safe. It can be shown that the set of classes of the objects that can be dispatched to are in fact substitutable for their statically declared classes.

A simple means of ensuring substitutability is to show that the entire class hierarchy is a proper type hierarchy. In other words, every class fulfills the requirements of all its superclasses.

FAQ #15: Must all classes in an application be type consistent?

Reference: Sections OO.4.2.n and OO.6.2.g

Keywords: class; substitutability; type consistency

Answer:

Type consistency can be considered globally or locally. Global type consistency is type consistency through the entire class hierarchy of the system. Local type consistency is a subset of global type consistency. Local type consistency is type consistency in the context where substitution can occur. A local context may be the context of a variable, an attribute, or a parameter. It is generally sufficient to consider local type consistency for type safety. It is this property that can be used to ensure that only applicable methods are called at any given call point.

This supplement only requires local type substitutability. This means that if it can be demonstrated that a given class cannot be substituted by any of its subclasses in the context of the software, this class hierarchy may not need to be considered for LSP. Of course, when new code is introduced, new declarations may need to be included in the LSP analysis. In general, the lower a class is in its class hierarchy, the greater the chance is that it will need to be LSP compatible with its parent class.

FAQ #16: What impact does the Liskov Substitution Principle (LSP) have on architecture and low-level requirements?

Reference: Section OO.5.2.2.i

Keywords: architecture; Liskov Substitution Principle; LSP; low-level requirements

Answer:

The LSP depends on having well defined preconditions and postconditions for each subprogram, and invariants for each class. Invariants describe constraints on the state of each object defined by a class, that is, its attributes. These preconditions, postconditions, and invariants are part of the low-level requirements of the subprograms defined in the classes in the system. The classes themselves are part of the architecture. Fulfilling LSP may require adding additional layers of classes to properly separate requirements. For example, a speed controller may define a common method for getting the current speed, but different implementations may have different valid speed ranges for the return value. A gauge that depends on a specific limit may not be able to use a base class, but one that can display the full range of the return data type could. If the base class included a specific range limitation, then subclasses that support a wider range would not be valid subclasses.

FAQ #17: Are there garbage collection techniques that should be avoided?

Reference: Section OO.1.6.2.6.4, Annex OO.D.1.6.1, and FAQ #26

Keywords: garbage collection

Answer:

Yes. In particular, there exists a type of garbage collector known as a "conservative" collector. Such a collector scans all memory which may contain references and considers any data value which appears to be a memory reference (that has a bit pattern which corresponds to a valid memory reference) to actually be a memory reference and therefore refer to live memory. Unfortunately, such a bit pattern might not actually be such a memory reference, but some piece of data (for example, a mantissa) that simply appears to match the pattern of a memory reference but in actuality is not such a reference.

FAQ #18: Should inheritance be used for code sharing?

Reference: Section OO.1.6.2.1

Keywords: code sharing; inheritance; parametric polymorphism

Answer:

Inheritance is not intended primarily for code sharing, but for sharing common behavior with subtypes. There are instances when inheritance may not be the best way to share code. The delegation pattern is an alternative technique that does not involve inheritance. Here, the work of a method is implemented by calling a method on an object held in a private field. Some languages offer an additional mechanism for sharing code: parametric polymorphism (called generics in Java and Ada, and templates in C++). Parametric polymorphism provides a mechanism for sharing code that parameterizes the type to which the code can be applied. Different subclasses with different parameters are not a problem because they are not the same type or a subtype of the base class.

FAQ #19: What techniques can be used to minimize potential problems with the use of multiple inheritance?

Reference: Annex OO.D.1.1.1

Keywords: implementation inheritance; interface inheritance; multiple inheritance

Answer:

As mentioned in Annex OO.D.1.1.1, there are two kinds of multiple inheritance: interface inheritance and implementation inheritance. Of these, implementation inheritance is the more problematic.

Problems with multiple inheritance are less likely to be encountered if a program never uses implicit or explicit type casting. This, however, is probably not a realistic solution as it is not unreasonable to encounter a situation where the receiver of an assignment (the l-value) expects a reference to only one of the (multiply inherited) types of the reference being assigned (the r-value). Since the l-value type cannot be changed to match that of the r-value, the reverse type cast needs to be made and this can create vulnerabilities.

Once full multiple inheritance is allowed, types are permitted to contain arbitrary sets of other types. Consider the situation where there exists types A, B, and C. Type A is a simple type with no inheritance. Type B is a subtype of A, and type C is also a subtype of A. In this situation, casting from type C or type B to type A introduces no particular confusion or cause for concern. If, however, a new type D is introduced which is a subtype of both B and C, type casting of a D reference to an A reference becomes problematic. Each of the A's in the inherited B and C can have different state, so simply casting a D reference as an A is ambiguous, as it does not indicate which of the A's is the intended referent.

A typical technique to resolve this in practical programming languages is to tell the compiler the intent of the programmer via some sort of annotation (static_cast or dynamic_cast, for example, in C++). In the last example, the situation could be disambiguated such that the A portion of the D referent was intended to come from its B portion (yes, this is complicated — hence the increased vulnerability) by first casting the D referent to a B, then casting that to an A. Unfortunately, this involves the programmer telling the compiler how to interpret the type casting of a reference. This can (certainly in the case of C++) simply be incorrect as the compiler may have no way of verifying that such a cast is actually type consistent, thus having to fall back on nothing more substantial than the programmer's good intentions.

Recommended techniques to minimize the impact of multiple inheritance therefore include using extreme diligence in verification that manual type casting is done in a type safe manner. The need for this can be minimized by excluding the possibility of allowing a type to be inherited by multiple other types that are in turn inherited by other types. Often this is implemented by the very severe restriction of never allowing any form of multiple inheritance other than interface inheritance (the inheritance of behavior without the inheritance of state). However, if an implementation inheritance hierarchy can be made clear enough that any required manual verification of type consistency can be performed dependably, and it can be verified that a type can never contain multiple instances of some other inherited type (such as strict usage of dynamic_cast and virtual base classes in C++), multiple inheritance can be used with minimal additional vulnerabilities.

FAQ #20: Is static dispatch safer than dynamic dispatch?

Reference: Annex OO.D.1.1.1 and FAQ #24

Keywords: static dispatch; dynamic dispatch

Answer:

Static dispatch is not safe in the presence of method overriding, as explained in Annex OO.D.1.1.1. An example of static dispatch and its potential problems in various programming languages is provided in FAQ #24.

<u>FAQ #21</u>: How can Liskov Substitution Principle (LSP) be violated in a strongly-typed language, such as Ada or Java?

Reference: Section OO.1.6.1.2.1

Keywords: Liskov Substitution Principle; LSP; strongly-typed language

Answer:

As an example of a situation that violates the LSP, consider a class that provides a speed controller. Initially, the speed controller class is designed to have a constructor which sets the initial speed and subsequently causes speed changes to occur through the execution of its adjust speed method, which takes a speed change as its argument. Later, this class is used as the base for a derived speed controller class which manages a different kind of speed controller, an auto speed controller that, rather than take a speed change delta as its parent class does, changes the speed by the execution of a set speed method, which takes the desired resultant speed as an argument.

The auto speed controller class disables the inherited adjust speed method (since it is no longer deemed applicable), as the set speed method is intended to be used in its place. This is done without considering the postcondition of the inherited set speed method — the assumption that when invoked it causes the speed to change by the desired increment. Since this postcondition is violated, the derived class (auto speed controller) does not satisfy the LSP.

The following provides example code in C++, Ada, and Java of this situation. Part of this example is the use of the adjust speed method in a substituted auto speed controller with its implied postcondition of a change in speed from an existing zero value; that is, after the method is invoked the speed will be non-zero. This value is then used as the denominator in a division causing a divide by zero fault to occur.

```
////////////////////////////////////////////////////////////////
//  Listing 1. Violation of LSP - Java
////////////////////////////////////////////////////////////////
// Base class which implements a speed controller
class SpeedController {

   public int getSpeed() {
      return speed;
   }

   public void adjustSpeed( int increment ) {
      speed += increment;
      // code that tells controller hardware the speed increment
      // postcondition: speed augmented by 'increment'
   }

   // Return the time to traverse the given distance at the current
speed
   public int timeToGo( int distance ) {
      // here we expect that getSpeed() returns non-zero value
      return distance / getSpeed();
   }
   protected int speed = 0;
}
////////////////////////////////////////////////////////////////
// Subclass of Controller which violates LSP.
// It uses setDesiredSpeed() rather than adjustSpeed() to change the speed.
// The now LSP-broken adjustSpeed() does nothing, ignoring its postcondition.
class AutomaticSpeedController extends SpeedController {

   public void setDesiredSpeed( int val ) {
      desiredSpeed = val;
      // Code that tells hardware what the desired speed is
   }

   @Override
   public void adjustSpeed( int increment ) {
      // do nothing
      // postcondition: speed doesn't change
   }
   private int desiredSpeed = 0;
}
/////////////////////////////////////////////////////////////////
public class Main {
   public static void main( String[] args ) {
      final SpeedController controller =
         // new SpeedController(); // This substitution would have been fine.
         new AutomaticSpeedController(); // This substitution violates LSP
      // Taking the traditional view, the speed is incremented by 2 units
      controller.adjustSpeed( 2 );
      // Expected postcondition: speed is non-zero - since we just changed it
      System.out.println( "Time to go: " + controller.timeToGo( 5 ) );
      // Exception thrown only for instance of AutomaticSpeedController
   }
}

/////////////////////////////////////////////////////////////////
//  Listing 2. Violation of LSP - C++10
/////////////////////////////////////////////////////////////////
#include <stdio.h>
/////////////////////////////////////////////////////////////////
class Controller {
public:
   int Speed() {
      return speed;
      }
   virtual void adjustSpeed( int increment ) {
      if ( speed + increment > 0)
           speed += increment;
      }
   // postcondition: speed augmented by 'increment'
   Controller() {
      speed = 0;
      }
private:
   int speed;
};
/////////////////////////////////////////////////////////////////
// routine relying on adjustSpeed post condition
int computeTimeToGo( Controller* controller , int distance ) {
   controller->adjustSpeed( 3 );
   // here we expect that controller.Speed() is non-zero
   return distance / controller->Speed();
}
```

```
/////////////////////////////////////////////////////////////////
// derived class violating LSP on adjustSpeed
class AutoController : public Controller {
public:
   int DesiredSpeed();

   void setDesiredSpeed( int val );

   virtual void adjustSpeed( int increment ) {} // does nothing
   // postcondition : speed doesn't change

   AutoController() {
      desiredSpeed = 1;
   }
private:
   int desiredSpeed;
};
/////////////////////////////////////////////////////////////////
int main (int argc, char** argv) {
   Controller* controller = new AutoController();
   int time = computeTimeToGo(controller, 5);
   printf( "Time: %d\n", time );
   return 0;
}
```

```
--------------------------------------------------------
--  Listing 3. Violation of LSP – Ada (1 of 2)
--------------------------------------------------------
-- Basic speed controller definition
package Speed1 is
   subtype Speed_Type  is Integer range 0 .. 200;
   subtype Speed_Delta is Integer range -5 .. +5;

   type Controller is tagged private;
   function Speed (This : Controller) return Speed_Type;
   procedure Adjust_Speed (This : in out Controller; Increment : 
Speed_Delta);
   pragma Postcondition (This.Speed = This'Old.Speed + Increment);
private
   type Controller is tagged record
      Actual_Speed : Speed_Type := 0;
   end record;
end Speed1;
--------------------------------------------------------
package body Speed1 is
   function Speed (This : Controller ) return Speed_Type is
   begin
      return This.Actual_Speed;
   end Speed;

   procedure Adjust_Speed
      (This : in out Controller; Increment : Speed_Delta) is
   begin
      This.Actual_Speed := This.Actual_Speed + Increment;
   end Adjust_Speed;
end Speed1;
--------------------------------------------------------
-- routine relying on adjustSpeed post condition
with Speed1; use Speed1;
procedure Compute_Time_To_Go (C : in out Controller'Class;
   Distance : Integer; Time_To_Go : out Integer) is
begin
   C.Adjust_Speed (3);
   -- Controller's Adjust_Speed guarantees that C.Speed /= 0
   -- but we have a divide by 0 here if C is an
Automatic_Speed_Controller
   Time_To_Go := Distance / C.Speed;
end Compute_Time_To_Go;


---------------------------------------------------------
--  Listing 4. Violation of LSP – Ada (2 of 2)
---------------------------------------------------------
-- Derived class violating LSP on adjustSpeed
with Speed1; use Speed1;
package Speed2 is
   type Auto_Controller is new Controller with private;

   function Desired_Speed (This : Auto_Controller) return Speed_Type ;

   procedure Set_Desired_Speed
      (This : in out Auto_Controller; Val : Speed_Type);
   pragma Postcondition (This.Desired_Speed = Val);

   overriding procedure Adjust_Speed
      (This : in out Auto_Controller; Increment : Speed_Delta);
   pragma Postcondition (This'Old.Speed = This.Speed);

private
   type Auto_Controller is new Controller with record
      Desired_Speed : Speed_Type := 0;
   end record;
end Speed2;
---------------------------------------------------------
package body Speed2 is
   function Desired_Speed ( This : Auto_Controller) return Speed_Type
is
   begin
      return This.Desired_Speed;
   end Desired_Speed;

   procedure Set_Desired_Speed
      ( This : in out Auto_Controller; Val : Speed_Type) is
   begin
      This.Desired_Speed := Val;
   end Set_Desired_Speed;

   procedure Adjust_Speed
      (This : in out Auto_Controller; Increment : Speed_Delta ) is
   begin
      null;
   end Adjust_Speed;
end Speed2;
-----------------------------------------------------
-- program raises an exception due to violation of LSP
with Speed2; use Speed2;
with Compute_Time_To_Go;
procedure Main is
   Res : Integer;
   Ctrl : Auto_Controller;
begin
   Compute_Time_To_Go (Ctrl, 5, Res);
end Main;
```

FAQ #22: What would need to be done to make the examples in FAQ #21 conform to the Liskov Substitution Principle (LSP)?

Reference: FAQ #21

Keywords: Liskov Substitution Principle; LSP

Answer:

The problem comes from inheriting the adjust speed method and its associated postcondition. One solution would be to refactor the speed controller base class into an abstract class with no adjust speed method, and then have derived classes for manual speed controller (with an adjust speed method) and an auto speed controller (with a set speed method). If this were done, an auto speed controller would not be substitutable for a manual speed controller — and this would be enforced by the type consistency requirements of the three languages considered.

FAQ #23: Does the discussion on preconditions, postconditions, and invariants imply the use of a particular design methodology?

Reference: Section OO.1.6.1.2.1

Keywords: design methodology; invariants; preconditions; postconditions

Answer:

No, these terms predate the design methodologies that use it. Though preconditions, postconditions, and invariants are core principles of particular design methodologies, those methodologies generally have other features as well, such as language support for preconditions, postconditions, and invariants, a method for moving from requirements to design, tool support, and other development process features. While programming languages that support these design methodologies might aid in the identification of defects when designing, implementing, and using classes, they may not be complete enough to demonstrate type safety. Showing that the LSP is maintained does not require these language additions.

OO.B.4 Programming Language Considerations

FAQ #24: What is an example of static dispatch with method overriding?

Reference: Annex OO.D.1.1.1

Keywords: method overriding; static dispatch

Answer:

The following are examples in C++ and Ada.

In C++, static dispatch can occur in two ways:

- The method of the superclass is explicitly called at the dispatch point.
- The method of the superclass is non-virtual.

```
class Superclass
{
public:
   // designed to be overridden
   virtual void    Method1() {}
   // NOT designed to be overridden
   void      Method2() {}
};

class Subclass : public Superclass
{
public:
   // method overriding with dynamic binding
   virtual void    Method1() {}
   // static binding with method overriding: possible defects!
   void      Method2() {}
};

void UseSuperType(Superclass * ptrToSuperclass)
{
// No issue here. The subprogram that is called depends on the
// underlying type.
   ptrToSuperclass->Method1();

// Intentional static dispatch. If a Subclass is passed in, then
// Subclass::Method1() will NOT be called. Instead,
// Superclass::Method1 will always be called. The designer
// explicitly stated this for this dispatch point.
   ptrToSuperclass->Superclass::Method1();

// Likely defect! If a Subclass is passed in, then
// Subclass::Method2() will NOT be called. Instead,
// Superclass::Method2 will always be called. The designer
// probably did not intend this.
   ptrToSuperclass->Method2();
};
```

In Ada (as of Ada 2005), dynamic dispatch happens when a method (primitive operation in Ada terminology) is called with a class wide actual. It means that in the body of a method, calls to other methods of the same class are not dispatching unless a type conversion is used. Since Ada is a strongly-typed language, it usually does not cause problems but there are two constructs that can lead to the static dispatch vulnerability described in Annex OO.D.1.1.1: view conversions and method inheritance. Here is an illustration:

```
type C1 is tagged private;  -- Class
procedure Method1 (X : C1);
procedure Method2 (X : C1);
procedure Method1 (X : C1) is
   Y : C1'Class renames C1'Class (X);
begin
   X.Method2;  -- Static dispatch
   Y.Method2;   -- Dynamic dispatch
end Method1;
…
type C2 is new C1 with private;  --subclass
--  Method1 is inherited
overriding procedure Method2 (X : C2);
…
Obj : C2'Class := …;
C1(Obj).Method2; -- static dispatch through view conversion
Obj.Method1; -- dynamic dispatch to an inherited primitive
             --  Leading  to  potentially  improper  static  dispatch  to
Method2
             -- in the body of Method1
```

In Java, only final methods may be statically dispatched and they cannot be overridden.

<u>FAQ #25</u>: Is it acceptable to use a COTS library, such as the C++ Standard Template Library, without explicitly specifying high-level and low-level software requirements, tests, and documents for this library?

Reference: Annex OO.D.2.3

Keywords: COTS; library; reuse

Answer:

No. Such a library is a reusable software component as discussed in Annex OO.D.2.3 Component-Based Development.

<u>FAQ #26</u>: What considerations are important when using garbage collection?

Reference: Sections OO.1.6.2.6, OO.6.8, OO.6.8.2, Annex OO.D.1.6, and FAQ #17

Keywords: dynamic memory management; garbage collection

Answer:

There are several points to be considered when using garbage collection, related to both the collector itself and to the application. They are the same as for any other dynamic memory management technique, as described in section OO.1.6.2.6 and elaborated on in Annex OO.D.1.6. Of the seven activities listed in section OO.6.8.2 for ensuring that the objective of section OO.6.8 for dynamic memory management is met, all but one can be fulfilled by the implementation of a garbage collector.

The most important aspect of a garbage collector is that it is exact: the collector needs to be able to determine which data in memory are references to other objects and which are not. This is necessary to ensure that all reachable objects can be found and no object that is not reachable is considered reachable because its address matches some non-pointer data (see FAQ #17 in OO.B.3). Activities in sections OO.6.8.2.a and OO.6.8.2.e are relevant here.

Another aspect to address is the timeliness of the collector: that it enables timely response and that it recycles memory quickly enough for subsequent allocations. The first point means that the cycle of determining what objects are currently reachable and returning unreachable objects to the free list needs to be broken into sufficiently small uninterruptable steps. The second point means that the collector needs to complete a collection cycle fast enough to keep enough memory on the free list to satisfy subsequent allocations. If the collection activity is bound to allocation activity without using an additional independent task or thread of control, then collection activity is simply part of the execution time of the code that allocates memory; otherwise, the collector task may need to be scheduled and the application may have to be analyzed to determine the rate of object allocation. Activities in sections OO.6.8.2.c and OO.6.8.2.g are relevant here.

Another important aspect is fragmentation avoidance. Either objects are allocated in fixed size chunks so that fragmentation becomes unimportant, or free memory needs to be consolidated. Whereas the first point may increase the memory overhead a bit, the second requires moving objects in memory, where it is important to ensure that this does not cause an unbound blocking operation in the garbage collector. Activities in sections OO.6.8.2.b and OO.6.8.2.f, and possibly OO.6.8.2.g, are relevant here.

The last activity, section OO.6.8.2.d, is specific to each application. This requires analyzing or testing the application to ensure that there is always sufficient memory for the application. For testing, a means of tracing the available free memory is needed to ensure that sufficient free memory is always available. A formal method, such as data flow analysis, may also be used to verify that free memory is not exhausted by an application.

FAQ #27: What considerations are important when using a virtual machine?

Reference: Sections OO.1.6.2.7, OO.4.2, OO.11.7, OO.11.8, and Annex OO.D.1.7

Keywords: interpreter; virtual machine; virtualization

Answer:

Virtual machine is not a precise term, but typically involves some type of hardware or program (the "machine") that interprets one program instruction sequence to another representation. A virtual machine may include some kind of interpreter and a set of run-time libraries. Examples include a Java virtual machine, an XML interpreter, and resource virtualization software. Interpreters are used as part of some virtualization techniques as described in section OO.1.6.2.7 and Annex OO.D.1.7. In order to demonstrate the correctness of an interpreter, a clear specification of its semantics is needed. These can be the low-level requirements for verifying the interpreter. Section OO.4.2.m is relevant.

OO.B.5 <u>Verification Considerations</u>

<u>FAQ #28</u>: What additional objectives and activities have been added to support OOT&RT in the verification process?

Reference: Sections OO.6.3.3.a, OO.6.3.3.b, OO.6.3.4.b, OO.6.3.4.f, OO.6.7.1, OO.6.8.1, and OO.6.4.2.1.e

Keywords: local type consistency; memory management; verification

Answer:

There are modified and additional objectives for the verification process, and also additional activities.

Modified objectives include:

- Software architecture is compatible with high-level requirements (section OO.6.3.3.a).
- Software architecture is consistent (section OO.6.3.3.b).
- Source Code complies with software architecture (section OO.6.3.4.b).
- Source Code is accurate and consistent (section OO.6.3.4.f).

Additional objectives include:

- Verify local type consistency (section OO.6.7.1).
- Verify the use of dynamic memory management (section OO.6.8.1).

Additional activities include:

- Executable Object Code complies with high-level requirements (section OO.6.4.2.1.e).
- Executable Object Code complies with low-level requirements (section OO.6.4.2.1.e).

<u>FAQ #29</u>: In section OO.5.5.d, what is meant by "a requirement which traces to a method"?

Reference: Section OO.5.5.d

Keywords: trace

Answer:

Section OO.5.5.d states, "Due to subclassing, a requirement, which traces to a method implemented in some class, should also trace to the method in its subclasses when the method is overridden in the subclass."

This means that a high-level requirement that is traced to a low-level requirement for a superclass method may also trace through that low-level requirement to the implementation of the corresponding subclass method that overrides the superclass method.

FAQ #30: How does the formal verification activity for ensuring local type consistency differ from a low-level requirements review?

Reference: Section OO.1.6.1.2.1, and Formal Methods Supplement to DO-178C and DO-278A

Keywords: Liskov Substitution Principle; LSP

Answer:

The formal verification activity for local type consistency is an analysis activity that shows that the LSP (see section OO.1.6.1.2.1) has been met. A review is a qualitative assessment of acceptability with the aid of a standard, whereas the formal verification activity (reference Formal Methods Supplement to DO-178C and DO-278A) is a reproducible analysis that the requirements of the subclass method are compatible with those of every superclass.

FAQ #31: Does the guidance related to dynamic dispatch also apply to the use of function pointers (for example, function pointer tables and callback functions)?

Reference: Section OO.6.7

Keywords: dynamic dispatch; function pointers; polymorphism

Answer:

In general, the guidance related to dynamic dispatch does not apply to the use of function pointers because it is more general than dynamic method dispatch, and introduces additional vulnerabilities. The use of function pointers is a general issue not specific to OOT.

When a type is defined that uses function pointers to implement dynamic dispatch, then the guidance of section OO.6.7 applies, otherwise the guidance does not apply. For example, it is possible to implement classes and objects using the C programming language by using complicated structures and function pointer tables. The guidance in section OO.6.7 would apply to this case because the resulting construct is equivalent to a polymorphic type.

FAQ #32: What is the difference between fulfilling type consistency locally and globally?

Reference: Section OO.6.2.g

Keywords: type consistency

Answer:

The most general means of fulfilling type consistency in an application is to verify global type consistency. In this case, no further analysis is necessary about the concrete dispatch points within that application. If for all classes in an application there is verification that each class fulfills all requirements of every one of its superclasses, then global type consistency is ensured.

However, it is not always possible to achieve or verify global type consistency. As an example, if an application uses an object-oriented library where it is not known if the class hierarchy within this library is type consistent, it may be too complex to verify the type consistency of this library. In such a case, it may be made evident that the application does not have any dispatch point where possibly non-consistent types of the library can be used. With this evidence, verifying the type consistency of all except the library's classes ensures local type consistency.

As a result, fulfilling local instead of global type consistency is only possible in conjunction with an analysis of the dispatch points within the application.

FAQ # 33: How can local type consistency be verified?

Reference: Section OO.6.7.1

Keywords: Liskov Substitution Principle; LSP; type consistency

Answer:

As stated in section OO.6.7.1, either testing or formal methods can be used to verify the LSP, but verification using formal methods can be more complete. No matter which technique is used, each class in question should fulfill the requirements of all its supertypes, that is, pass all the tests, or fulfill the same proof obligations. Code review is insufficient to verify local type consistency.

FAQ # 34: How can local type consistency with an abstract superclass be verified by testing without using pessimistic testing?

Reference: Section OO.6.7

Keywords: type consistency

Answer:

As stated in section OO.6.7, Local Type Consistency Verification, each class needs to pass all tests of all of its superclasses for which it can be substituted. This means, that tests should be written for all classes including abstract superclasses for which the tests cannot directly be executed. In this case, they should be executed with all concrete subclasses.

<u>FAQ #35</u>: How can classes that include unencapsulated data be verified for local type consistency?

Reference: Section OO.6.7

Keywords: data encapsulation; type consistency

Answer:

Unencapsulated data, that is, data that can be accessed from other classes directly, has vulnerabilities similar to those of static methods. Unless the underlying language synthesizes accessor methods, references to object fields are resolved statically. As long as a subclass does not reuse a field name, there are no vulnerabilities other than ensuring that increased data coupling, that is, methods in other classes are involved in maintaining any class invariants involving those fields.

When there are no class invariants, local type consistency for unencapsulated data is verified by default as long as the subclass does not somehow restrict access to the inherited data members. Without class invariants, unencapsulated data is verified by default because there are no preconditions and the postcondition is simply that the data member is changed.

Class invariants restrict the valid states of an object. When data is not encapsulated, class invariants are easily broken by client software that intentionally or unintentionally forces the object into invalid states or through invalid state transitions.

While a type design with unencapsulated fields is not necessarily defective, such software requires careful documentation of the class invariants and careful and detailed management of those invariants by the client software. Low-level requirements and Source Code reviews may ensure that class constraints and client responsibilities are both well documented.

When a subclass inherits unencapsulated data along with class invariants, local type consistency verification should ensure the subclass does not itself break any of the class invariants of the super class.

Unencapsulated data can be present even when class methods do not have public access. For example, C++ allows subclasses to access data members of the super class that are designated as "protected". When the subclass has unrestricted access to any data within the superclass, local type consistency may be verified as described above for that data with respect to class invariants.

Consider the following simple unencapsulated C++ class:

```
class GeographicPoint
{
  public:
    float latitude;
    float longitude;
    GeographicPoint();
};


class Geographic3dPoint : public GeographicPoint
{
  public:
    float elevation;
    Geographic3dPoint();
};
```

In the example above, the low-level requirements defining the GeographicPoint type would typically include class invariants for the latitude and longitude data members ensuring that these data members contain values that lie within valid real-world values. If Geographic3dPoint is substituted for GeographicPoint in the application somewhere, then the constructor and any other methods defined for Geographic3dPoint should ensure that the class invariants for latitude and longitude are not violated by the subclass Geographic3dPoint.

FAQ #36: Does this supplement require tracing of low-level requirements to preconditions, postconditions, and invariants?

Reference: Section OO.5.5.d

Keywords: invariants; preconditions; postconditions; trace

Answer:

No, preconditions, postconditions, and invariants are a means of stating low-level requirements.

FAQ #37: Is "flattened class testing" a way of verifying local type consistency?

Reference: OOTiA, Volume 3; and Annex OO.D.2.2.3

Keywords: flattened class

Answer:

"Flattened class" testing (defined in the OOTiA Handbook) does not ensure that the same tests that are run on a class are also run on all its subclasses. Therefore, this method may not be appropriate for fulfilling the objectives stated in this supplement. "Flattened class" testing should be combined with type consistency analysis, see Annex OO.D.2.2.3.

FAQ #38: When preconditions, postconditions, and invariants are checked at run-time, what verification must be performed on the additional Source Code?

Reference: None

Keywords: preconditions; postconditions; invariants

Answer:

It is important to distinguish between low-level requirements stated as preconditions, postconditions, and invariants and language features or Source Code that check these constraints. The former specify conditions that must be fulfilled, whereas language features, such as an Ada precondition aspect or an assert statement, provide a means of checking all or part of these conditions. Source Code injected by these languages features or manually developed is typically active during a portion of the verification process. This code should be addressed as any other robustness code.

FAQ #39: Is there a problem with covariant collections?

Reference: None

Keywords: contravariance; covariance

Answer:

Yes. The ability to have covariant collections can prevent the compiler from detecting some problems at compile time and, instead, having them appear at run-time. In the following Java example, note that Float and Double are both subtypes of Number:

```
// Float is covariant to Number
Float[] floats = new Float[] {new Float( 1.0 ), new Float( 2.0 )};
// Number is contravariant to Float, so the assignment is allowed
Number[] numbers = floats;
// Throws an ArrayStoreException at run-time
numbers[1] = new Double( 1.2 );
```

The problem is not as acute with class-based collection types using parametric polymorphism:

```
ArrayList<Float> floats = new ArrayList<Float>();
floats.add(new Float(1.0);
floats.add(new Float(2.0);
// The following generates a compiler error.
ArrayList<Number> numbers = floats;
```

Still, if a language allows the parametric type to be removed (type erasure), one can have a similar problem:

```
ArrayList<Float> floats = new ArrayList<Float>();

floats.add(new Float(1.0);

floats.add(new Float(2.0);

// Here the type parameter is erased

ArrayList numbers = floats;

// Now on can add any object to the ArrayList

number.add(new Double(2.2);
```

The problem exists only in mutable collections; if the collection were immutable the attempt to assign a non-substitutable value (or any other value) could be detected at compile time.