Methodology Handbook
Efficient Development of Safe Avionics Software with DO-178C Objectives Using SCADE Suite®

First Edition
CONTACTS

Headquarters
Esterel Technologies SAS
Parc Euclide - 8, rue Blaise Pascal
78990 Blaincourt
FRANCE
Phone: +33 1 30 68 61 60
Fax: +33 1 30 68 61 61
Submit questions to Technical Support at scade-support@esterel-technologies.com.
Contact one of our Sales representatives at scade-sales@esterel-technologies.com.
Direct general questions about Esterel Technologies to scade-info@esterel-technologies.com.
Discover latest news on our products and technology at www.esterel-technologies.com.

LOCAL SUPPORT SITES

United States
1082 North Alafaya Trail
Suite 124
FL 32826 Orlando
United States
Phone: +1 724-746-3304
Southern Europe
Esterel Technologies
Park Avenue - 9, rue Michel Labrousse
31100 Toulouse
FRANCE
Phone: +33 5 34 60 90 50

Central Europe
Esterel Technologies GmbH c/o ANSYS Germany GmbH
Birkenweg 14a
64295 Darmstadt
GERMANY
Phone: +49 6151 3644-112

China
ANSYS – Esterel Technologies
20F Verdant Place, No.128,
West Nanjing Road Huang Pu District,
Shanghai, P.R. CHINA
Phone: +86-21-63351885

DISCLAIMER: The content of this handbook is distributed for informational use only, is subject to change without notice, and should not be construed as a complete and contractual information from Esterel Technologies SAS for developing fully certified software. Although every precaution was taken to prepare this handbook, Esterel Technologies assumes no responsibility or liability for any errors that may be contained in this book or any damages resulting from the use of the information contained herein.

© 2015 Esterel Technologies SAS - A wholly-owned subsidiary of ANSYS, Inc. All rights reserved. ANSYS®, SCADE®, SCADE System®, SCADE Suite®, SCADE Display®, SCADE LifeCycle®, and SCADE Test are trademarks or registered trademarks of ANSYS, Inc. or its subsidiaries in the U.S. or other countries.

Shipping date: June 2015 Revision:SC-HB-DO178C-KCG64 - DOC/rev/86004-06
Abstract

This handbook provides detailed explanations on how to fully satisfy DO-178C objectives with a SCADE model-based approach while promoting an efficient development and verification strategy aimed at reducing costs and increasing productivity. The handbook reviews the regulatory guidelines before presenting the optimization of the development and verification processes that can be achieved with the SCADE Suite® methodology and tools. SCADE Suite supports the automated production and verification of a large part of the development life-cycle elements. The effect of using SCADE Suite together with its qualified KCG Code Generator is presented in terms of savings in the development and verification activities, following a step-by-step approach and considering the objectives that have to be met at each step.
# Table of Contents

1. **Document Background, Objectives, and Scope**
   1.1 Background
   1.2 Objectives and Scope
   1.3 Challenges in Airborne Software Development?
      1.3.1 Avoid multiple descriptions of the software
      1.3.2 Prevent ambiguity and lack of accuracy in specifications
      1.3.3 Avoid design and coding errors
      1.3.4 Allow efficient implementation of code on target
      1.3.5 Find specification and design errors as early as possible
      1.3.6 Lower complexity of updates
      1.3.7 Improve verification efficiency
      1.3.8 Provide efficient way to store Intellectual Property (IP)

2. **Development of Safety-Critical Airborne Software**
   2.1 ARP 4754A and DO-178C Guidelines
      2.1.1 Introduction
      2.1.2 ARP 4754A/ED-79
      2.1.3 DO-178C/ED-12C
      2.1.4 Relationship between ARP 4754A and DO-178C
      2.1.5 Development assurance levels
      2.1.6 Objective-oriented approach
      2.1.7 DO-178C processes overview
   2.2 DO-178C Development Processes
   2.3 DO-178C Verification Processes
      2.3.1 Objectives of software verification
      2.3.2 Reviews and analyses of the low-level requirements and architecture
      2.3.3 Reviews and analyses of the source code
      2.3.4 Software testing process
   2.4 DO-331 Model-Based Development and Verification Processes
      2.4.1 Model Definition
      2.4.2 Model Categorization
      2.4.3 Impact of Model-Based Development in DO-178C Development Processes
      2.4.4 Impact of Model-Based Development in DO-178C Verification Processes
      2.4.5 Model coverage analysis for design models
# Table of Contents

2.4.6 Model coverage criteria .................................................. 18
2.5 DO-330 Software Tools Qualification Attention ......................... 18
   2.5.1 Purpose of tool qualification ........................................ 19
   2.5.2 Tool criteria .......................................................... 19
   2.5.3 Tool Qualification Levels ............................................ 20
   2.5.4 Tool Stakeholders ................................................... 20
3. Model-Based Development with SCADE ...................................... 21
   3.1 What is SCADE? ................................................................ 21
   3.2 SCADE Modeling Techniques ........................................... 23
      3.2.1 Modeling behavior with SCADE Suite ............................ 23
         3.2.1.1 Familiarity and accuracy reconciled ........................ 23
         3.2.1.2 Scade operator .................................................. 24
         3.2.1.3 Data flow diagrams for continuous control ................ 25
         3.2.1.4 State Machines for discrete control ......................... 27
         3.2.1.5 Combining data and control flows ......................... 27
         3.2.1.6 Scade data typing .............................................. 28
      3.2.2 SCADE Suite cycle-based intuitive computation model ....... 28
      3.2.3 SCADE Suite as a model-based development environment .... 30
      3.2.4 SCADE Suite modeling and safety benefits .................... 31
4. Software Development Activities with SCADE Suite ....................... 33
   4.1 Overview of Software Development Activities ........................ 33
   4.2 Software Requirements Process ........................................ 34
   4.3 Software Design Process with SCADE Suite ......................... 35
      4.3.1 Architecture design ................................................ 35
      4.3.2 SCADE low-level requirements development .................. 37
      4.3.3 Re-usable components and library management ............... 39
      4.3.4 Robustness management ......................................... 41
   4.4 Software Coding Process ............................................... 42
      4.4.1 Code generation from SCADE Suite models .................... 43
      4.4.2 Code generation from multiple components ...................... 46
   4.5 Software Integration Process ........................................... 47
      4.5.1 Integration aspects ................................................. 47
      4.5.2 Interface with the external environment ........................ 47
# Table of Contents

4.5.3 SCADE module integration 48  
4.5.4 Integration of external data and code 48  
4.5.5 SCADE Suite scheduling and tasking 48  

4.6 Teamwork 52  

5. Software Verification Activities 53  
5.1 Overview 53  

5.2 Verification of High-Level Requirements 53  
5.2.1 Verification objectives for HLR 53  
5.2.2 Verification methods for HLR 54  
5.2.3 Verification summary for HLR 54  

5.3 Verification of the SCADE Low-Level Requirements and Architecture 55  
5.3.1 Verification objectives for the LLR and architecture 55  
5.3.2 Model accuracy and consistency 60  
5.3.3 Compatibility with target computer 61  
5.3.4 Conformity to standards 64  
5.3.5 Traceability from SCADE Suite LLR to HLR 64  
5.3.6 Algorithms accuracy 66  
5.3.7 Partitioning 66  
5.3.8 Verification of simulation cases, procedures and results (MB. specific objectives) 66  

5.4 Verification of Coding Outputs and Integration Process 69  
5.4.1 Verification objectives for coding output and integration process 69  
5.4.2 Impact of code generator qualification 69  
5.4.3 Verification of parameter data item 70  

5.5 Testing of Outputs from Integration Process 71  
5.5.1 Testing objectives for outputs from integration process 71  
5.5.2 SCADE Combined Testing Process overview 72  
5.5.3 Compliance of EOC with HLRs (MB.A-6 #1) and robustness with HLRs (MB.A-6 #2) 73  
5.5.4 Compliance of EOC to LLRs (MB.A-6 #3) 73  
5.5.5 Robustness of EOC with LLRs (MB.A-6 #4) 74  
5.5.6 Compatibility of EOC with target (MB.A-6 #5) 74  
5.5.7 Verification summary for testing outputs from integration process 75
Table of Contents

6. Verification of the Verification Activities 77
   6.1 Verification Objectives 77
   6.2 Verification of Test Procedures and Results 78
   6.3 HLR Coverage Analysis 78
   6.4 Coverage Analysis with SCADE Test Model Coverage 78
      6.4.1 Test Model Coverage overview 79
      6.4.2 SCADE Suite LLRs coverage analysis (MB.A-7#4) 79
         6.4.2.1 LLR coverage analysis with Model-Level Coverage 80
         6.4.2.2 Model coverage criteria 82
      6.4.3 Source code coverage analysis (from MB.A-7#5 to MB.A-7#8) 86
      6.4.4 Data and control coupling verification (MB.A-7#8) 86
         6.4.4.1 Definitions 86
         6.4.4.2 Verification of data and control coupling in SCADE Suite models 87
         6.4.4.3 Verification of data and control coupling between model and external environment 87
      6.4.5 Verification of additional code untraceable to source code (MB.A-7#9) 87
      6.4.6 Verification of simulation cases, procedures and results (MB.A-7#10, #11 and #12) 87
   6.5 Summary of Verification of Verification 88

Appendices and Index 91
A References 93
B Acronyms and Glossary 95
C DO-178C Qualification of SCADE Suite KCG 99
   C-1 What Does Qualification Mean and Imply? 99
   C-2 Development of SCADE Suite KCG 99
   C-3 SCADE Suite KCG Life-Cycle Documentation 100
D SCADE Suite Compiler Verification Kit (CVK) 101
   D-1 CVK Product Overview 101
   D-2 CVK Representativity 102
   D-3 Strategy for Developing SCADE Suite CVK 103
   D-4 Use of SCADE Suite CVK 104
INDEX 107
List of Figures

Figure 2.1: Relation between ARP 4754A and DO-178C processes 6
Figure 2.2: DO-178C documents structure 8
Figure 2.3: DO-178C life-cycle processes structure 10
Figure 2.4: DO-178C development processes 10
Figure 2.5: DO-178C testing process 13
Figure 3.1: Model-based development with SCADE products (Display, Suite, Test, LifeCycle) 21
Figure 3.2: SCADE addresses the applicative part of software 22
Figure 3.3: Control engineering view of a Controller 23
Figure 3.4: A software engineering view 23
Figure 3.5: Graphical notation for an integrator operator 24
Figure 3.6: Sample of SCADE Suite model data flows from a Flight Control system 25
Figure 3.7: Detection of a causality problem 26
Figure 3.8: Functional expression of concurrency in SCADE Suite 26
Figure 3.9: Detection of a flow initialization problem 26
Figure 3.10: Initialization of flows 26
Figure 3.11: Mixed data and control flows from Flight Control 28
Figure 3.12: Cycle-based execution model of SCADE Suite 29
Figure 4.1: Software development processes with SCADE Suite 33
Figure 4.2: Software design process with SCADE Suite 35
Figure 4.3: Top-level view of a simple Flight Control System 36
Figure 4.4: A first order filter 38
Figure 4.5: Complex display logic and simple functions 38
Figure 4.6: State machine in Flight mode management 39
Figure 4.7: Concept of SCADE Suite library 40
Figure 4.8: Example of generic operator instantiated with int and bool types 40
List of Figures

Figure 4.9: Example of operator parameterized by size 40
Figure 4.10: Inserting Confirmator in Boolean input flow 41
Figure 4.11: Inserting Limiter in output flow 42
Figure 4.12: The software coding process with SCADE Suite 43
Figure 4.13: Scade data flow to generated C source code traceability 44
Figure 4.14: SCADE State Machine to generated C source code traceability 44
Figure 4.15: Sample of kcg_trace.xml for traceability between Scade model and source code 45
Figure 4.16: Non-expanded and Expanded modes 45
Figure 4.17: Code generation and multiple components 47
Figure 4.18: SCADE Suite execution semantics 48
Figure 4.19: SCADE Suite code integration 49
Figure 4.20: Modeling a bi-rate system 50
Figure 4.21: Timing diagram of a bi-rate system 50
Figure 4.22: Modeling slow system over four cycles 51
Figure 4.23: Timing diagram of distributed computations 51
Figure 4.24: Typical teamwork organization 52
Figure 5.1: Verification cases creation and management in Test Environment for Host 57
Figure 5.2: Simulation results from running verification cases on host 57
Figure 5.3: Model coverage analysis with SCADE Test Model Coverage 58
Figure 5.4: Traceability information analysis with Requirements Management Gateway 59
Figure 5.5: Observer operator containing landing gear safety property 60
Figure 5.6: Connecting the observer operator to the landing gear controller 60
Figure 5.7: Timing and Stack analysis global visualization 63
Figure 5.8: Timing Verifier analysis reports 63
Figure 5.9: Capturing LLR/HLR traceability links with Requirements Management Gateway 65
List of Figures

Figure 5.10: Built-in report selection or customization of traceability report for HLR/LLR 65
Figure 5.11: Factor simulation and test cases with SCADE Test 72
Figure 5.12: SCADE Combined Testing Process 72
Figure 5.13: Factor simulation and test cases with SCADE Test Target Execution 73
Figure 5.14: Robustness Strategy with SCADE Suite design 74
Figure 6.1: Test Model Coverage as integrated solution for model and code coverage analysis 79
Figure 6.2: Position of SCADE Test Model Coverage within verification flow 80
Figure 6.3: Model coverage analysis/resolution with SCADE Test Model Coverage 81
Figure 6.4: Non activated Confirmator 82
Figure 6.5: Uncovered “reset” activation 82
Figure 6.6: Data Path Activity concept within SCADE Test Model Coverage 83
Figure 6.7: MC/DC Coverage of a Boolean expression 84
Figure 6.8: Predefined integration coverage criteria 85
Figure D.1: Role of KCG and CVK in verification of user development environment 101
Figure D.2: Strategy for developing and verifying CVK 103
Figure D.3: Use of CVK items in user processes 104
Figure D.4: Position of CVK items in the Compiler Verification Process 105
List of Tables

Table 2.1: Top-Level function DAL assignment 7
Table 2.2: Example of test cases 15
Table 2.3: Model usage examples (DO-331 Table MB.1-1) 16
Table 3.1: Components of Scade functional modules: operators 24
Table 5.1: DO-178C Table A-3 53
Table 5.2: DO-178C Table A-3 Objectives Achievement 54
Table 5.3: DO-331 Table MB.A-4 55
Table 5.4: DO-331 Table MB.A-4 Objectives Achievement 67
Table 5.5: DO-331 Table MB.A-5 69
Table 5.6: DO-331 Table MB.A-5 Objectives Achievement 71
Table 5.7: DO-331 Table MB.A-6 71
Table 5.8: DO-331 Table MB.A-6 Objectives Achievement 75
Table 5.9: MC/DC Activation Conditions for Boolean operators 84
Table 6.1: User-defined integration coverage criteria 85
Table 6.2: Coverage criteria in SCADE Test Model Coverage 86
Table 6.3: DO-331 Table MB.A-7 88
Table C.1: Documents delivered for KCG qualification audit by Certification Authorities 100
1. Document Background, Objectives, and Scope

1.1 Background

The avionics industry has a very long tradition of rigorous software development. The function and architecture of an embedded software system (i.e., Flight Control, Braking, Cockpit Display, etc.) are defined by system engineers; the associated control laws are developed by control engineers using some informal notation or a semi-formal notation mainly based on schema-blocks and/or state machines; and the embedded production software is finally specified textually and coded by hand in C or Ada by software engineers.

In this context, the support of a model-based qualified tool chain (including but not limited to qualified code generation) carries strong Return On Investment (ROI), while preserving the safety of the application. Basically, the idea is to describe the application through a software model, including the control laws as described above, and to automatically generate the code from this model using a code generator qualified with respect to DO-330. All of this has several advantages for the development life cycle when a proper modeling approach is defined:

- It fulfills the needs of the control engineers, typically using such notations as data flow diagrams and state machines.
- It fulfills the needs of the software engineers by supporting the accurate definition of the software requirements and by providing efficient automatic code generation of software having the qualities that are expected for such applications (i.e., efficiency, determinism, static memory allocation, etc.).
  - It allows for establishing efficient new processes to ensure that DO-178C objectives are met.
  - It saves coding time, as this is automatic.
  - It saves a significant part of verification time, as the use of such tools guarantees that the generated source code conforms to the software model.
  - It allows for identifying problems earlier in the development cycle, since most of the verification activities can be carried out at model level.
  - It reduces the change cycle time, since modifications can be done at model level and code can automatically be regenerated.

1.2 Objectives and Scope

This document provides a careful explanation of a DO-178C compliant software life cycle as described in DO-178C and DO-331 guidelines [DO-178C] and [DO-331]. It also presents a quick overview of ARP 4754A compliant system life cycle (see [ARP-HB] for more information). The rest of the document explains how the use of proper modeling techniques and qualified code generation from models can drastically improve productivity in the development and verification of safety critical software. It is organized as follows:
Section 2. This section provides an introduction to the regulatory guidelines of ARP 4754A. DO-178C and DO-331 used when developing embedded aeronautics systems and software. It then describes the main challenges in the development of safety-critical applications, in terms of specification, verification, and efficiency of the resulting software. It also addresses Tools Qualification considerations according to the DO-330 guidelines.

Section 3. This section presents an overview of SCADE Suite methodology and tools, including how SCADE Suite achieves the highest-quality standards while reducing costs thanks to model-based development and verification, with a strong emphasis on the following points:

- A unique and accurate software description, which enables the prevention of many specification or design errors, can be shared among all project participants.
- The early identification of design errors makes it possible to fix them in the requirements/design phase rather than in the testing or integration phase.
- Qualified code generation not only saves writing the code by hand, but also the cost of verifying it.

Section 4. This section is devoted to the software development activities using SCADE Suite tools, including the use of the SCADE Suite KCG qualified code generator. It also presents the integration of SCADE Suite-generated code on target, including when it has to be connected with an RTOS (Real-Time Operating System).

Section 5, and Section 6. These sections present the verification activities to be performed when using SCADE Suite. Several model-based verification methods and techniques are presented. They rely on various verification modules of the SCADE Suite, SCADE Test, and SCADE LifeCycle products.

This document contains several appendices:

Appendix A provides a reference list.
Appendix B lists all acronyms used in this document and explains key terminology in a glossary.
Appendix C details the KCG qualification process.
Appendix D details the Compiler Verification Kit (SCADE Suite CVK).

The concepts and methodology described in this document are applicable from the following SCADE product configuration:

- SCADE Suite R16, including SCADE Suite KCG 6.4
- SCADE Test Model Coverage 6.4.5
- SCADE Test Environment for Host and SCADE Test Target Execution R16.2
- SCADE LifeCycle R16
1.3 Challenges in Airborne Software Development?

This section introduces the main challenges that a company faces when developing safety-critical airborne software.

1.3.1 Avoid multiple descriptions of the software

In such a process, software development is divided into several phases from software requirements phase to the coding phase with their outputs.

At each step, it is important to try to avoid rewriting the software description. This rewriting is not only expensive, it is also error-prone. Major risks of inconsistencies between different descriptions are very likely. This necessitates devoting a significant effort to the compliance verification of each level with the previous level. The purpose of many activities, as described in [DO-178C], is to detect the errors introduced during transformations from one written form to another.

1.3.2 Prevent ambiguity and lack of accuracy in specifications

Requirements and design specifications are traditionally written in some natural language, possibly complemented by non-formal figures and diagrams. Natural language is an everyday subject of interpretation, even when it is constrained by requirements standards. Its inherent ambiguity can lead to different interpretations depending on the reader.

This is especially true for any text describing the dynamic behavior. For instance, how does one interpret the combination of fragments from several sections of a document, such as “A raises B,” “if both B and C occur, then set D,” “if D or Z are active, then reset A”?

1.3.3 Avoid design and coding errors

Coding is the last transformation in a traditional development life cycle. It takes as input the last formulation in natural language (or pseudo-code). Since programmers have generally a limited understanding of the system, they are sensitive to ambiguities in the specification. Moreover, the code they produce is generally not understandable by the author of the system specification.

In the traditional approach, the combined risk of interpretation and coding errors is so high that a major part of the software life-cycle’s verification effort is consumed by code testing.

1.3.4 Allow efficient implementation of code on target

Code that is produced must be simple, deterministic, and efficient. It should require as few resources as possible, in terms of memory and execution time. It should be easily and efficiently re-targetable to a given processor.
1.3.5 Find specification and design errors as early as possible

Many specification and design errors are only detected during software integration testing. One cause of this is that the requirement/design specification is often ambiguous and subject to interpretation. The other cause is that it is difficult for a human reader to understand details regarding dynamic behavior without being able to exercise it. In a traditional process, the first time one can exercise the software is during integration. This is too late in the process. When a specification error can only be detected during the software integration phase, the cost of fixing it is much higher than if it had been detected during the specification phase.

1.3.6 Lower complexity of updates

There are many sources of changes in the software, ranging from fixing defects to function improvement or the introduction of new functions. When something has to be changed in the software, all products of the software life cycle have to be updated consistently, and all verification activities must be performed accordingly.

1.3.7 Improve verification efficiency

The level of verification for safety-critical airborne software is much higher than for other non safety-critical commercial software. For high-integrity software, the overall verification cost (including testing) may account for up to 80 percent of the development budget. Verification is also a bottleneck to project completion. So, clearly, any change to the speed and/or cost of verification has a direct impact on project time and budget.

1.3.8 Provide efficient way to store Intellectual Property (IP)

A significant part of aircraft or equipment companies' know-how resides in software. It is therefore of utmost importance to provide tools and methods to efficiently store and access Intellectual Property (IP) relative to these safety-critical systems. Such IP vaults contain:

- Textual system and software safety requirements
- Graphical models of the software requirements
- Source code
- Test cases and procedures
- Other
2. Development of Safety-Critical Airborne Software

2.1 ARP 4754A and DO-178C Guidelines

2.1.1 Introduction

The aeronautics industry requires that safety-critical software be assessed according to strict certification authority\(^1\) guidelines before it may be used on any commercial aircraft. ARP 4754A and DO-178C are guidelines used both by the companies developing airborne equipment and by the certification authorities.

2.1.2 ARP 4754A/ED-79

The Aerospace Recommended Practice ARP4754A is the Guidelines For Development Of Civil Aircraft and Systems. It is issued from SAE International, dealing with the development processes which support certification of Aircraft systems. Revision A was released in December 2010. It was recognized by the FAA and by EASA. EUROCAE jointly issues the document as [ED-79].

This document discusses the certification aspects of highly integrated or complex systems installed on an aircraft, taking into account the overall aircraft operating environment and functions. The term “highly integrated” refers to systems that perform or contribute to multiple aircraft-level functions.

The material is also applicable to engine systems and related equipment. See [ARP-HANDBOOK], chapter 3 for deeper information on ARP 4754A.

ARP 4754A excludes specific coverage of detailed aspects, including software and hardware design processes beyond those of significance in establishing the safety of the implemented system. More detailed coverage of the software aspects of design are dealt within the RTCA DO-178C document [DO-178C]. Coverage of complex hardware aspects of design are dealt with in document RTCA DO-254 [DO-254].

2.1.3 DO-178C/ED-12C

DO-178C, the new version of “Software Considerations in Airborne Systems and Equipment Certification” was published in 2011 by RTCA, Inc., in a joint effort with EUROCAE. This replaces DO-178B as the primary document by which certification authorities such as FAA, EASA, Transport Canada, CAAC, ANAC, and ARMAK will approve all commercial software-

---

\(^1\) For example, the United States Federal Aviation Administration (FAA), the European Aviation Safety Agency (EASA), ARMAK (Russia), Transport Canada, Brazil’s Agência Nacional de Aviação Civil (ANAC), Civil Aviation Administration of China (CAAC).
based aerospace systems. The new document is called DO-178C/ED-12C and was completed in November 2011 and approved by the RTCA in December 2011. This document was approved in July 2013 by the FAA (see [AC 20-115]) and in September 2013 by the EASA (see [AMC 20-115]), making it a recognized “means of compliance with the applicable airworthiness regulations for the software aspects of airborne systems”.

The objective of the guidelines is to ensure that software performs its intended function with a level of confidence in safety that complies with airworthiness requirements.

These guidelines specify:

- Objectives for software life-cycle processes.
- Description of activities and design considerations for achieving those objectives.
- Description of the evidence indicating that the objectives have been satisfied.

2.1.4 Relationship between ARP 4754A and DO-178C

ARP 4754A and DO-178C are complementary guidelines:

- ARP 4754A provides guidelines for the system life-cycle processes.
- DO-178C provides guidelines for the software life-cycle processes.

The information flow between the system and software processes is summarized in Figure 2.1.

DO-178C (§2.2.1) provides the list of data that is passed from the system processes to the software life cycle processes:

a. “System requirements allocated to software.

b. System safety objectives.

c. Software level for software components and a description of associated failure condition(s), if applicable.

d. System description and hardware definition.

e. Design constraints, including external interfaces, partitioning requirements, etc.

f. Details of any system activities proposed to be performed as part of the software life cycle. Note that system requirement validation is not usually part of the software life cycle processes. The system life cycle processes are responsible for assuring any system activities proposed to be performed as part of the software life cycle.

g. Evidence of the acceptability, or otherwise, of any data provided by the software processes to the system processes on which any activity was conducted by the system processes.
Examples of such activity are the system processes' evaluations of:
1. Derived requirements provided by the software processes to determine if there is any impact on the system safety assessment and system requirements.
2. Issues raised by the software processes with respect to the clarification or correction of system requirements allocated to software.
3. Evidence of software verification activities performed by the system life cycle processes, if any.

On the other side DO-178C (§2.2.2) provides the information flow from Software Processes to System Processes:

a. Details of derived requirements created during the software life cycle processes.
b. A description of the software architecture, including software partitioning.
c. Evidence of system activities performed by the software life cycle processes, if any.
d. Problem or change documentation, including problems identified in the system requirements allocated to software and identified incompatibilities between the hardware and the software.
e. Any limitations of use.
f. Configuration identification and any configuration status constraints.
g. Performance, timing, and accuracy characteristics.
h. Data to facilitate integration of the software into the system.
i. Details of software verification activities proposed to be performed during system verification, if any.”

2.1.5 Development assurance levels

ARP 4754A defines guidelines for the assignment of so-called “Development Assurance Levels” (DAL) to the system, to its components, and to software, with regard to the most severe failure condition of the corresponding part.

ARP 4754A defines a DAL for each item and allocates a Software Level to each software component as summarized in the following table:

<table>
<thead>
<tr>
<th>Level</th>
<th>Effect of anomalous behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Catastrophic failure condition for the aircraft (e.g., aircraft crash).</td>
</tr>
<tr>
<td>B</td>
<td>Hazardous/severe failure condition for the aircraft (e.g., several persons could be injured).</td>
</tr>
<tr>
<td>C</td>
<td>Major failure condition for the aircraft (e.g., flight management system could be down, the pilot would have to do it manually).</td>
</tr>
<tr>
<td>D</td>
<td>Minor failure condition for the aircraft (e.g., some pilot-ground communications could have to be done manually).</td>
</tr>
<tr>
<td>E</td>
<td>No effect on aircraft operation or pilot workload (e.g., entertainment features may be down).</td>
</tr>
</tbody>
</table>
2.1.6 DO-178C documents structure

The DO-178C Standard is composed of a core document and a set of supplements as illustrated in Figure 2.2.

**Figure 2.2: DO-178C documents structure**

DO-178C “Software Considerations in Airborne Systems and Equipment Certification” is the core document. It defines a set of common objectives and activities for each process considered in the production of software for airborne systems and equipment (see §2.1.7 and §2.1.8 for further information on DO-178C objectives and processes).

This core document is completed by supplements that shall be considered, depending on the techniques used for the production of the airborne software:

- **DO-331 “Model-based Development and Verification Supplement”** supplements the guidance given in DO-178C (core document) for the software components developed with model-based techniques (see Section 2.4).
- **DO-332 “Object-Oriented Technology and Related Techniques Supplement”** is applicable when object-oriented technology or related techniques are used as part of the software development life cycle. This supplement, in conjunction with DO-178C, is intended to provide a common framework for the evaluation and acceptance of object-oriented technology (OOT) and related techniques (RT)-based systems.
- **DO-333 “Formal Methods Supplement”** is applicable in conjunction with DO-178C when Formal Methods are used as part of the software life cycle. Formal methods are mathematically-based techniques for the specification, development and verification of software aspects of systems.

**COMMON STRUCTURE FOR ALL DO-178C DOCUMENTS**

Each supplement has the same structure as the core document (i.e., section titles are the same). For any unchanged section, the supplement explicitly states there is no change and does not repeat the core document.

On the other side, each supplement identifies the additions, modifications, and substitutions to DO-178C for a given technique:

- New and/or revised activities, explanatory text and software life cycle data are highlighted in the body of the supplement within existing sections or dedicated new sections.
- New and/or revised objectives are displayed in the Annex A of the supplement.

Two other documents can also be considered in the context of DO-178C (see Figure 2.2):

- **DO-330 “Software Tools Qualification Considerations”**: this standalone document (it is not considered as a supplement to DO-178C) defines the Tool Qualification Processes for both tool users and tool developers. It is interesting to note that the DO-330 document, as a standalone document, enables
SCADE Suite with DO-178C Objectives

and encourages the use of this guidance outside the airborne software domain.

• DO-248C “Supporting Information for DO-178C” addresses the questions of both the industry and regulatory authorities. It contains frequently asked questions (FAQs), discussion papers (DPs) and rationale.

2.1.7 Objective-oriented approach

The approach of DO-178C is based on the formulation of appropriate objectives and on the verification that these objectives have been achieved. The DO-178C authors acknowledged that objectives are more essential and stable than specific procedures. The ways of achieving an objective may vary from one company to another, and they may vary over time with the evolution of methods, techniques, and tools. DO-178C never states that one should use design method X, coding rules Y, or tool Z. DO-178C does not even impose a specific life cycle.

The general approach is the following:

• Ensure appropriate goals are defined. For instance:
  a  Software level
  b  Design standards

• Define procedures for the verification of these goals. For instance:
  a  Verify that independence of activities matches the software level
  b  Verify that design standards are met and that the design is complete, accurate, and traceable

• Define procedures for verifying that the above-mentioned verification activities have been performed satisfactorily. For instance:
  a  Reviews of requirements-based test cases and procedures is achieved
  b  Coverage of requirements by testing is achieved

2.1.8 DO-178C processes overview

DO-178C structures activities as a hierarchy of “processes”, as illustrated in Figure 2.3. The term “process” appears several times in the document. DO-178C defines three top-level groups of processes:

• The software planning process that defines and coordinates the activities of the software development and integral processes for a project.

• The software development processes that produce the software product. These processes are the software requirements process, the software design process, the software coding process, and the integration process.

• The integral processes that ensure the correctness, control, and confidence of the software life-cycle processes and their outputs. The integral processes are the software verification process, the software configuration management process, the software quality assurance process, and the certification liaison process. The integral processes are performed concurrently with the software development processes and the planning process throughout the software life cycle.
2.2 DO-178C Development Processes

The software development processes, as illustrated below in Figure 2.4, are composed of:

- The software requirements process, which produces the high-level requirements (HLR)
- The software design process, which produces the low-level requirements (LLR) and the software architecture through one or more refinements of the HLR
- The software coding process, which produces the source code and object code
- The integration process, which produces executable object code and builds up to the integrated system or equipment
The high-level software requirements (HLR) are produced directly through analysis of system requirements and system architecture and their allocation to software.

They include specifications of functional and operational requirements, timing and memory constraints, hardware and software interfaces, failure detection and safety monitoring requirements, as well as partitioning requirements.

The HLR are further developed during the software design process, thus producing the software architecture and the low-level requirements (LLR). These include descriptions of the input/output, the data and control flow, resource limitations, scheduling and communication mechanisms, as well as software components. If the system contains “deactivated” code (see Appendix B), the description of the means to ensure that this code cannot be activated in the target computer is also required.

Through the coding process, the low-level requirements (LLR) are implemented as source code.

The source code is compiled and linked by the integration process into an executable code loaded on the target environment.

At all stages of the development process, traceability is required: between system requirements and HLR; between HLR and LLR; between LLR and source code; and also between requirements and tests.

### 2.3 DO-178C Verification Processes

#### 2.3.1 Objectives of software verification

The purpose of the software verification processes is “to detect and report errors that may have been introduced during the software development process.” DO-178C defines verification objectives, rather than specific verification techniques, since the later may vary from one project to another and/or over time.

Testing is part of the verification processes, but verification is not just testing: the verification processes also rely on reviews and analyses. Reviews are qualitative and comply with DO-178C (§6.3), whereas analyses are more detailed and should be reproducible (e.g., compliance with coding standards).

Verification activities cover all the processes, from the planning process to the development processes; there are even verifications of the verification activities.
2.3.2 Reviews and analyses of the high-level requirements

The objective of reviews and analyses is to confirm that the HLRs satisfy the following:

- **Compliance with system requirements**
- **Accuracy and consistency**: each HLR is accurate, unambiguous and sufficiently detailed; requirements do not conflict with each other
- **Compatibility with target computer**
- **Verifiability**: each HLR has to be verifiable
- **Compliance with standards** as defined by the planning process
- **Traceability** with the system requirements
- **Algorithm accuracy**

2.3.3 Reviews and analyses of the low-level requirements and architecture

The objective of these reviews and analyses is to detect and report errors possibly introduced during the software design process. These reviews and analyses confirm that the software LLRs and architecture satisfy the following:

- **Compliance with high-level requirements**: the software LLRs satisfy the software HLRs
- **Accuracy and consistency**
- **Compatibility with target computer**: no conflicts exist between the software requirements and the hardware/software features of the target computer, especially the use of resources (e.g., bus loading), system response times, and input/output hardware
- **Verifiability**: each LLR can be verified
- **Compliance with Software Design Standards** as defined by the software planning process
- **Algorithm aspects**: ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities (e.g., mode changes, crossing value boundaries)
- **The Software Architecture is compatible with the HLR**: is consistent and compatible with the target computer, is verifiable, and conforms to standards
- **Software partitioning integrity** is confirmed

2.3.4 Reviews and analyses of the source code

The objective is to detect and report errors that may have been introduced during the software coding process. These reviews and analyses confirm that the outputs of the software coding process are accurate, complete, and can be verified. Primary concerns include correctness of the code with respect to the LLRs and the software architecture, and compliance with the Software Code Standards. These reviews and analyses are usually confined to the source code. The topics should include:

- **Compliance with low-level requirements**: the source code is accurate and complete with respect to the software LLRs; no source code implements an undocumented function
- **Compliance with software architecture**: the source code matches the data flow and control flow defined in the software architecture
• **Verifiability**: the source code does not contain unverifiable statements and structures, and the code does not have to be altered to test it.

• **Compliance with standards**: the Software Code Standards (defined by the software planning process) were followed during the development of the code, especially complexity restrictions and code constraints that would be consistent with the system safety objectives. 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.

• **Traceability**: the source code implements all software LLRs

• **Accuracy and consistency**: the objective is to determine the correctness and consistency of the source code, including stack usage, fixed-point arithmetic overflow and resolution, resource contention, worst-case execution timing, exception handling, use of non initialized variables or constants, unused variables or constants, and data corruption due to task or interruption conflicts.

2.3.5 Software testing process

Testing of aeronautics software has two complementary objectives. One objective is to demonstrate that the software satisfies its requirements. The second objective is to demonstrate, with a high degree of confidence, that all errors, which could lead to unacceptable failure conditions as determined by the system safety assessment process, have been removed.

There are three types of testing activities:

• **Low-level testing**: to verify that each software component complies with its LLRs

• **Software integration testing**: to verify the interrelationships between software requirements and components and to verify the implementation of the software requirements and software components within the software architecture

• **Hardware/software integration testing**: to verify correct operation of the software in the target computer environment

As shown in Figure 2.5, DO-178C dictates that all test cases, including low-level test cases, be requirements-based; namely that all test cases be defined from the requirements and the error sources inherent to the software development.
processes, but never from the code. When it is not possible to verify specific software requirements by exercising the software in a realistic test environment, other means and their justification shall be provided according to DO-178C, §6.2b as illustrated by Additional Verification Considerations in Figure 2.5.

TEST COVERAGE ANALYSIS
Test coverage analysis is a two-step activity:

1. Requirements-based test coverage analysis determines how well the requirement-based testing covered the software requirements. The main purpose of this step is to verify that all requirements have been implemented. Requirements-based Test coverage analysis shall be considered for both HLRs and LLRs.

2. Structural coverage analysis determines which code structures including interfaces between components, are exercised by requirement-based test procedures. Its purposes are:
   - Ensures all code structures, including interfaces, was executed at least once
   - Finds unintended and untested functions
   - Identifies extraneous code, including dead code (see glossary in Appendix B)
   - Helps to confirm if deactivated code is truly deactivated
   - Serves as completion criteria for testing efforts

STRUCTURAL COVERAGE RESOLUTION
If structural coverage analysis reveals code structures including interfaces that were not exercised, resolution is required:

- If it is due to shortcomings in the test cases, then test cases should be supplemented or test procedures changed.
- If it is due to inadequacies in the requirements, then the requirements must be changed and test cases developed and executed.
- If it is extraneous code, including dead code (i.e., it is not traceable to any system or software requirement and its presence is an error), then this code should be removed.
- If it is deactivated code (it cannot be executed, but its presence is not an error):
  - If it is not intended to be executed in any configuration, then analysis and testing should show that the means by which such code could be inadvertently executed are prevented, isolated, or eliminated.
  - If it is only executed in certain configurations, the operational configuration for execution of this code should be established and additional test cases should be developed to satisfy coverage objectives.

STRUCTURAL COVERAGE CRITERIA
The structural coverage criteria that have to be achieved depend on the software level:

- **Level C**: Statement Coverage is required, meaning every statement in the source code was exercised.
- **Level B**: Decision Coverage is required, meaning every entry and exit point in the program was invoked at least once and every decision has taken all possible outcomes at least once (e.g., the outcome of an “if” construct was true and false, even if there is no “else”).
- **Level A**: MC/DC (Modified Condition/Decision Coverage) is required, meaning:
• every entry and exit point in the program was invoked at least once;
• every condition in a decision has taken all possible outcomes at least once;
• every decision in the program has taken all possible outcomes at least once;
• each condition in a decision was shown to independently affect that decision’s outcome. 
This may be shown by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition, while holding fixed all other possible conditions that could affect the outcome.

For instance, the following fragment requires four test cases, as shown below in Table 2.2.

```
If A or (B and C)
Then do action1
Else do action2
Endif
```

Table 2.2: Example of test cases

<table>
<thead>
<tr>
<th>Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>2</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>3</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>4</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

2.4 DO-331 Model-Based Development and Verification Processes

Model-based techniques are more and more used in the design of safety critical software components because they are considered as a very efficient approach to develop complex software while increasing productivity. The DO-331 technical supplement in conjunction with the DO-178C core document (see §2.1.6) are the applicable standards when model-based techniques are used for the development and verification of a given software component.

2.4.1 Model Definition

According to the DO-331 glossary, a model is “an abstract representation of a given set of aspects of a system that is used for analysis, verification, simulation, code generation, or any combination thereof. A model should be unambiguous, regardless of its level of abstraction.”

DO-331, MB.1.0 addresses model(s) that have the following characteristics:

a. “The model is completely described using an explicitly identified modeling notation. The modeling notation may be graphical and/or textual.

b. The model contains software requirements and/or software architecture definition.

c. The model is of a form and type that is used for direct analysis or behavioral evaluation as supported by the software development process or the software verification process.”

2.4.2 Model Categorization

DO-331, MB.1.6.2 defines two types of models: specification model and design model.

A Specification Model represents “high-level requirements that provides an abstract representation of functional performance, interface, or safety characteristics
of software components”. It supports an understanding of software functionality and does not prescribe a specific software implementation or architecture.

A **design model** defines “any software design such as low-level requirements, software architecture, algorithms, component internal data structures, data flow and/or control flow.” It describes in particular the internal details of a given software component.

Moreover, two important properties are attached to these concepts of model:

- A model cannot be categorized as both specification model and design model;
- Whatever the model (specification or design), there must be requirements above the model. They should be external to the model and should be a complete set of requirements and constraints.

### 2.4.3 Impact of Model-Based Development in DO-178C Development Processes

**DO-331 Table MB.1-1** provides some examples of Model usage in the context of industrial projects that illustrate different strategies for Model-Based Development (MBD).

<table>
<thead>
<tr>
<th>Process generating life-cycle data</th>
<th>MB Example 1</th>
<th>MB Example 2</th>
<th>MB Example 3</th>
<th>MB Example 4</th>
<th>MB Example 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Requirements and System Design Processes</td>
<td>Requirements allocated to software</td>
<td>Requirements from which the model is developed</td>
<td>Requirements from which the model is developed</td>
<td>Requirements from which the model is developed</td>
<td>Requirements from which the model is developed</td>
</tr>
<tr>
<td>Software Requirements and Software Design Processes</td>
<td>Requirements from which the model is developed</td>
<td>Specification Model</td>
<td>Specification Model</td>
<td>Design model</td>
<td>Design model</td>
</tr>
<tr>
<td></td>
<td>Design model</td>
<td>Design model</td>
<td>Textual description</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the context of “MB Example 1”, the DO-178C traditional development process such as described in Section 2.2 can be significantly improved as follow:

- Software requirements are textual requirements derived from System Requirement Allocated to Software (SRATS)
- A design model is developed for LLRs and Architecture
- Source Code is developed with the support of an automatic code generator
2.4.4 Impact of Model-Based Development in DO-178C Verification Processes

Various verification techniques are available when model-based development is used. In particular, Model Simulation can be considered as one of the most efficient.

DO-331 provides a precise definition and some specific guidance in §MB.6.8: Model Simulation is defined as “The activity of exercising the behavior of a model using a model simulator”. In this context, the model simulator may or may not be executing code that is representative of the target code. It is different of testing that means exercising the “real” Executable Object Code (EOC) on target.

Model Simulation supports the verification of objectives of DO-331, Table MB.A-4, such as

- Compliance with HLRs for models containing LLRs
- Accuracy and consistency
- Verifiability
- Algorithm aspects

On the other side, Model Simulation cannot be used to satisfy objectives such as compatibility with the target computer, conformance to standards, traceability or partitioning integrity. Reviews and analyses are then required to complete the model verification.

If Model Simulation is used for verification to satisfy a DO-178C objective, the model simulator shall be qualified as a verification tool (see Section 2.5 for more information on tool qualification) and new DO-331 objectives shall be considered during the verification of the software LLRs.

In particular, the following objectives are considered in addition to the existing objectives described in Section 2.3.3 (a-i):

- “Simulation cases are correct” (Table MBA-4 objective MB14)
- “Simulation procedures are correct” (Table MBA-4 objective MB15)
- “Simulation results are correct and discrepancies explained” (Table MBA-4 objective MB16)

2.4.5 Model coverage analysis for design models

Model coverage analysis concerns so-called “design models” (as opposed to “specification models”). Model coverage analysis determines which requirements expressed by the design model were not exercised by verification based on the requirements from which the design model was developed. This analysis may assist in particular in finding unintended functionality in the design model.

Model coverage analysis is different than structural coverage analysis. Both model coverage and structural code coverage shall be achieved to satisfy objectives of DO-331 table MBA-7.

DO-331 objective MBA-7#4 identifies model coverage (§6.7) as a supporting activity for assessing coverage of the low-level requirements contained in a design model.
Moreover, as identified in MB.6.7.2, model coverage analysis contributes to the detection and resolution of:

a Shortcomings in requirements-based verification cases or procedures
b Inadequacies or shortcomings in requirements from which the design model was developed
c Derived requirements expressed by the model
d Deactivated functionality expressed by the design model
e Unintended functionality expressed by the design model

2.4.6 Model coverage criteria

There is a large diversity of modeling notations that differ significantly regarding for instance the following aspects:

• Degree of formality: modeling notations range from non-formal (e.g., UML) to formal (e.g., Scade, SDL)
• They may be based on various concepts and representations such as data flow, state machines, sequence charts
• They may be synchronous (e.g. Scade) and/or asynchronous (e.g., SDL, UML)

The committee agreed that it was not possible to impose specific detailed model coverage criteria in DO-331, and the following approach was retained:

1 DO-331 provides general principles for model coverage criteria and requires model coverage criteria to be defined in the Software Verification Plan.

2 Each applicant defines in its SVP the model coverage criteria that it will apply.

See [DO-331], Table MB.6-1 for an example of Model Coverage Criteria compatible with DO-178C, §6.4.2.1 and §6.4.2.2.

2.5 DO-330 Software Tools Qualification Attention

Efficient software development and verification techniques (including model-based) rely on tools to automate, reduce, or eliminate some activities.

DO-330 (see Section 2.1.6) glossary defines a tool as “A computer program or a functional part thereof, used to develop, transform, test, analyze, produce, or modify another program, data, or its documentation”. Typical examples are automated code generators, compilers or test tools.

Tool qualification guidance was expanded and separated from the DO-178C core document for the following main reasons:

• The nature of tools is different from the nature of software using the tools. It is not relevant to apply airborne-related guidance in the context of the development and verification of a software tool.
• This standalone document can be used by other domains than airborne, for instance, in the context of system and/or hardware tools.
2.5.1 Purpose of tool qualification

Qualification of a tool is needed when processes required by DO-178C are eliminated, reduced or automated by the use of a software tool without its output being verified.

The purpose of the tool qualification process is to obtain confidence in the tool functionality. The tool qualification effort varies based upon the potential impact that a tool error could have on the system safety and upon the overall use of the tool in the software life cycle process. The higher the risk of the tool error adversely affects system safety, the higher the rigor is required for tool qualification.

Tool qualification is the process necessary to obtain certification credit for a tool. This credit may only be granted within the context of a project requiring approval.

2.5.2 Tool criteria

The qualification level of a tool is based on the tool use and its potential impact in the software life cycle process.

DO-178C core section 12.2.2 defines 3 criteria to determine the impact of a tool:

- **Criteria 1**: A tool whose output is part of the resulting software and thus could insert an error.
- **Criteria 2**: A tool that automates verification process(es) and thus could fail to detect an error, and whose output is used to justify the elimination or reduction of:
  - Verification process(es) other than that automated by the tool, or;
  - Development process(es) (which could have an impact on the resulting software).
- **Criteria 3**: A tool that, within the scope of its intended use, could fail to detect an error.

Moreover, DO-330, §1.5.3.3 provides additional considerations about the tool criteria selection and some examples:

“Criteria 1 is applied to the tools that automatically produce a part of the outputs of one of the software development processes, whatever the input and output format are. This criteria encompasses the tools that transform a higher level of requirements to a lower requirement level (or same level but in a different formalism), to Source Code, to data files, to configuration files, or to Executable Object Code. [...] Criteria 2 and Criteria 3 are applied to all tools that verify or analyze software life cycle data, compute software characteristics, etc. Application of one of these two criteria differs based on the certification/approval credit claimed by the applicant.

a If the certification/approval credit claim is only for the objective directly satisfied by the activity performed by the tool, criteria 3 is applied.

b An alternative for the applicant is to claim that other objectives are also satisfied or partially satisfied through the use of the tool. In this case, criteria 2 applies.”

A typical example can be that a static code analyzer may be used to automate some verification of Source Code review. Criteria 3 could be applied based on this tool’s usage and credit claimed. However, if the applicant claims not to include some specific mechanisms in the resulting software in order to detect and treat
possible overflows and run-time errors based on the confidence of the tool, then Criteria 2 becomes applicable. In this case, it corresponds to “a reduction of software development process(es)”. 

### 2.5.3 Tool Qualification Levels

The Tool Qualification Level (TQL) for a given tool is based on the tool qualification criteria and the level of the software application (as defined above).

DO-330 identifies five levels of TQL as follows. TQL-1 is the most demanding level whereas TQL-5 is the least demanding one as presented in the following table.

<table>
<thead>
<tr>
<th>Software Level</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>TQL-1</td>
</tr>
<tr>
<td>B</td>
<td>TQL-2</td>
</tr>
<tr>
<td>C</td>
<td>TQL-3</td>
</tr>
<tr>
<td>D</td>
<td>TQL-3</td>
</tr>
</tbody>
</table>

### 2.5.4 Tool Stakeholders

One major improvement of the DO-330 with regard to DO-178B is a clear separation of responsibility between the tool user and the tool developer. This is particularly relevant in the context of COTS tool qualification.

Two roles are identified in DO-330, §11.3.1:

- The **Tool developer** is in charge of developing, verifying, documenting, and producing the tool
- The **Tool user** is in charge of selecting, using, and qualifying the tool in the scope of a given software application

Both roles have different activities to consider and objectives to satisfy.

These objectives for tools are summarized in Annex A Tables. These tables are similar to the ones defined in DO-178C for software application. However, there are some differences that are listed below:

- Tables are numbered as T-x, rather than A-x to distinguish them from DO-178C
- DO-330 defines 11 Annex A Tables (instead of 10): table T-0 is a DO-330 tool specific table (there is no equivalent in DO-178C). This table includes in particular seven objectives to address the tool operation from the user’s perspective.

In the context of COTS tool qualification, DO-330, §11.3.2 and its associated Table 11-1 identify the objectives and activities typically applicable to the tool developer. Highlighting the need of providing tool qualification documents such as developer-TOR, TQP, TCI and TAS.

On the other side, DO-330, §11.3.3 and its associated Table 11-2 provide information on the typical tool user objectives and activities. From a user’s perspective, documents such as TOR, TQP, TCI, and TAS are considered.

For further information on Tool Qualification Processes and Data, please refer to DO-330.
3. Model-Based Development with SCADE

3.1 What is SCADE?

SCADE ORIGIN AND APPLICATION DOMAIN

SCADE is a product family that includes the following product lines:

- SCADE Suite for the design of embedded control applications;
- SCADE Display for the design of embedded displays graphics;
- SCADE Test for the dynamic verification of the models and the code;
- SCADE LifeCycle for the application life cycle management of these applications.

The name SCADE stands for “Safety-Critical Application Development Environment”. When spelled Scade it refers to the language on which SCADE Suite is based.
In its early academic inception, the Scade language was designed for the development of safety-critical software. It relies on the theory of languages for real-time applications and, in particular, on the Lustre and Esterel languages as described in [Lustre] and [Esterel]. The Scade language has evolved from this base and currently is a formal notation spanning a full set of features needed to model complex, hard real-time safety-critical applications.

From its early industrial stages, SCADE Suite was developed in conjunction with companies developing safety-critical software. SCADE Suite was used on an industrial basis for the development of safety-critical software, such as flight control software (Airbus, Airbus Helicopters), Full Authority Digital Engine Control, nuclear power plant safety systems (Rolls-Royce Civil Nuclear), and railway switching systems (Ansaldo STS).

SCADE addresses the applicative part of software, including graphics and associated logic, as illustrated in Figure 3.2. This is usually the most complex and changeable aspect of software. It typically represents 60 percent to 80 percent of the software embedded in an airborne computer.

For any development projects including display software, you can request a dedicated handbook from Esterel Technologies [DISPLAY-HB] dealing with the joint development of graphics with SCADE Display and the associated logic with SCADE Suite.

A BRIDGE BETWEEN CONTROL ENGINEERING AND SOFTWARE ENGINEERING

Control engineers and software engineers typically use quite different notations and concepts:

- Control engineers describe systems and their controllers using block diagrams and transfer functions (s form for continuous time, z form for discrete time), as shown below in Figure 3.3.
Software engineers describe their programs in terms of tasks, flow charts, and algorithms, as shown below in Figure 3.4.

These differences make transition from control engineering specifications to software engineering specifications complex, expensive, and error-prone.

\[ X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n} \] (bilateral z transform)

To address this problem, SCADE Suite offers rigorous software constructs that reflect control engineering constructs:

- Its data flow structure fits the block diagram approach.
- Its clocks support formal expression of sample rates.
- Its time operators fit the z operator of control engineering. For instance, \( z^{-1} \), the operator of control engineering (meaning a unit delay), has an equivalent operator called “pre” in Scade.

3.2 SCADE Modeling Techniques

3.2.1 Modeling behavior with SCADE Suite

3.2.1.1 Familiarity and accuracy reconciled

SCADE Suite uses a combination of two specification formalisms that are familiar to control engineers:

- State machines to specify modes and transitions in an application (e.g., taking off, landing, etc.)
- Data flow diagrams to specify control algorithms (control laws, filters, etc.)

The modeling techniques of SCADE Suite add a very rigorous view of these well-known but often insufficiently defined formalisms. The Scade language has a formal foundation and provides a precise definition of concurrency; it ensures that all programs generated from Scade...
models behave deterministically. SCADE Suite allows for automatic generation of C/Ada code from these two formalisms.

### 3.2.1.2 Scade operator

The basic building block in Scade is called an operator. An operator is either a pre-defined operator (e.g., +, delay) or a user-defined operator that decomposes itself using other operators. It is thus possible to build complex applications in a structured way. An operator can be represented either graphically (see Figure 3.5), or textually (see Table 3.1 below).

![Graphical notation for an integrator operator](image)

**Figure 3.5:** Graphical notation for an integrator operator

An operator is a functional module made of the following components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Textual Notation for an Integrator Operator ¹</th>
<th>Graphical Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formal interface</td>
<td>node IntegrFwd(U: real ; hidden Period: real) returns (Y: real) ;</td>
<td>Arrows and rectangles</td>
</tr>
<tr>
<td>Local variables</td>
<td>var delta : real ; last_y : real;</td>
<td>Named wires</td>
</tr>
<tr>
<td>declarations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equations</td>
<td>delta = u * Period ; y = delta + last_y ;</td>
<td>Network of operator calls</td>
</tr>
<tr>
<td></td>
<td>last_y = fby(y , 1 , 0.0) ;</td>
<td></td>
</tr>
</tbody>
</table>

¹ In Scade, an operator that needs to memorize variables from one cycle to the next (e.g., a counter) is called a node. Otherwise it is called a function.

Actually, the textual notation is the semantic reference, which is stored in files and used by all tools; the graphical representation is a projection of the textual notation, taking into account secondary layout details.

SCADE Suite Editor supports a user-friendly structured editing mode for graphical and textual operators.

An operator is fully modular:
- There is a clear distinction between its interface and its body
- There can be no side-effects from one operator to another one
- The behavior of an operator does not depend on its context of use
- An operator can be used safely in several places in the same model or in another one
3.2.1.3 Data flow diagrams for continuous control

By “continuous control”, we mean regular periodic computation such as sampling sensors at regular time intervals, performing signal-processing computations on their values, computing control laws and outputting the results. The same sequential function applies to each computation cycle.

In Scade, continuous control is graphically specified using data flow diagrams, such as the one illustrated in Figure 3.6 below.

![Diagram](image)

**Figure 3.6:** Sample of SCADE Suite model data flows from a Flight Control system

Operators compute mathematical functions, filters, and delays, while arrows denote data flowing between operator instances. Operator instances that have no functional dependency are computed concurrently. Flows may carry numeric, Boolean, enumeration, or structured values used or produced by operators.

Scade operators are fully hierarchical: operators at a description level can themselves be composed of smaller operators interconnected by local flows. In SCADE Suite Editor, one can zoom into hierarchical operators. Hierarchy makes it possible to break design complexity by a divide-and-conquer approach and to design reusable library operators.

Scade is modular: the behavior of an operator does not vary from one context to another.

The Scade language is strongly typed, in the sense that each data flow has a type, and that type consistency in Scade models is verified by the SCADE Suite tools.

SCADE Suite makes it possible to deal properly with issues of timing and causality. Causality means that if data x depends on data y, then y has to be available before the computation of x starts. A recursive data circuit poses a causality problem, as shown in Figure 3.7 below, where the “Throttle” output depends on itself via the ComputeTargetSpeed and ComputeThrottle operators. The SCADE Suite Semantic Checker detects this error and signals that this output has a recursive definition.
Figure 3.7: Detection of a causality problem

As shown in Figure 3.8, inserting an FBY (delay with initial value) operator in the feedback loop solves the causality problem, since the input of the ComputeTargetSpeed operator is now the value of “Throttle” from the previous cycle.

Figure 3.8: Functional expression of concurrency in SCADE Suite

The Scade language provides a simple and clean expression of concurrency and functional dependency at the functional level, as follows:

- Operators SetRegulationMode and ComputeTargetSpeed are functionally parallel; since they are independent, the relative computation order of these operators does not matter (because, in Scade, there are no side-effects).
- ComputeThrottle functionally depends on an output of ComputeTargetSpeed. SCADE Suite

KCG Code Generator takes this into account: it generates code that executes ComputeTargetSpeed before ComputeThrottle. The computation order is always up-to-date and correct, even when dependencies are very indirect and when the model is updated. The users do not need to spend time performing tedious and error-prone dependency analyses to determine sequencing manually. They can focus on functions rather than on coding.

Another important feature of the Scade language is related to the initialization of flows. In the absence of explicit initialization for instance by using the - > (Init) operator, the SCADE Suite Semantic Checker emits errors, as illustrated in Figure 3.9, which models a counter.

Figure 3.9: Detection of a flow initialization problem

As shown in Figure 3.10, inserting an Init operator in the feedback loop solves the initialization problem. The second argument of the + operator is 0 in step 1 (initial value), and the previous value of flow N in steps 2, 3, etc. Mastering initial values is indeed a critical subject for safety-critical software.

Figure 3.10: Initialization of flows
3.2.1.4 State Machines for discrete control

By “discrete control” we mean changing behavior according to external events originating either from discrete sensors and user inputs or from internal program events, for example, value threshold detection. Discrete control is used when behavior varies qualitatively as a response to events. This is characteristic of modal human-machine interfaces, alarm handling, complex mode handling, or communication protocols.

As a topic of very extensive studies over the last fifty years, state machines and their theory are well-known and accepted. However, in practice, they have not been adequate even for medium-size applications, since their size and complexity tend to explode very rapidly. For this reason, a richer concept of hierarchical state machines was introduced.

States can be either simple states or macro states, themselves recursively containing a full state machine. When a macro state is active, so are the state machines it contains. When a macro state is exited by taking a transition out of its boundary, the macro state is exited and all the active state machines it contains are preempted, whichever state they were in. State machines communicate by exchanging signals that may be scoped to the macro state that contains them.

The definition of state machines specifically forbids dubious constructs found in other hierarchical state machine formalisms: transitions crossing macro state boundaries, transitions that can be taken halfway and then backtracked, and so on. These are non modular, semantically ill-defined, and very hard to figure out, hence inappropriate for safety-critical designs. They are usually not recommended by methodology guidelines.

3.2.1.5 Combining data and control flows

Large applications contain cooperating continuous and discrete control parts. SCADE Suite gives developers the ability to freely and rigorously combine and nest data flows and control flows, as shown in Figure 3.11.
3.2.1.6 Scade data typing

The Scade language is strongly typed. The following data types are supported:

- Predefined types: Boolean, Integer, Real, Enumeration, Character
- Structured types:
  - Structures make it possible to group data of different types. Example:
    
    \[ T_s = \{ x: \text{int}, y: \text{real} \}; \]

- Arrays group data of a homogeneous type. They have a static size. Example:

  \[ \text{tab} = \text{real}^3; \]

- Imported types that are defined in C or Ada (to interface with legacy software)
  All variables are explicitly typed, and type consistency is verified by the SCADE Suite Semantic Checker.

3.2.2 SCADE Suite cycle-based intuitive computation model

The cycle-based execution model of SCADE Suite is a direct computer implementation of the ubiquitous sampling-actuating model of control engineering. It consists in performing a continuous loop of the form illustrated in Figure 3.12 below.
In this loop, there is a strict alternation between environment actions and program actions. Once the input sensors are read, the cyclic function starts computing the cycle outputs. During that time, the cyclic function is blind to environment changes. When the outputs are ready, or at a given time determined by a clock, the output values are fed back to the environment, and the program waits for the start of the next cycle.

State machines have the very same notion of a cycle. For a simple state machine, a cycle consists in performing the adequate transition from the current state to this cycle’s active state and compute actions in the active state. Concurrent state machines communicate with each other, receiving the signals sent by other machines and possibly sending signals back. Finally, data flow diagrams and state machines in the same design also communicate at each cycle.

This cycle-based computation model carefully distinguishes between logical concurrency and physical concurrency. The application is described in terms of logically concurrent activities, data flow diagrams or state machines. Concurrency is resolved at code generation time, and the generated code remains standard, sequential, and deterministic C code, all contained within a very simple subset of this language. What matters is that the final sequential code behaves exactly as the original concurrent specification, which can be formally guaranteed. Note that there is no overhead for communication, which is internally implemented using well-controlled shared variables without any context switching.

2. It is still possible for interrupt service routines or other task to run, as long as they do not interfere with the cyclic function.
3.2.3 SCADE Suite as a model-based development environment

SCADE Suite is an environment for the development of safety-critical aeronautics software.

- The SCADE Suite model is usually considered as a design model. It mainly represents the software low-level requirements and software architecture. The model relies on a formally defined notation.
- The model can be managed under configuration control.
- Documentation is automatically and directly generated from the model: it is correct and up-to-date by construction.
- Semantics checking can be performed in order to check that the SCADE Suite model follows the rules of the Scade notation semantics.
- The model can be exercised by simulation to verify dynamically its behavior according to upper-level requirements.
- Model coverage analysis can be performed to assess how thoroughly the model was tested and to detect unintended functions in the model.
- Formal verification techniques can be directly applied to the model to detect corner cases defects or to prove safety properties.
- Time and stack analysis can be performed in order to perform early verification of compatibility in term of execution time and memory size between the SCADE Suite model and the target platform.
- Target compatibility with SCADE Suite KCG-generated code can also be verified on a representative code sample (using CVK) in particular to anticipate some potential issues with the cross-compiler used to generate the target EOC.

- Code is automatically and directly generated from the model with the KCG qualified Code Generator: the source code complies with the semantic of the input model by construction.
- SCADE Suite-generated code can be wrapped in an RTOS task, thus implementing the needed cyclic function.
- The DO-178C Certification Kit provides all of the evidence that is needed to qualify SCADE Suite KCG as DO-330/TQL-1 (see Appendix C).

SCADE Suite applies the following “golden rules”:

- **Share unique and accurate specifications.**
- **Do things once:** Do not rewrite descriptions from one activity to another. For instance, between software architecture and software system design, between simulation and target testing, between module design and code.
- **Do things right at first shot:** Detect errors in the early stages of a project.

**Benefits of the Design-Verify-Generate Principle**

SCADE Suite allows saving the time spent on significant verification efforts because models can be verified as soon as they are available (even in parts) thus avoiding situations where code has to be developed before any verification can start and every error that is detected requires a lengthy change cycle.
SCADE Suite models formalize a significant part of the software architecture and design. The model is written and maintained once in the project and shared among team members. Expensive and error-prone rewriting is thus avoided; interpretation errors are minimized. All members of the project team, from the specification team to the review and testing teams, can share SCADE Suite models as a reference.

This formal definition can even be used as a contractual requirement document with subcontractors. Basing the activities on an identical formal definition of the software may save a lot of rework, and acceptance testing is faster using simulation scenarios.

The remainder of this handbook explains how full benefit can be obtained using SCADE Suite in a DO-178C project.

3.2.4 SCADE Suite modeling and safety benefits

In conclusion to 3.2, we have shown that SCADE Suite strongly supports safety at model level because:

• The Scade modeling language is rigorously defined. Its interpretation does not depend on readers or any tool. It relies on more than 25 years of academic research ([Esterel], [Lustre]). The semantic kernel of Scade is very stable: it has not changed over the last 25 years.

• The Scade modeling language is simple. It relies on very few basic concepts and simple combination rules of these concepts.

• Control structures remain at a high-level of abstraction. For example, array operations in Scade are expressed as such and do not require low-level loops and indexes. There is no need for goto's, no need for the creation of memory at runtime, no way to incorrectly access memory through pointers or an index out of bounds in an array. Moreover, these principles are reflected in the generated code out of SCADE Suite KCG.

• The Scade modeling language contains specific features oriented towards safety: strong typing, mandatory initialization of flows, and so on.

• A Scade model is deterministic. A system is deterministic if it always reacts in the same way to the same inputs occurring with the same timing. In contrast, a non-deterministic system can react in different ways to the same inputs, the actual reaction depending on internal choices or computation timings.

• The Scade modeling language provides a simple and clean expression of concurrency at functional level (data flows express dependencies between operators). Within a Scade model, this avoids the traditional problems of deadlocks and race conditions.

• SCADE Suite performs the complete verification of language rules, such as type and clock consistency, initialization of data flows, or causality in Scade models.
4. Software Development Activities with SCADE Suite

This section provides a more detailed view of the development activities that were introduced in the previous section.

4.1 Overview of Software Development Activities

A typical SCADE Suite software development process is a combination of SCADE Suite Model-Based development flow with a traditional manual coding flow. Industrial projects show that the fraction developed with SCADE Suite typically ranges from 60 percent to 90 percent of the applicative part of the software.

Figure 4.1 shows the software development processes and differentiates in shadowed boxes those where SCADE Suite is used.

Some companies start using SCADE Suite to define control laws during the system definition.
The simulation of SCADE Suite models can be used very early in the software development life cycle to refine, improve, and validate the textual high-level requirements that are an input of the software design process (see Section 4.2).

SCADE Suite modeling is used extensively in the software design process to develop major parts of the architecture and the low-level requirements. The SCADE Suite model is a design model according to DO-331 definition (see Section 2.4). The corresponding source code is then generated from such model by using SCADE Suite KCG.

Traceability within the software development process (as defined in § 5.5 [DO-178C]) requires bi-directional traceability between:

- System requirements allocated to software and HLRs
- HLRs and LLRs (i.e., SCADE Suite model)
- LLRs and source code

The traceability process between HLRs and the SCADE Suite model can be easily supported by SCADE LifeCycle Requirements Management Gateway as mentioned in Figure 4.1 above and as illustrated in Section 4.3.

### 4.2 Software Requirements Process

The main goal of the software requirements process is to develop the high-level requirements (HLRs). These HLRs are usually in textual form and include functional, performance, interface, and safety-related requirements.

SCADE Suite modeling capabilities can be efficiently used to refine, improve, and validate the functional part of textual HLRs defined as input of the software design process.

In this context, a prototype may be developed in SCADE Suite for all or parts of the functional HLRs with a focus on complex dynamic algorithms. Such prototype can be simulated using SCADE Test Rapid Prototyper (see [TEST-UM] for more information on prototyping and simulation capabilities).

The Scade formal notation and interactive simulation capabilities are a good support:

- To improve quality and productivity in the development of textual software requirements
- To speed up safety impact analysis if requirements change
4.3 Software Design Process with SCADE Suite

As explained in [DO-178C] §5.2, “the high-level requirements are refined through one or more iterations in the software design process to develop the software architecture and the low-level requirements”.

Figure 4.2 illustrates the design flow with SCADE Suite that is detailed in next sections.

4.3.1 Architecture design

GLOBAL ARCHITECTURE DESIGN
The first step in the design process is to define the global application architecture, taking into account both Scade and manual software elements.

The application is decomposed functionally into its main components. The characteristics of these components serve as a basis for allocating their refinement in terms of techniques (Scade, C, ...) and team. Among those characteristics, one has to consider, for a software component:

- The type of processing (e.g., filtering, decision logic, byte encoding)
- The interaction it has with hardware or the operating system (e.g., direct memory access, interrupt handling)
- Activation conditions (e.g., initialization) and frequency (e.g., 100 Hz)

SCADE Suite is well-adapted to the functional parts of the software, such as logic, filtering, regulation. It may be less appropriate for low-level software such as hardware drivers, interrupt handlers, and encoding/decoding routines.

SCADE ARCHITECTURE DESIGN
An architecture design model as shown in next figure can be developed at this stage to:

- Identify the high-level functions: one would typically develop a functional breakdown down to a depth of two or three
- Define the interfaces of these functions; names, data types (see I/O handling)
- Describe the data flows and control flows between these functions
- Verify consistency of the data flows between these functions using the SCADE Suite Semantic Checker
- Prepare the framework for the detailed design process: define the top-level functions while ensuring consistency of their interfaces
This architecture design model is extremely important because it lays the foundations for the detailed design.

In particular, a good architecture aims at ensuring:

- **Stability and maintainability**: The team needs a stable framework during the initial development as well as when there are updates.
- **Readability and verifiability**: Readability comes naturally through the clear and unambiguous Scade language semantics, and simple and intuitive graphical symbology. Verifiability comes naturally with a formal notation such as Scade, but also requires to minimize the complexity of the model.
- **Efficiency**: There is no magic recipe to achieving a good Scade model architecture, rather it requires a mix of experience, creativity, and rigor. Here are a few suggestions:
  - Be reasonable and realistic: nobody can build a good architecture in one shot. Do not develop the full model from your first draft, but build two or three architecture variants, then analyze and compare them; otherwise, you may have to live with a bad architecture for a long time.
  - Review and discuss the architecture with peers.
  - Generate code and test on target the impact in terms of robustness and performances of some changes that are likely to occur (e.g., new interface and/or data structuring).
  - Select the architecture that minimizes complexity of connections and is robust to changes.

For example, the architecture, shown in Figure 4.3, groups several logical controls in one structured top-level operator. Such design is
more maintainable than if each individual control would have its own function with duplicated interfaces in the model.

**INPUT/OUTPUT HANDLING**

Raw acquisition from physical devices and/or from data buses are usually implemented with specific drivers externally to the SCADE Suite model (or through imported code) and with a manual coding approach. Inputs/Outputs at the frontier of a SCADE Suite model or within a SCADE Suite model are generally normalized and grouped according to a given functional meaning.

4.3.2 SCADE low-level requirements development

Once the SCADE Suite architecture is defined, every SCADE Suite component is refined to design the low-level requirements (LLRs) with SCADE Suite. The objective of this activity is to produce a complete and consistent SCADE Suite design model.

The definition and granularity of an LLR in a SCADE Suite model is determined and customized by the user itself.

A commonly used definition for LLR is to consider data flow diagrams (for block diagrams) and states and transitions (for state machines). Some diagrams can be categorized as architecture diagram contrary to LLR diagrams. These diagrams express decomposition and data flow connections only and contain no decision or processing.

Distinction between LLR diagram and architecture diagram is performed by the user during the design phase with the support of SCADE Suite annotation mechanism.

For further information on LLR and architecture definition within a SCADE Suite model, please refer to [SC-SDVST].

The Scade modeling language includes both a graphical and a textual representation. It supports a unified modeling style that enables the design of complex algorithms and the design of complex control software. Both styles can be combined without restriction while the modularity of the design is continuously supported.

This language efficiently supports good practices for the development of high-integrity software such as:

- **Encapsulation** (modularity)
- **Strong typing**
- **Concurrency**
- **Re-usable components** (interface definition, genericity, library)

The following sections provide some examples of Scade modeling patterns that illustrate the above concepts.

**FILTERING AND REGULATION**

Filtering and regulation algorithms are usually designed by control engineers. Their design is often formalized in the form of block diagrams and transfer functions defined in terms of “$z$” expressions.
The Scade graphical notation allows representing block diagrams exactly the same way as control engineers, using the same semantics. The Scade time operators fit the \( z \) operator of control engineering. For instance, the \( z^{-1} \) operator of control engineering (meaning a unit delay) has equivalent operators called “pre” and “fby” in Scade. For example, if a control engineer has written an equation such as \( s = K_1u - K_2z^{-1}s \), which means \( s(k) = K_1u(k) - K_2s(k-1) \), this can be expressed in textual Scade as \( s = K_1u - K_2\text{pre}(s) \) or graphically, as shown in Figure 4.4 below.

**Figure 4.4:** A first order filter

Scade can implement both Infinite Impulse Response (IIR) and Finite Impulse Response (FIR) filters. In a FIR filter, the output depends on a finite number of past input values; in an IIR filter such as the one above, the output depends on an infinite number of past input values because there is a loop in the diagram.

**DECISION LOGIC**

In modern controllers, logic is often more complex than filtering and regulation. The controller has to handle:

- Identification of the situation
- Detection of abnormal conditions
- Decision making
- Management of redundant computation chains

SCADE Suite offers a variety of techniques for handling logic:

- Logical operators (such as and/ or/ xor) and comparators.
- Selecting flows, based on conditions, with the “if” and “case” constructs.
- Building complex functions from simpler ones.

Scade supports encapsulation and modularity with the concept of user-defined operators. For instance, the UnitConvert is built from basic counting, comparison, and logical operators; it can in turn be used in more complex functions to make them simpler and more readable, as in Figure 4.5.

**Figure 4.5:** Complex display logic and simple functions

- Conditional activation of operators depending on Boolean conditions.
- State Machines as in Figure 4.6. For instance, both modes of a Flight Controller System can be easily designed with a state machine including two states: one state for the Manual mode, one state for the AutoPilot mode.
WHICH TECHNIQUE FOR DECISION LOGIC?
When starting with SCADE Suite, one may ask which of the above-mentioned techniques to select for describing logic. Here are some hints for the selection of the appropriate technique:

- **Selecting state machines or logical expressions**:
  - Does the output depend on the past? If it only depends on current inputs, this is just combinatorial logic: simply use a logical expression in the data flow. A state machine that jumps to state Xi when condition Ci is true independently from current state, is and does not need to be a state machine.
  - Does the state have strong qualitative influence on behavior? This favors a state machine.
- **Expressing concurrency**:
  - Simply design parallel data flows and state machines: this is natural and readable, and the code generator is in charge of implementing this parallel specification into sequential code.

4.3.3 Re-usable components and library management

Some general-purpose components (e.g., matrix product, integrator, rising edge detector) should not be redone and maintained multiple times, but should rather be shared among projects in a library. Some libraries may also be managed for sharing components at the application level (special type of filter). Development and verification artifacts are managed in shared libraries. Using library operators has advantages:

- It saves time;
- It relies on validated components;
- It makes models more readable and maintainable.
  
  For instance, a call to an Integrator is much more readable than the set of lower-level operators and connections that implement an Integrator;
- It enforces consistency throughout the project;
- It factors the code.

SCADE LANGUAGE ADVANCED CONCEPTS FOR REUSABILITY

The Scade language supports several concepts that facilitate the development of re-usable components. It includes:

- Library
- Genericity/Polymorphism
- Parameterization by size

Figure 4.7 shows a predefined SCADE Suite library (libmath.etp as mathematical library can be re-used for application design). Users can create their own library and reference them in the upper-level application (e.g., libPlane library in FlightControl project).
A library may include generic operators (called polymorphic operators). Such operators are defined independently from the type of their arguments and can be instantiated with various types. The figure below illustrates a GenericToggle operator instantiated once with integer and another time with Boolean.

For algorithms on arrays (iterative scheme), the size of input/output arrays for an operator can be parameterized. The size identifier is part of the formal interface of this operator.

Figure 4.9 shows an operator (MaxParametric) that computes the maximum value of a set of integer values implemented as array. It is parameterized by size and can be instantiated with a static value (literal 5 in this example).
SCADE SUITE LIBRARY SOFTWARE LIFECYCLE

A SCADE Suite library must be developed as any other SCADE Suite component, taking into account the following considerations:

- Library components are usually identified during the design process of a given application and can be considered in most cases as implementation choices, not necessarily described in the upper-level requirements (HLRs) of the application.
- Good practices consist in defining functional requirements (derived HLRs) for these library components as a separate document and in developing and verifying the components from its derived HLRs.
- When a library is shared between several applications, a self-contained development package may be considered, including its own project plans and standards, requirements, design data, verification reports, Software Quality Assurance reports and Software Configuration Management reports.

4.3.4 Robustness management

Robustness of a safety critical software cannot be addressed locally. It requires a general robustness policy for the whole system and should be addressed at each step of the development and verification processes.

The robustness policy should be defined in the Software Design Standards, Software Coding Standards, and Software Architecture Design Document. As an example, the way for handling arithmetic exceptions should be defined at this global level.

There should be explicit decisions about robustness and failure handling in the software requirements.

The HLRs (including derived HLRs for library components) should specify responses to abnormal input data and to any invalid data that may be produced by computation described in the HLR (e.g., for X=Y/Z, the HLR should specify the expected behavior to Z near zero, except if there is evidence that Z is far from zero, or more precisely that Y/Z cannot generate a division by zero). This is required to achieve accuracy and determinism of requirements and to perform requirements-based testing for robustness tests.

COMMUNICATION WITH EXTERNAL ENVIRONMENT

A golden design rule is to never trust an external input without appropriate verification and to build consolidated data from the appropriate combination of available data.

By using SCADE Suite component libraries, one can, for instance, insert:

- A voting function
- A low pass filter and/or limiter for a numeric value
- A Confirmator for Boolean values, as shown in Figure 4.10

Figure 4.10: Inserting Confirmator in Boolean input flow
In a similar way, outputs to actuators have to be value-limited and rate-limited, which can be ensured by inserting Limiter operators before the output, as shown in Figure 4.11 below.

![Figure 4.11: Inserting Limiter in output flow](image)

Since the data flow is very explicit in a SCAD E Suite model, it is both easy to insert these components in the data flow and to verify their presence when reviewing a model.

**DEFENSIVE PROGRAMMING**

Defensive programming is a well-known technique to make a design robust. It means the following:

- Normal and abnormal input domains are identified
- The SCAD E Suite operator is designed/ coded as such that it reacts in a safe way to abnormal inputs
- It is not critical for the environment of this function to care about normal conditions

For example, such a defensive programming strategy for a square root operator amounts to implementing a specific behavior (according to the upper-level requirements) when the input is negative.

This approach is systematic and the direct benefit is robustness. The potential drawback is run-time cost, even in cases when there is evidence that the normal conditions hold, for example square root of \((x**2+y**2)\).

Another alternative to optimize run-time efficiency is to consider a contractual programming approach as presented below.

**CONTRACTUAL PROGRAMMING**

This approach allows for alleviating the design from the overhead of some defensive constructs when given preconditions are fulfilled on a given operator. For instance, the precondition for a non robust square root function is that the input is non-negative. In this context, this is the responsibility of the SCAD E Suite operator calling the square root function to ensure that this precondition is fulfilled.

This approach is efficient for performance purposes but the drawback is vulnerability: extreme care must be taken when verifying design with contractual programming.

### 4.4 Software Coding Process

The SCAD E Suite KCG code generator automatically generates the complete code that implements the software design defined in Scade for both data flows and state machines (see Figure 4.12). It is not just a generation of skeletons, the complete dynamic behavior is implemented.
4.4.1 Code generation from SCADE Suite models

The SCADE Suite model completely defines the expected behavior of the generated code. The code generation options define the implementation choices for the software. However, these options never complement nor alter the behavior of the model.

**PROPERTIES OF THE GENERATED CODE**

Independently from the choice of the code generation options, the generated code has the following properties:

- The code is portable: it is [ISO-C] compliant.
- The code structure reflects the model architecture for data-flow parts when there is no expansion and/or optimization during code generation. For control-flow parts, traceability between state names and C code is ensured.
- The code is readable and traceable to the input Scade model through the use of corresponding names, specific comments, and traceability file.
- Memory allocation is fully static (no dynamic memory allocation).
- There is no recursive call.
- Only bounded loops are allowed, since they use static values known at SCADE Suite KCG code generation time.
- Execution time is bounded.
- Expressions are explicitly parenthesized.
- No dynamic address calculation is performed (no pointer arithmetic).
- There are no implicit conversions.
- There is no expression with side-effects (no i++, no a += b, no side-effect in function calls).
- No functions are passed as arguments.

Traceability of the code to a Scade data flow model is illustrated in Figure 4.13.
Traceability of the code to the SCADE State Machine model is illustrated in Figure 4.14.
To further support automated analysis, a traceability file is generated by SCADE Suite KCG as shown in Figure 4.15.

```
<NoExpNode scadename="AbContrib" headerFile="AbContrib_FlightControl.im" targetCycleFct="AbContrib_FlightControl">
  <Input scadename="allTarget" scadetype="real" targetName="allTarget" targetType="Kcg_real"/>
  <Input scadename="allSensor" scadetype="real" targetName="allSensor" targetType="Kcg_real"/>
  <Output scadename="allContrib" scadetype="real" targetName="allContrib" targetType="Kcg_real"/>
  <Local scadename="L14" scadetype="real" targetName="allContrib" targetType="Kcg_real"/>
  <Local scadename="L16" scadetype="real" targetName="allTarget" targetType="Kcg_real"/>
  <Local scadename="L17" scadetype="real" targetName="allSensor" targetType="Kcg_real"/>
  <NodeInlining scadename="func.Gen" instName="1"/>
</NoExpNode>
```

**Figure 4.15**: Sample of kcg_trace.xml for traceability between Scade model and source code

**TUNING CODE TO TARGET AND PROJECT CONSTRAINTS**

Various code generation options can be used to tune the generated code to a particular target and project constraints. Static analysis methods are available in SCADE Suite. Specified as a Scade model, the applicative software can be analyzed from the execution time point of view allowing to tune modeling choices and code generation options according to users’ needs. Basically, there are two ways to generate code from a Scade operator:

- **Non-expanded mode**: the operator is generated as a C function.
- **Expanded mode**: the whole code for the operator is inlined where it is called.

This is illustrated in Figure 4.16.

**Figure 4.16**: Non-expanded and Expanded modes

Both of these code generation modes (Non-expanded or Expanded) can be composed at will, performing a call for some operators and inlining for other operators.

Note that the expansion directives (see Non-expanded mode and Expanded mode above) and some interface directives (see definition below about global_root_context option and separate_io option/pragmas) may have an impact on the structure of the generated code, on the integration of the SCADE Suite KCG-generated code, and even on the verification strategy.
These options and directives can be considered as a design choice and should be identified very early in the software development life cycle, preferably during the SCADE Suite architecture decomposition:

- The `global_root_context` SCADE Suite KCG option is a code generation mode where the inputs, outputs and context variables of the root operators are defined as C global variables and not passed as arguments of the root C functions. This change on the signature of root C functions impacts the integration of KCG generated code.
- The `separate_io` SCADE Suite KCG option and/or pragma applies to a SCADE Suite operator. When it is set, the code generated for the cycle function is different: outputs are no more in the context but passed as separate parameters. As for the global root context, it impacts the integration of SCADE Suite KCG generated code.

4.4.2 Code generation from multiple components

The SCADE Suite KCG code generator is specified and designed for verifying a complete application and generating the corresponding complete set of C files in one global run, in order to ensure maximal consistency of the generated code.

This process is usually sufficient because it ensures the global consistency of the code generated from a single SCADE Suite component. Yet, it may not be appropriate in the context of complex software architecture. A complex SCADE Suite application can result from several SCADE Suite components (interacting or not together) where each component corresponds to a single SCADE Suite library model with a given root node. It is the case for instance, when the SCADE Suite application includes several tasks and each task is designed with a separate SCADE Suite model.

As shown in Figure 4.17, there are two alternatives for generating code with SCADE Suite KCG:

1. Generating all code in one run, using the "multi-root operators" SCADE Suite KCG option (see SCADE Suite User Manual for further information on options). This applies whether both root operators are defined in the same model or not. When both operators do not belong to the same model, a new integration model, which references the input models as libraries, is created (see integration model in Figure 4.17).

2. Generating code for each root node separately and then integrating both generated codes into the application.

Note that the coding process described in the first alternative is highly recommended unless there is a major reason for not using it. It is the safest and cleanest way to integrate the different root nodes. It is also highly recommended as a means for performing verification and validation of the global behavior.

Even if the use of some SCADE Suite KCG directives such as manifest pragma and/or global prefix option (see below) may support the application of the second alternative, it requires a strict coding and integration process with additional verification activities to check the consistency of the interfaces and of the
integration. For further information on modular code generation process, please contact Esterel Technologies.

Figure 4.17: Code generation and multiple components

- The manifest pragma is used to control the type names generated by SCADE Suite KCG. It ensures better stability of the code between two code generation sessions.
- The global prefix SCADE Suite KCG option is used to prevent name conflict during integration of generated code. It adds a prefix (user-specified) in front of the names of C global identifiers.

4.5 Software Integration Process

4.5.1 Integration aspects

Integration of SCADE Suite-generated code is about:

- Interface with the external environment
- SCADE Suite module integration
- Integration of external data and code
- Scheduling and tasking

4.5.2 Interface with the external environment

Interface to physical sensors and/or to data buses is usually handled by drivers. If data acquisition is done sequentially, while the Scade function is not active, then a driver may pass its data directly to the Scade input. If it is complex data, it may be passed by address for efficiency reasons. If a driver is interrupt-driven, then it is necessary to ensure that the inputs of the Scade function remain stable, while the Scade function is computing the current cycle. This can be ensured by separating the internal buffer of the driver from the Scade input vector and by performing a transfer (or address swap) before each SCADE Suite computation cycle starts. These drivers are usually not developed in Scade but in C or assembly language.
4.5.3 SCADE module integration

A module refers here to the C code generated by SCADE Suite KCG from a SCADE Suite component. Depending on the selected code generation process (see Figure 4.17 in Section 4.4.2), the user has to manage the integration of one or several modules with the rest of the software application.

The SCADE Suite KCG directives for tuning the generated code (such as options and pragmas defined in Section 4.4.1) shall be considered by the user as early as possible while integrating the generated code.

Moreover, module integration depends on the implementation of basic SCADE Suite types (int, real, bool, char) which must be mapped to C types. A default type definition is given in the generated code but it is possible to redefine these default types by providing the implementation of each basic type (the same definition as this used for external code, see Section 4.5.3) in a user configuration file.

4.5.4 Integration of external data and code

SCADE Suite allows for using external data types and functions. In the model, external data types have to be declared as “imported,” and for functions, their interface also has to be declared. Examples of such functions are trigonometric functions or byte encoding and checksum functions. At integration time, these functions have to be compiled and linked to the SCADE-generated code. SCADE Suite Simulator automatically compiles and links external code when the path names of the source files are given in the project settings.

4.5.5 SCADE Suite scheduling and tasking

Scheduling has to be addressed in the preliminary design phase, but for the sake of simplicity it is described below. First, the section recalls the execution semantics of Scade, and then examines how to model and implement scheduling of a Scade model in single or multirate mode, while in single tasking or multitasking mode.

SCADE SUITE EXECUTION SEMANTICS

The execution semantics is based on the cycle-based execution model as described in Section 3.2.2. This model can be represented with Figure 4.16.

SCADE SUITE EXECUTION SEMANTICS

The software application samples the inputs from the environment and sets them as inputs for the Scade code. The main Scade function of the generated code is called. When Scade code execution ends, the calculated outputs can be used to act upon the environment. The software application is ready to start another cycle.
BARE SYSTEM IMPLEMENTATION

Typically, a cycle can be started in three different ways:

- **Polling**: a new cycle is started immediately after the end of the previous one in an infinite loop.

- **Event triggered**: a new cycle is started when a new start event occurs.

- **Time triggered**: a new cycle is started regularly, based on a clock signal.

The SCADE Suite-generated code can be simply included in an infinite loop, waiting or not for an event or a clock signal to start a new cycle:

```plaintext
begin_loop
  waiting for an event (usually a clock signal)
  setting Scade inputs
  calling the SCADE-generated main function
  using Scade outputs
end_loop
```

SINGLE-TASK INTEGRATION OF SCADE SUITE FUNCTION WITH AN RTOS

A SCADE Suite design can be easily integrated in an RTOS task in the same way that it is integrated in a general-purpose code, as shown in Figure 4.19. The infinite loop construct is replaced by a task. This task is activated by the start event of the SCADE Suite design, which can be a periodic alarm or a user activation.

This architecture can be designed by hand for any RTOS. SCADE Suite currently provides automation of this code production through the necessary Adaptors for “VxWorks® 653” from Wind River®, for “Integrity®-178B” from Green Hills® Software, for PikeOS from SYSGO, and for many platforms at major suppliers and integrators.

Note that concurrency is expressed functionally in the model and that the Code Generator takes into account the model structure to generate sequential code, taking into account this functional concurrency and the data flow dependencies. There is no need for the user to spend time sequencing parallel flows, neither during modeling nor during implementation.
There is no need to develop multiple tasks with complex and error-prone synchronization mechanisms. Note that other code, such as hardware drivers, may run in separate tasks, provided they do not interfere with the SCADE-generated code.

**MULTIRATE, SINGLE-TASK APPLICATIONS**

SCADE Suite can be used to design multirate applications in a single OS task. Some parts of the SCADE Suite design can be executed at a slower rate than the SCADE Suite top-level loop. Putting a slow part inside an `activate` operator can do this. Slowest rates are derived from the fastest rate, which is always the top-level rate. This ensures a deterministic behavior.

The following application has two rates: Sys1 (as fast as the top-level) and Sys2 (four times slower), as shown in [Figure 4.20](#).

The schedule of this application is as shown in [Figure 4.21](#) below:

![Timing diagram of a bi-rate system](#)

Sys2 is executed every four times only. It is executed within the same main top-level function as Sys1. This means that the whole application, Sys1 + Sys2, is executed at the fastest rate. This implies the use of a processor fast enough to execute the entire application at a fast rate. This could be a costly issue.

The solution consists in splitting the slow part into several smaller slow parts and distributing their execution on several fast rates. This is a simple way to design a multirate application. Scheduling of this application is fully deterministic and can be statically defined.

The previous application example can be redesigned as shown in [Figure 4.22](#):
SCADE Suite with DO-178C Objectives

Figure 4.22: Modeling slow system over four cycles

The slow part, Sys2, is split into four subsystems. These subsystems are executed sequentially, one after the other, in four cycles, as shown in Figure 4.23 below:

Figure 4.23: Timing diagram of distributed computations

Note Sys1 execution time can be longer than with the previous design. Thus, a slower, but less expensive, processor can be used.

The multirate aspect of a SCADE Suite design is achieved using standard Scade constructs. This has no effect on the external interface of the SCADE Suite-generated code. This code can be integrated following the infinite loop construct as described earlier.

Such design has advantages but also constraints:

- **Advantages:**
  - Static scheduling: fully deterministic, no time slot exceeded or crushed, no OS deadlock
  - Data exchanges between subsystems handled by SCADE Suite wrt. dataflow execution order
  - SCADE Suite simulation and proof are valid for the generated code
  - Same code interface as a monorate application

- **Constraints:**
  - Need to know the WCET (Worst Case Execution Time) of each subsystem to validate scheduling in all cases
  - Split of slow subsystems can be difficult with high rate ratio (e.g., 5ms and 500ms)
  - Constraint for design evolutions and maintenance

**MULTITASKING IMPLEMENTATION**

The single tasking scheme described above was used for fairly large industrial systems. There are situations where implementation of the SCADE Suite-generated code on several tasks is useful, for instance, if there is a large ratio between slow and fast execution rates.

It is possible to build a global Scade model, which formalizes the global behavior of the application, while implementing the code on different tasks. While it is also possible to build and implement separate independent models, this global model allows representative simulation and formal verification of the complete system. The distribution over several tasks requires specific analysis and implementation (see [Camus] and [Caspi] for details).
4.6 Teamwork

Working efficiently on a large project requires both distribution of the work and consistent integration of the pieces developed by each team. The Scade language is modular: there is a clear distinction between the interfaces and the contents of operators and there are no side effects from one operator to another.

A typical project organization as in Figure 4.24:

- A software architect defines the top-level operators, their interfaces, and connections.
- Utility libraries are developed.
- Each major subfunction, corresponding to a top-level operator, is developed by a specific team; the interfaces of these top-level operators define a framework for these teams, which maintain consistency of the design.

At each step, the team can verify in a mouse click that the subsystem remains consistent with its interface. Later, the integration of those parts into a larger model can be achieved by linking these "projects" to the larger one. At any stage, the SCADE Suite Semantic Checker verifies the consistency of this integration in a mouse click. All these data have to be kept under strict version and configuration management control by using any commercial Configuration Management System. Reuse is also an important means of improving productivity and consistency in a project or a series of projects. SCADE Suite libraries can store definitions of operators and/or data types for reuse in several places. These range from basic operators such as latches or integrators to complex, customer-specific systems (see Section 4.3.3).
5. Software Verification Activities

5.1 Overview

According to DO-178C, validation is “the process of determining that the requirements are the correct requirements and that they are complete.” Verification is “the evaluation of the results of a process to ensure correctness and consistency with respect to the inputs and standards provided to that process.”

In other terms, the difference lies in the nature of the errors that are found. Validation always concerns the requirements, even when a requirement error is found by testing an implementation that conforms to its (bad) requirement(s); this differs from an implementation error, which occurs when the implementation does not conform to the requirements.

The software verification process is an assessment of the results of both the software development process and the software verification process. It is satisfied through a combination of reviews, analyses, and tests.

The software testing process is a part of the verification process; it is aimed at demonstrating that the software satisfies its requirements both in normal operation and in the presence of errors that could lead to unacceptable failure conditions.

5.2 Verification of High-Level Requirements

5.2.1 Verification objectives for HLR

Table 5.1 lists verification objectives for software high-level requirements.

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Activity Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Software high-level requirements comply with system requirements</td>
<td>6.3.1</td>
</tr>
<tr>
<td>2. High-level requirements are accurate and consistent</td>
<td>6.3.1</td>
</tr>
<tr>
<td>3. High-level requirements are compatible with target computer</td>
<td>6.3.1</td>
</tr>
<tr>
<td>4. High-level requirements are verifiable</td>
<td>6.3.1</td>
</tr>
<tr>
<td>5. High-level requirements conform to standards</td>
<td>6.3.1</td>
</tr>
<tr>
<td>6. High-level requirements are traceable to system requirements</td>
<td>6.3.1</td>
</tr>
<tr>
<td>7. Algorithms are accurate</td>
<td>6.3.1</td>
</tr>
</tbody>
</table>

In a typical SCADE Suite development process, the high-level requirements are usually in textual form and include functional, performance, interface and safety-related requirements (for further information on software requirements process, see Section 4.2). They must be verified against the objectives of DO-178C Table A-3.
When the requirements from which a SCADE Suite model is developed are an output of the system process (for instance system requirements allocated to software), the guidance related to high-level requirements should be applied to these requirements according to DO-331, MB.1.6.3- Note 1 and the verification methods presented below still apply.

5.2.2 Verification methods for HLR

Due to the textual form of the requirements, this compliance is mainly addressed in a traditional way by peer review.

5.2.3 Verification summary for HLR

Table 5.2 summarizes verification objectives and methods for software high-level requirements.

Table 5.2: DO-178C Table A-3 Objectives

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Activity Ref</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Software high-level requirements comply with system requirements</td>
<td>6.3.1</td>
<td>Peer review</td>
</tr>
<tr>
<td>2 High-level requirements are accurate and consistent</td>
<td>6.3.1</td>
<td>Peer review</td>
</tr>
<tr>
<td>3 High-level requirements are compatible with target computer</td>
<td>6.3.1</td>
<td>Peer review</td>
</tr>
<tr>
<td>4 High-level requirements are verifiable</td>
<td>6.3.1</td>
<td>Peer review</td>
</tr>
<tr>
<td>5 High-level requirements conform to standards</td>
<td>6.3.1</td>
<td>Peer review</td>
</tr>
<tr>
<td>6 High-level requirements are traceable to system requirements</td>
<td>6.3.1</td>
<td>Peer review</td>
</tr>
<tr>
<td>7 Algorithms are accurate</td>
<td>6.3.1</td>
<td>Peer review</td>
</tr>
</tbody>
</table>
5.3 Verification of the SCADE Low-Level Requirements and Architecture

5.3.1 Verification objectives for the LLR and architecture

The SCADE Suite design model (see Section 4.3) has to be verified against the objectives of DO-331 Table MB.A-4 (see Table 5.3).

Note that for LLR that are not developed in SCADE Suite, verification activities have to be performed in the traditional way against the objectives of DO-178C Table A-4.

Table 5.3: DO-331 Table MB.A-4

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Activity Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Low-level requirements comply with high-level requirements</td>
<td>MB.6.3.2 MB.6.7 MB.6.8.1 (see Item 1)</td>
</tr>
<tr>
<td>2 Low-level requirements are accurate and consistent</td>
<td>MB.6.3.2 MB.6.8.1 (see Item 1)</td>
</tr>
<tr>
<td>3 Low-level requirements are compatible with target computer</td>
<td>MB.6.3.2</td>
</tr>
<tr>
<td>4 Low-level requirements are verifiable</td>
<td>MB.6.3.2 MB.6.8.1 (see Item 1)</td>
</tr>
<tr>
<td>5 Low-level requirements conform to standards</td>
<td></td>
</tr>
<tr>
<td>6 Low-level requirements are traceable to high-level requirements</td>
<td>MB.6.3.2</td>
</tr>
<tr>
<td>7 Algorithms are accurate</td>
<td>MB.6.3.2 MB.6.8.1 (see Item 1)</td>
</tr>
<tr>
<td>8 Software architecture is compatible with high-level requirements</td>
<td>MB.6.3.3 MB.6.8.1 (see Item 1)</td>
</tr>
<tr>
<td>9 Software architecture is consistent</td>
<td>MB.6.3.3 MB.6.8.1 (see Item 1)</td>
</tr>
<tr>
<td>10 Software architecture is compatible with target computer</td>
<td>MB.6.3.3</td>
</tr>
<tr>
<td>11 Software architecture is verifiable</td>
<td>MB.6.3.3 MB.6.8.1 (see Item 1)</td>
</tr>
<tr>
<td>12 Software architecture conforms to standards</td>
<td>MB.6.3.3</td>
</tr>
<tr>
<td>13 Software partitioning integrity is confirmed</td>
<td>MB.6.3.3</td>
</tr>
<tr>
<td>14 Simulation cases are correct (see Item 1)</td>
<td>MB.6.8.1 MB.6.8.3.2</td>
</tr>
<tr>
<td>15 Simulation procedures are correct (see Item 1)</td>
<td>MB.6.8.1 MB.6.8.3.2</td>
</tr>
<tr>
<td>16 Simulation results are correct and discrepancies explained (see Item 1)</td>
<td>MB.6.8.1 MB.6.8.3.2</td>
</tr>
</tbody>
</table>

Table 5.3: DO-331 Table MB.A-4 (Continued)

Item 1: As described in section MB. 6.8.1 of this supplement [DO-331], simulation may be used as a means of compliance for objectives 1, 2, 4, 7, 9, or 11 of this table. If simulation is used as this means, objectives MB.14, MB.15, and MB.16 are required.
5.3.2 Compliance with high-level requirements

The compliance with HLR is verified through a combination of techniques:
- Model simulation
- Peer review
- Formal verification

**MODEL SIMULATION WITH SCADE TEST ENVIRONMENT**

Model simulation allows exercising the behavior of a model. The main purpose is to provide repeatable evidence of compliance of the model to the requirements from which the model was developed (as stated in [DO-331], MB.6.8.1). Moreover, model simulation is an efficient way to detect functional issues very early in the software design and/or upper-level requirements.

Simulation of SCADE Suite design models requires the following activities:
- SCADE Suite verification cases and procedures are developed from the requirements from which the SCADE Suite model was developed (HLRs)
- SCADE Suite verification cases and procedures must address the same considerations as those for normal range and robustness test cases and procedures and possible error sources (see [DO-178C] 6.4.2)
- HLRs are covered by SCADE Suite verification cases and procedures
- SCADE Suite verification cases and procedures are reviewed to confirm that they are correct (see objectives MB.14 and MB.15)
- SCADE Suite models are exercised by HLRs-based SCADE Suite verification cases and procedures in the host environment
- SCADE simulation results are reviewed to confirm that they are complete and correct and all deficiencies are explained (see objective MB.16) of Table MBA-A-4

“SCADE Suite verification cases and procedures” is a generic term to refer to both:
- Simulation cases and procedures used for SCADE Suite model simulation on host during design verification ([DO-331] Table MBA-A-4);
- Test cases and procedures used for Executable Object Code (EOC) testing on target ([DO-331] Table MBA-A-6).

SCADE Test Environment combined with SCADE LifeCycle Requirements Management Gateway fully supports the following activities:

1. **SCADE Test Environment for Host** provides an integrated environment that allows validation and verification engineers to both automate the creation and management of verification cases (see Figure 5.1) and then to run on host the verification cases created from the HLRs (see Figure 5.2).
With SCADE Test, conformance report generation is automated, enabling significant time and cost savings over manual verification. SCADE Test Environment for Host is qualified as a verification tool for DO-178C/DO-330 at TQL-5. This qualification evidence allows applicants to claim credit from SCADE Test Environment for Host simulation for the verification of the compliance of a SCADE Suite model with its HLRs.

**Figure 5.1:** Verification cases creation and management in Test Environment for Host

**Figure 5.2:** Simulation results from running verification cases on host
SCADE Test Model Coverage is a coverage analysis tool that executes and reports coverage from HLR-based SCADE Suite verification cases and procedures both at model level and at generated C code level. Only model coverage analysis (see Figure 5.3) is required during SCADE Suite design verification with the objectives to assess the completeness. SCADE Test Model Coverage is qualified as a verification tool for DO-178C/DO-330 at TQL-5. This qualification evidence allows applicants to claim credit from model coverage measurement.

Figure 5.3: Model coverage analysis with SCADE Test Model Coverage

For further information about Test Model Coverage concepts and usage, refer to Chapter 6.
SCADE LifeCycle Requirements Management Gateway (RM Gateway) provides a complete and interactive solution to manage traceability links between the HLRs and the verification cases and procedures (see Figure 5.4). The objective is to verify that 100% of HLRs have been covered by the verification cases and procedures.

![Figure 5.4: Traceability information analysis with Requirements Management Gateway](image)

**PEER REVIEW WITH SCADE LIFECYCLE REPORTER**

Peer review of SCADE Suite models may reveal problems that cannot be detected by simulation, for example extraneous constructs and/or Scade operators. Such a peer review is focused on HLR/LLR traceability analysis and robustness analysis. For this review a report containing all SCADE Suite model data is automatically generated by **SCADE LifeCycle Reporter**. SCADE LifeCycle Reporter is qualified as verification tool for DO-178C/DO-330 at TQL-5. This qualification ensures completeness and consistency of the generated report according to the input SCADE Suite model. The Scade notation has several advantages compared to a textual notation:

- Its formal definition: the description is not subject to interpretation
- Its graphical representation is simple and intuitive

**FORMAL VERIFICATION WITH SCADE SUITE DESIGN VERIFIER**

Formal methods are complementary to simulation and testing techniques for the verification of software. The DO-333 technical supplement (see §2.1.6) is applicable in conjunction with DO-178C when formal methods used as part of the software life cycle [DO-333].

**SCADE Suite Design Verifier** fourth provides an original and powerful verification technique based on formal verification technologies.

---

4. Powered by Prover® Plug-In. Prover, Prover Technology, Prover Plug-In and the Prover logo are trademarks or registered trademarks of Prover Technology AB in Sweden, the United States and in other countries.
Formal verification of software consists of a set of activities using a mathematical framework to reason about software behaviors and properties in a rigorous way. The recipe for formal verification of safety properties is:

1. Define a formal model of the software; namely a mathematical model representing the states of a software and its behaviors. When modeling LLRs in the Scade language, the model is already formal, so there is no additional formalization effort required.

2. Define for the formal model a set of formal properties to verify. These properties correspond to high-level requirements or system requirements.

3. Perform state space exploration to mathematically analyze the validity of the safety properties.

Assume one has a landing gear control system, which may trigger a landing gear retraction command. Assume one wants to verify the following safety property:

"for all possible behaviors of this controller, it will never send a landing gear retraction command while the aircraft is in landing mode or on the ground"

In a Scade operator one would express the safety property shown in Figure 5.5 below, reflecting the property above. This operator is called an observer.

Then, we would connect the observer operator to the controller in a verification context operator, as shown in Figure 5.6 below.

In very specific contexts, Design Verifier may support the detection of specification errors at the early phase of the software flow, minimizing the risk of discovering these errors during the final integration and validation phases. Design Verifier is not a qualified tool. It can only be used as additional verification means in particular for logical-oriented applications, where relevant.

5.3.3 Model accuracy and consistency

SCADE Suite’s Syntactic and Semantic Checkers perform an in-depth analysis of model consistency, including:

---

5. Even if it is the second objective in Table MBA-4, it is verified first.
SCADE Suite with DO-178C Objectives

- Detection of missing definitions
- Warnings on unused definitions
- Detection of non-initialized variables
- Coherence of data types and interfaces
- Detection of causality issues on data-flow dependencies
- Coherence of “clocks,” namely of production/consumption rates of data

5.3.4 Compatibility with target computer

For SCADE Suite-generated code, verifying compliance of the SCADE Suite models with the target computer uses the following techniques:

- Execution of the SCADE Suite Compiler Verification Kit (CVK) on target platform
- Analysis of SCADE Suite model complexity
- Analysis of SCADE Suite model execution time and memory size with SCADE Suite Time and Stack Optimizer (TSO) and SCADE Suite Time and Stack Verifiers (TSV)

CVK EXECUTION ON TARGET PLATFORM

The purpose of CVK is to verify that the type of code generated by SCADE Suite KCG is correctly compiled/executed with a given cross-compiler on target platform.

CVK relies on a sample-based approach such as described in DO-248C DP#12. This approach is relevant due to the characteristics of SCADE Suite KCG generated code: regular patterns that strictly conform to restricted coding standards defined in [KCG-TOR] documentation.

For further information related to CVK principles and CVK development strategy, refer to Appendix D.

SCADE MODEL COMPLEXITY ANALYSIS

The main objective is to monitor the complexity of SCADE Suite models to avoid potential issues during the software development and target execution.

The rules related to the management of SCADE Suite model complexity are defined in the SCADE Development Standards [SC-SDVST]. Such rules must be checked either automatically or manually.

In the context of automatic verification, the user is able to develop its own design rule checker by using the Tcl and/or JAVA scripting capabilities supported by SCADE Suite.

For further information on Tcl and/or JAVA SCADE Suite API, refer to SCADE Suite Technical Manual [SUITE-TM].

SCADE MODEL EXECUTION TIME AND MEMORY ANALYSIS

The main objective of this analysis is to anticipate potential timing problems and stack usage problems during the software design phase.
Timing problem: The ability of an application to complete its task on time using a given CPU is usually addressed during target integration testing. Schedulability analysis must be performed to demonstrate the properties of the integrated system with respect to timing requirements. Hence it is necessary to determine an upper bound for execution time, which results from a process called Worst-Case Execution Time (WCET) analysis.

Measurement of WCET raises several challenges that impose major costs and risks on the integration testing phase of any software development project:

- Measurement is only possible when all elements of the system are available: application software, system software, target system, and a complete set of test cases. It is often too late when a problem is found in these project phases. Late changes of software and/or target result in very high costs and risky delays.
- Measurement is not precise or implies code instrumentation which may alter test results in non-predictable ways.
- Tracing of execution time phenomena back to code or even to the model is very tedious, if even possible, and imposes serious challenges on the root cause analysis of such effects.
- Measurements cannot be demonstrated to be safe (i.e., is it really the worst case we encountered?).

Stack usage problem: Stack overflow is also a serious safety issue. The absence of stack overflow is a property that must be demonstrated during target integration verification. However, the nature and complexity of the problem makes prediction and avoidance very hard to achieve and even harder to demonstrate. A common and traditional method for verifying stack usage is to write a short program which fills the stack with a given bit-pattern, and then execute the application and count how many stack registers still have the bit-pattern.

But how can you be sure that you really have the most pessimistic execution order and data usage in your application?

SCADE Suite includes two different modules that support timing and stack analysis of SCADE Suite models:

**Timing and Stack Optimizer (TSO)** computes the WCET and stack size estimation for a generic platform. TSO is usually used to compare different versions of a SCADE Suite model to determine the most efficient design. SCADE Suite users can use it to monitor the performances of their SCADE Suite design with respect to WCET and stack usage. This tool is relevant, in particular, for early verification of the compatibility between the SCADE Suite model and the target platform.

**Timing and Stack Verifiers (TSV)** compute precise WCET and stack size for a SCADE Suite model on a specific hardware target. Such analysis runs with respect to specific target processors and C compilers, and requires fine-grained customization to comply with the hardware characteristics. Even if TSV is still relevant during early verification of the target compatibility analysis, its operating mode is quite complex (due to the number of parameters...
to be set) and it is usually relevant only when precise WCET and stack size measurements are required during final integration testing on target platform.

**Timing and Stack Optimizer** and **Timing and Stack Verifiers** are fully integrated into the SCADE Suite environment. The analysis results are directly shown within SCADE Suite and hyperlinks are available for direct reference to the model constructs matching each WCET and/or stack size results.

**Figure 5.7** illustrates global visualization results. **Figure 5.8** illustrates global and detailed results for Timing analysis.

For further information on TSO/TSV, refer to SCADE Suite User Manual [SUITE-UM].
5.3.5 Verifiability

The SCADE Suite model describes the low-level requirements and the architecture of the corresponding software part. Since the Scade notation has a formal definition, a SCADE Suite model is formally verifiable.

Such verifiability is confirmed by SCADE Suite Syntactic and Semantic Checkers if the SCADE Suite models satisfy the Scade language syntax and semantics, and no errors or no warnings that cannot be justified are raised.

The SCADE Suite model complexity must be also monitored to ensure design verifiability. For more information, see SCADE Suite model complexity analysis in Section 5.3.4 and model complexity rules in [SC_SDVST].

5.3.6 Conformity to standards

Two levels of rules must be considered for SCADE Suite LLRs:

- **SCADE Suite KCG built-in rules** for the Scade language defined in [SCS-KCG-LRM]. This document mathematically defines the syntactic and semantic rules that a Scade model has to follow. These rules are automatically verified with SCADE Suite KCG Semantic Checker,

- **User design rules** related to SCADE Suite models are defined in [SC_SDVST]. These rules must be checked either automatically or manually. In the context of automatic verification, users are able to develop their own design rule checker by using the Tcl and/or JAVA scripting capabilities supported by SCADE Suite.

5.3.7 Traceability from SCADE Suite LLR to HLR

HLR/LLR bi-directional traceability is required as stated in [DO-178C], §5.5. For the definition and granularity of a SCADE Suite LLR within a SCADE Suite model, please refer to Section 4.3.2.

Trace data must confirm that:

- All HLRs are covered by SCADE Suite LLRs;
- All SCADE Suite LLRs are correctly traced to HLRs;
- All SCADE Suite LLRs that are not traced to HLRs are explicitly identified as derived SCADE Suite LLRs by design choice.

Derived requirements must be provided to safety process according to [DO-178C] §2.3. Other untraced SCADE Suite LLRs must be removed from the design.

This traceability analysis is efficiently supported by SCADE LifeCycle Requirements Management Gateway that provides an integrated solution:

- To capture HLR/LLR traceability from the SCADE environment with a simple drag and drop user action (see Figure 5.9)
• To generate automatically built-in or customized traceability reports (see Figure 5.10)

Figure 5.9: Capturing LLR/HLR traceability links with Requirements Management Gateway

Figure 5.10: Built-in report selection or customization of traceability report for HLR/LLR
5.3.8 Algorithms accuracy

The accuracy of algorithms is verified through a combination of model simulation and peer review.

The review of SCADE Suite LLR algorithms focuses on the analysis of numerical algorithms to verify their robustness to precision issues and detect potential numerical issues (division by zero, overflow, etc.). Simulation of SCADE Suite models with SCADE Test Environment for Host is a strong support to the verification of numerical algorithms. This technique may reveal failure of an algorithm such as convergence and/or precision issues.

For further information on SCADE Test Environment for Host, refer to Section 5.3.2.

5.3.9 Partitioning

SCADE Suite introduces no specific risks, but provides no partitioning mechanism. Partitioning is beyond the scope of SCADE Suite. It has to be ensured by low-layer hardware and software mechanisms such as memory partitioning and interrupt service routines. This is provided by operating systems such as VxWorks 653 from Wind River®.

5.3.10 Verification of simulation cases, procedures and results (MB. specific objectives)

The objectives MB.A-4#MB14, #MB15 and #MB16 are required when simulation is used as a means of compliance for objectives 1, 2, 4, 7, 8, 9, or 11 of table MB.A-4 (see [DO-331], Table MB.A-4 Item 1).

The verification of SCADE Suite verification cases, procedures, and results relies on peer review (see Section 5.3.2 for definition of SCADE Suite verification cases and procedures).

The review of SCADE Suite verification cases must confirm that:

- SCADE Suite verification cases are traceable to HLRs
- SCADE Suite verification cases satisfy criteria of normal and robustness testing
- All HLRs are covered by SCADE Suite verification cases

The review of SCADE Suite verification procedures shall confirm that SCADE Suite verification cases, including expected results, are correctly developed into SCADE Suite verification procedures.

The review of SCADE simulation results must confirm that:

- Simulation results are expected results
- Discrepancies between actual and expected results generate problem reports
SCADE Suite simulation results are generated by SCADE Test Environment for Host and include a pass/fail status for each verification case. Note that the qualification of SCADE Test Environment for Host (DO-330 TQL-5) ensures that simulation results are correctly evaluated.

### 5.3.11 Verification summary for LLR and architecture

Table 5.4 summarizes verification objectives and methods for the software low-level requirements and architecture.

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Ref</th>
<th>Activity Ref</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Low-level requirements comply with high-level requirements</td>
<td>MB.6.3.2.a</td>
<td>MB.6.3.2 MB.6.7 MB.6.8.1 (see Item 1)</td>
<td>Model Simulation with SCADE Test Environment for Host Model Coverage with SCADE Test Model Coverage Peer review with SCADE LifeCycle Reporter</td>
</tr>
<tr>
<td>2 Low-level requirements are accurate and consistent</td>
<td>MB.6.3.2.b</td>
<td>MB.6.3.2 MB.6.8.1 (see Item 1)</td>
<td>Automated by SCADE Suite Syntactic and Semantic Checker</td>
</tr>
<tr>
<td>3 Low-level requirements are compatible with target computer</td>
<td>MB.6.3.2.c</td>
<td>MB.6.3.2</td>
<td>SCADE Suite CVK Execution on target Analysis of SCADE Suite model complexity</td>
</tr>
<tr>
<td>4 Low-level requirements are verifiable</td>
<td>MB.6.3.2.d</td>
<td>MB.6.3.2 MB.6.8.1 (see Item 1)</td>
<td>Automated by SCADE Suite Syntactic and Semantic Checker Analysis of SCADE Suite model complexity</td>
</tr>
<tr>
<td>5 Low-level requirements conform to standards</td>
<td>MB.6.3.2.e</td>
<td>MB.6.3.2</td>
<td>SCADE Suite KCG built-in rules: Automated by SCADE Suite Syntactic and Semantic Checker User design rules: peer review or automated by design rule checker (if existing)</td>
</tr>
<tr>
<td>6 Low-level requirements are traceable to high-level requirements</td>
<td>MB.6.3.2.f</td>
<td>MB.6.3.2</td>
<td>Traceability analysis with SCADE LifeCycle Requirements Management Gateway</td>
</tr>
<tr>
<td>7 Algorithms are accurate</td>
<td>MB.6.3.2.g</td>
<td>MB.6.3.2 MB.6.8.1 (see Item 1)</td>
<td>Peer review Model Simulation of numerical algorithms with SCADE Test Environment for Host</td>
</tr>
<tr>
<td>8 Software architecture is compatible with high-level requirements</td>
<td>MB.6.3.3.a</td>
<td>MB.6.3.3 MB.6.8.1 (see Item 1)</td>
<td>Peer review with SCADE LifeCycle Reporter</td>
</tr>
<tr>
<td>9 Software architecture is consistent</td>
<td>MB.6.3.3.b</td>
<td>MB.6.3.3 MB.6.8.1 (see Item 1)</td>
<td>Automated by SCADE Suite Syntactic and Semantic Checker</td>
</tr>
<tr>
<td>10 Software architecture is compatible with target computer</td>
<td>MB.6.3.3.c</td>
<td>MB.6.3.3</td>
<td>SCADE Suite CVK Execution on target Analysis of SCADE Suite architecture model complexity</td>
</tr>
</tbody>
</table>
Table 5.4: DO-331 Table MB.A-4 Objectives Achievement (Continued)

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Ref</th>
<th>Activity Ref</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>11  Software architecture is verifiable</td>
<td>MB.6.3.3.d</td>
<td>MB.6.3.3</td>
<td>Automated by SCADE Suite Syntactic and Semantic Checker Analysis of SCADE Suite architecture model complexity</td>
</tr>
<tr>
<td>11  Software architecture conforms to standards</td>
<td>MB.6.3.3.e</td>
<td>MB.6.3.3</td>
<td>SCADE Suite KCG built-in rules: Automated by SCADE Suite Syntactic and Semantic Checker User design rules: peer review or automated by design rule checker (if existing)</td>
</tr>
<tr>
<td>13  Software partitioning integrity is confirmed</td>
<td>MB.6.3.3.f</td>
<td>MB.6.3.3</td>
<td>SCADE Suite introduces no specific risk, but provides no partitioning mechanism; traditional method has to be used</td>
</tr>
<tr>
<td>MB 14 Simulation cases are correct (see Item 1)</td>
<td>MB.6.8.3.2.a</td>
<td>MB.6.8.1 MB.6.8.3.2</td>
<td>Peer review of SCADE Suite verification cases</td>
</tr>
<tr>
<td>MB 15 Simulation procedures are correct (see Item 1)</td>
<td>MB.6.8.3.2.b</td>
<td>MB.6.8.1 MB.6.8.3.2</td>
<td>Peer review of SCADE Suite verification procedures</td>
</tr>
<tr>
<td>MB 16 Simulation results are correct and discrepancies explained (see Item 1)</td>
<td>MB.6.8.3.2.c</td>
<td>MB.6.8.1 MB.6.8.3.2</td>
<td>Analysis of test conformance report generated by SCADE Test Environment for Host</td>
</tr>
</tbody>
</table>

Item 1: As described in section MB. 6.8.1 of this supplement [DO-331], simulation may be used as a means of compliance for objectives 1, 2, 4, 7, 9, or 11 of this table. If simulation is used as this means, objectives MB.14, MB.15, and MB.16 are required.
5.4 Verification of Coding Outputs and Integration Process

5.4.1 Verification objectives for coding output and integration process

Table 5.5 lists verification objectives for outputs of the coding and integration process.

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Activity Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Source code complies with low-level requirements</td>
<td>MB.6.3.4</td>
</tr>
<tr>
<td>2. Source code complies with software architecture</td>
<td>MB.6.3.4</td>
</tr>
<tr>
<td>3. Source code is verifiable</td>
<td>MB.6.3.4</td>
</tr>
<tr>
<td>4. Source code conforms to standards</td>
<td>MB.6.3.4</td>
</tr>
<tr>
<td>5. Source code is traceable to low-level requirements</td>
<td>MB.6.3.4</td>
</tr>
<tr>
<td>6. Source code is accurate and consistent</td>
<td>MB.6.3.4</td>
</tr>
<tr>
<td>7. Output of software integration process is complete and correct</td>
<td>6.3.5</td>
</tr>
<tr>
<td>8. Parameter Data Item File is correct and complete</td>
<td>6.6</td>
</tr>
<tr>
<td>9. Verification of Parameter Data Item File is achieved</td>
<td>6.6</td>
</tr>
</tbody>
</table>

5.4.2 Impact of code generator qualification

The KCG Code Generator is qualified as a Criteria 1 tool because it was developed by Esterel Technologies to fulfill the DO-330 TQL-1 objectives (see Appendix C for details about qualification).

This has the following consequences:

**SOURCE CODE COMPLIES WITH LOW-LEVEL REQUIREMENTS**

The qualification of SCADE Suite KCG ensures that the source code generated from any correct set of SCADE Suite models complies with SCADE Suite LLRs contained in these models. Note that if the SCADE Suite model is not correct, no code is generated.

**SOURCE CODE COMPLIES WITH SOFTWARE ARCHITECTURE**

The qualification of SCADE Suite KCG ensures that the architecture of the source code generated from any correct set of SCADE Suite models complies with the software architecture.

The architecture of SCADE Suite KCG Generated Code is determined by SCADE Suite users. The definition of the architecture includes the SCADE Suite model structure, expansion directives, and interface directives as explained in Section 4.4.1, "Tuning Code to Target and Project Constraints."
SOURCE CODE IS VERIFIABLE
The qualification of SCADE Suite KCG ensures that the code structures generated from any correct set of SCADE Suite models have a simple and clear meaning, reflecting elements of the models.

SOURCE CODE CONFORMS TO STANDARDS
The qualification of SCADE Suite KCG ensures that the source code generated from any correct set of SCADE Suite models satisfies coding rules defined in SCADE Suite KCG Tool Operational Requirements (TOR) document [KCG-TOR].

SOURCE CODE IS TRACEABLE TO LOW-LEVEL REQUIREMENTS
The qualification of SCADE Suite KCG ensures that the source code generated from any correct set of SCADE Suite models is traceable to LLRs contained in these models.

SOURCE CODE IS ACCURATE AND CONSISTENT
The qualification of SCADE Suite KCG ensures that the source code generated from any correct set of SCADE Suite models accurately and consistently reflects the SCADE Suite model. This evidence is based on the requirements (see TOR document) of KCG that include:

• The verification that the input model complies with the syntactic/semantic rules of the Scade language;

• A code generation scheme ensuring that the generated code reflects the model.

The objectives listed above are met thanks to SCADE Suite KCG qualification, provided that the code was successfully generated by SCADE Suite KCG. This is confirmed by analysis of code generation logs.

OUTPUT OF THE SOFTWARE INTEGRATION PROCESS IS COMPLETE AND CORRECT
The verification of Executable Object Code (EOC) integration is a review of compiling, linking, and loading data to confirm that the EOC was built in a complete and correct way according to the software build and load procedure. This objective is independent from the fact that the EOC is obtained from SCADE-generated code or not.

5.4.3 Verification of parameter data item

According to DO-178C, §2.5.1, a Parameter Data Item (PDI) is a set of data that influences the behavior of software without modifying the Executable Object Code (EOC) and that is managed as a separate configuration item.

The verification of PDI is addressed in DO-178C, §6.6 and is out of the scope of this document related to Model-Based-Development with SCADE.
5.4.4 Verification summary for coding output and integration process

Table 5.6 summarizes verification objectives and methods for coding outputs and integration process.

Table 5.6: DO-331 Table MB.A-5 Objectives

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Activity Ref</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Source code complies with low-level requirements</td>
<td>MB.6.3.4</td>
<td>Ensured by SCADE Suite KCG qualification¹</td>
</tr>
<tr>
<td>2 Source code complies with software architecture</td>
<td>MB.6.3.4</td>
<td>Ensured by SCADE Suite KCG qualification¹</td>
</tr>
<tr>
<td>3 Source code is verifiable</td>
<td>MB.6.3.4</td>
<td>Ensured by SCADE Suite KCG qualification¹</td>
</tr>
<tr>
<td>4 Source code conforms to standards</td>
<td>MB.6.3.4</td>
<td>Ensured by SCADE Suite KCG qualification¹</td>
</tr>
<tr>
<td>5 Source code is traceable to low-level requirements</td>
<td>MB.6.3.4</td>
<td>Ensured by SCADE Suite KCG qualification¹</td>
</tr>
<tr>
<td>6 Source code is accurate and consistent</td>
<td>MB.6.3.4</td>
<td>Ensured by SCADE Suite KCG qualification¹</td>
</tr>
<tr>
<td>7 Output of software integration process is complete and correct</td>
<td>6.3.5</td>
<td>Analysis of compiling/linking/loading data</td>
</tr>
<tr>
<td>8 Parameter Data Item File is correct and complete</td>
<td>6.6</td>
<td>Not SCADE-specific; traditional method has to be used</td>
</tr>
<tr>
<td>9 Verification of Parameter Data Item File is achieved</td>
<td>6.6</td>
<td>Not SCADE-specific; traditional method has to be used</td>
</tr>
</tbody>
</table>

1. Users must verify the absence of any errors in the log file generated by SCADE Suite KCG.

5.5 Testing of Outputs from Integration Process

5.5.1 Testing objectives for outputs from integration process

Table 5.7 lists the verification objectives for testing outputs of the integration process.

Table 5.7: DO-331 Table MB.A-6

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Activity Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Executable Object Code complies with high-level requirements</td>
<td>6.4.2</td>
</tr>
<tr>
<td>2 Executable Object Code is robust with high-level requirements</td>
<td>6.4.2.1</td>
</tr>
<tr>
<td>3 Executable Object Code complies with low-level requirements</td>
<td>6.4.3</td>
</tr>
<tr>
<td>4 Executable Object Code is robust with low-level requirements</td>
<td>6.5</td>
</tr>
<tr>
<td>5 Executable Object Code is compatible with target computer</td>
<td>6.4.1.a</td>
</tr>
</tbody>
</table>

Item1: As described in section MB.6.8.2.a of this supplement [DO-331], the MB.6.8.2.a is only required when simulation is used as a means of compliance for objectives 1 or 2 of this table.
5.5.2 SCADE Combined Testing

Process overview

The Combined Testing Process (CTP) is a SCADE Suite model-based efficient and optimized testing process to fully satisfy the DO-178C Table MB.A-6 objectives while optimizing testing efforts.

1. **CTP is efficient**: SCADE Suite verification cases and procedures are primarily developed from HLRs. This verification strategy focuses first on HLR functionality and integration issues that are often poorly and lately addressed in a traditional verification process.

2. **CTP optimizes testing efforts**: In the context of level A and B applications, the development of verification cases and procedures usually requires a huge effort to satisfy all testing objectives. When using SCADE, this testing effort is significantly reduced for the following reasons:

   - Common requirement-based verification cases and procedures (see Section 5.3.2) are used for both model simulation on host and testing on target as in Figure 5.11.

   - There is no need to develop additional SCADE test cases and procedures for SCADE Suite LLRs that are already covered by HLRs-based SCADE test cases and procedures. As stated in [DO-178C], §6.4: “If a test case and its corresponding test procedure are developed and executed for hardware/software integration testing or software integration testing, and satisfy the requirements-based coverage and structural coverage, it is not necessary to duplicate the test for low-level testing.”

Figure 5.12 provides an overview of the SCADE Combined Testing Process.

- The testing effort is mainly focused on HLR-based testing for the application code and most of low-level tests can be removed for this software part that may change several times during the software life cycle. On the other side, low-level library components and drivers are usually developed with a traditional approach (manual coding) and low-level tests must be considered in this context. Because the corresponding code is quite stable during the software life cycle, the additional testing effort is not significant for this software part.
5.5.3 Compliance of EOC with HLRs (MB.A-6 #1) and robustness with HLRs (MB.A-6 #2)

SCADE test cases and procedures are developed primarily on the basis of HLRs and executed in the target environment. They should include both normal range test cases and robustness test cases.

For this, the user is able to reuse existing simulation cases developed for design verification (see Section 5.3.2) with the support of SCADE Test Target Execution as in Figure 5.13.

Figure 5.13: Factor simulation and test cases with SCADE Test Target Execution

5.5.4 Compliance of EOC to LLRs (MB.A-6 #3)

TRACEABLE LLRS

Test cases for traceable LLRs can be shared with HLR tests wherever appropriate: there is no need to develop additional SCADE test cases and procedures for SCADE Suite LLRs that are already covered by HLRs-based SCADE test cases and procedures.

The assessment of SCADE Suite LLRs coverage is supported by SCADE Test Model Coverage. Full model coverage with SCADE Test Model Coverage is required to verify all LLRs were sufficiently exercised by the test cases as highlighted in Figure 5.12.

For this, the user is able to reuse existing simulation cases developed for design verification (see Section 5.3.2) with the support of SCADE Test Target Execution as in Figure 5.13.

DERIVED LLRS

Additionally, verification objectives of Table MB.A-6 #3 require specific derived-LLRs based testing. These tests should include both normal and robustness considerations.

The testing strategy depends on the design choice selected for this derived LLR.

Regarding numerical computations algorithms that are not managed at library level, there is still a need for additional analyses to ensure the coverage of equivalence classes of such numerical computations and complementary test cases should be developed for non-covered equivalence classes, if any.
1. The derived LLR was implemented as a SCADE Suite library operator: the applicant must test the implementation of this operator from the HLRs established for this operator (component-based testing) and additionally, the integration of this operator within upper-level operators. This activity is complete when full model coverage is achieved with SCADE Test Model Coverage for both structural coverage criteria and integration coverage criteria and for coverage of equivalence classes other than those addressed by Test Model Coverage. For further information on Test Model Coverage and its coverage criteria, refer to Chapter 6.

2. The derived LLR was implemented directly within the applicative part (no factoring effect): specific LLRs-based testing must be considered in this case. The objective is to minimize as much as possible the number of non-library-based derived LLRs to maximize the benefit of model-based verification with SCADE Test.

5.5.5 Robustness of EOC with LLRs (MB.A-6 #4)

A robust design strategy is key, not only to make an application robust, but also to optimize the verification efforts required to verify the ability of the software to respond to abnormal inputs and conditions.

A robust design strategy may rely on a library of robust operators as in Figure 5.14. Two benefits can be highlighted in this context:

1. the corresponding application inherits robustness from its robust components;
2. the verification strategy of such robust components is optimized because the library operator is tested once according to its robustness requirements.

![Figure 5.14: Robustness Strategy with SCADE Suite design](image)

First, these library operators are unit-tested. Then, the verification of the integration of such robust components within the application is addressed in the context of MB.A-6 #2 objectives and is fully supported by SCADE Test (see Section 5.5.3).

In cases where robust LLRs are not implemented as library operators, specific robustness verification activities must be considered including low-level robustness tests.

5.5.6 Compatibility of EOC with target (MB.A-6 #5)

Compatibility of the EOC with target computer is verified by HW/ SW integration testing of the whole application in the target environment.
The whole software application usually includes several components (developed with SCADE or manually coded) and its scope can be beyond the SCADE Suite application itself.

Target testing of the whole system is generally performed from system-based requirements on a real test bench that includes communication drivers with interfaces such as ARINC 429 and/or ARINC 664 (AFDX).

5.5.7 Verification summary for testing outputs from integration process

Table 5.8 summarizes verification objectives and methods for testing outputs of the integration process.

Table 5.8: DO-331 Table MB.A-6 Objectives Achievement

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Activity Ref</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Executable object code complies with high-level requirements</td>
<td>6.4.2 6.4.2.1 6.4.3 6.5 MB.6.8.2.a (see Item 1)</td>
<td>HLRs-based testing in the target environment with SCADE Test Target Execution</td>
</tr>
<tr>
<td>2 Executable object code is robust with high-level requirements</td>
<td>6.4.2 6.4.2 6.4.3 6.5 MB.6.8.2.a (see Item 1)</td>
<td>HLRs-based robustness testing in the target environment with SCADE Test Target Execution</td>
</tr>
<tr>
<td>3 Executable object code complies with low-level requirements</td>
<td>6.4.2 6.4.2.1 6.4.3 6.5</td>
<td>HLRs-based testing in the target environment with full model coverage with SCADE Test Target Execution Requirements-based testing of library elements for derived LLRs implemented as SCADE Suite library operators with SCADE Test Target Execution Analysis of model coverage and equivalence classes for non-library-based numerical computations</td>
</tr>
<tr>
<td>4 Executable object code is robust with low-level requirements</td>
<td>6.4.2 6.4.2.2 6.4.3 6.5</td>
<td>Requirements-based robustness testing of library elements with SCADE Test Target Execution Complementary robustness low-level testing for functions not based on robust libraries</td>
</tr>
<tr>
<td>5 Executable object code is compatible with target computer</td>
<td>6.4.1.a 6.4.3.a</td>
<td>HW/SW Integration Testing of the whole application</td>
</tr>
</tbody>
</table>
6. Verification of the Verification Activities

6.1 Verification Objectives

As stated in [DO-178C] §6, the software verification process is a technical assessment not only of the outputs of the software planning process and software development processes but also of the outputs of the software verification process. In this context, we usually talk about the “verification of the verification outputs” with the objective to assess how well the verification activities mentioned in chapter 5 were performed.

Table 6.1 summarizes the objectives for the verification of verification process results.

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Activity Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Test procedures are correct</td>
<td>6.4.5</td>
</tr>
<tr>
<td>2 Test results are correct and discrepancies are explained</td>
<td>6.4.5</td>
</tr>
<tr>
<td>3 Test coverage of high-level requirements is achieved</td>
<td>6.4.4.1 MB.6.8.2.a</td>
</tr>
<tr>
<td>4 Test coverage of low-level requirements is achieved</td>
<td>6.4.4.1 MB.6.7</td>
</tr>
<tr>
<td>5 Test coverage of software structure (modified condition/decision coverage) is achieved</td>
<td>6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3 MB.6.8.2.b (see Item 1)</td>
</tr>
<tr>
<td>6 Test coverage of software structure (decision coverage) is achieved</td>
<td>6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3 MB.6.8.2.b (see Item 1)</td>
</tr>
<tr>
<td>7 Test coverage of software structure (statement coverage) is achieved</td>
<td>6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3 MB.6.8.2.b (see Item 1)</td>
</tr>
<tr>
<td>8 Test coverage of software structure (data coupling and control coupling) is achieved</td>
<td>6.4.4.2.c 6.4.4.2.d 6.4.4.3 MB.6.8.2.b</td>
</tr>
<tr>
<td>9 Verification of additional code, that cannot be traced to Source Code, is achieved</td>
<td>6.4.4.2.b</td>
</tr>
</tbody>
</table>

Table 6.1: DO-331 Table MB.A-7 (Continued)

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Activity Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB 10 Simulation cases are correct (see Item 2)</td>
<td>MB.6.8.3.2</td>
</tr>
<tr>
<td>MB 11 Simulation procedures are correct (see Item 2)</td>
<td>MB.6.8.3.2</td>
</tr>
<tr>
<td>MB 12 Simulation results are correct and discrepancies explained (see Item 2)</td>
<td>MB.6.8.3.2</td>
</tr>
</tbody>
</table>

Table 6.1: DO-331 Table MB.A-7 (Continued)

Item 1: As described in section MB.6.6.2.b of supplement [DO-331], the MB.6.6.2.b activity is only required when simulation is used as a means of compliance of any objectives 5, 6, 7, or 8 of this table.

Item 2: As described in section MB. 6.8.2 of supplement [DO-331], these three objectives are only required when simulation is used as a means of compliance of objectives 1 and 2 of Annex Table MB.A-6.
6.2 Verification of Test Procedures and Results

The review of SCADE test cases must confirm:

- SCADE test cases are traceable to HLRs;
- SCADE test cases satisfy criteria of normal and robustness testing;
- All HLRs are covered by SCADE test cases.

The review of SCADE test procedures must confirm that SCADE test cases, including expected results, are correctly developed into SCADE test procedures.

As illustrated by Figure 5.11 and Figure 5.13, both simulation and test cases are factorized with the support of SCADE Test Target Execution. In this context, the qualification of SCADE Test Target Execution as a DO-330/TQL-5 tool removes the need for reviewing target test harnesses if SCADE simulation cases and procedures were already reviewed during SCADE Suite design verification.

The review of test results must confirm:

- SCADE test results are correct;
- Discrepancies between actual and expected results generate problem reports.

SCADE test results are usually generated by the user target testing environments and include a pass/fail status. Typical testing environment for target such as RTRT from IBM, TestBed® from LDRA, or VectorCAST® from Vector Software support a qualification process at DO-330/TQL-5 and ensure that test results are correctly evaluated.

Traceability between test procedures and test results can be managed with SCADE LifeCycle Requirements Management Gateway (see example of traceability management with tests in Figure 5.4).

6.3 HLR Coverage Analysis

The objective of this activity is to verify that the HLRs are fully covered by test cases.

This is achieved by peer review of HLRs/SCADE test cases traceability matrices. Such matrices can be easily generated with SCADE LifeCycle Requirements Management Gateway (see Figure 5.4).

If common requirement-based verification cases and procedures are used for both model simulation on host and testing on target, HLRs/SCADE Suite verification cases traceability analysis was already performed (partially or fully) in the context of model simulation to satisfy some objectives of Table MB.A-4 (see Section 5.3.1).

6.4 Coverage Analysis with SCADE Test Model Coverage

SCADE coverage analysis designates both Model Coverage of SCADE Suite models and Code Coverage of SCADE Suite KCG-generated code. Both are measured with SCADE Test Model Coverage from HLR-based test cases and procedures.
6.4.1 Test Model Coverage overview

Test Model Coverage takes as inputs a SCADE Suite model and a set of HLRs-based test cases and procedures and supports two processing flows as illustrated below.

- Model coverage analysis with Model-Level Coverage
- Code coverage analysis with Code-Level Coverage

![Diagram](image)

**Figure 6.1:** Test Model Coverage as integrated solution for model and code coverage analysis

The creation of the test cases and procedures required to perform coverage analysis is done only once. Measurements are performed using the same SCADE Suite model and the same set of test cases and procedures. Model-Level Coverage and Code-Level Coverage share the same format of test case scenarios developed from the high-level requirements from which the SCADE Suite model is derived.

SCADE Test Model Coverage generates two distinct reports: a model coverage report and a code coverage report.

Both model coverage analysis and code coverage analysis must be addressed at model level and focus on the functional origin of coverage holes, whether they are due to lack of testing, inadequate high-level requirements, or dead, deactivated, or unintended low-level requirements.

The following sections describe the activities to be addressed in order to satisfy the DO-178C MB.A-7#4 to #7 objectives.

6.4.2 SCADE Suite LLRs coverage analysis (MB.A-7#4)

The objective of this activity is to verify that the LLRs are fully covered by test cases.

In the context of SCADE development, LLRs are described in the form of SCADE Suite models and model coverage analysis is a means of assessing how far the behavior of a model was explored. It is complementary to HLR/LLR traceability analysis and high-level requirements coverage analysis.

Model coverage analysis verifies that every element of the SCADE Suite model (representing a LLR) was fully exercised when requirements-based tests are exercised. It supports in particular the detection of unintended functions in the SCADE Suite model (see Section 2.4.4 and Section 5.3.1).
6.4.2.1 LLR coverage analysis with Model-Level Coverage

Model-Level Coverage measures the coverage of a SCADE Suite model by high-level requirements-based test cases. The purpose of this measure is to assess how thoroughly the SCADE Suite model was exercised. Model Coverage criteria are based on the observation of operator features activation, each operator being associated with a set of characteristic features (see "Model coverage criteria").

Figure 6.2 shows the position of SCADE Test Model Coverage within the software verification flow.

The use of SCADE Test Model Coverage is decomposed in the following phases:

1 **Model Coverage Acquisition**: Running test cases with the SCADE Test Environment for Host module, while measuring the coverage of each operator.

2 **Model Coverage Analysis**: Identifying the Scade operators that are not fully covered.

3 **Model Coverage Resolution**: Adding test cases or providing the explanation or the necessary fixes for each operator that is not fully covered. Fixes can be in the high-level requirements, in the SCADE Suite model, or both.

Figure 6.3 illustrates the use of SCADE Test Model Coverage. The coverage of each operator is indicated via colors and percentages. The tool provides also detailed explanations about operator features that are not fully covered.
Model coverage holes may reveal the following deficiencies:

1. **Shortcomings in high-level requirements-based test cases and/or procedures:** In that case, resolution consists in adding missing requirements-based test cases and/or procedures.

2. **Inadequacies or shortcomings in the high-level requirements:** In that case, resolution consists in fixing HLRs and updating the test suite.

3. **Previously unidentified derived-requirements:** In that case, the appropriate derived-requirement-based test cases and procedures must be developed and executed to provide the missing coverage (see Section 5.5.4 for derived low-level testing strategy).

4. **Deactivated functionality in SCADE Suite model:** Resolution must be done according to DO-331 MB.6.7.2.d. Moreover, the deactivated functionality should be identified as such in the design.
5 Unintended functionality in SCADE Suite model: In that case, resolution consists in removing the functionality and assessing the effects and needs for re-verification.

**EXAMPLE 1: INSUFFICIENT TESTING**

![Figure 6.4: Non activated Confirmator](image1)

- **Analysis:** The Confirmator in Figure 6.4 was not raised during testing activities. Analysis concludes that the requirement is correct but testing is not sufficient.
- **Resolution:** Develop additional tests.

**EXAMPLE 2: LACK OF ACCURACY IN THE HLR**

The Integrator in Figure 6.5 was never reset during the tests. Is the “reset” behavior an unintended function?

![Figure 6.5: Uncovered “reset” activation](image2)

- **Analysis:** Resetting the filter here is a correct software requirement, but the HLR did not specify that changing speed regulation mode implies resetting all filters, so no test case exercised this situation.
- **Resolution:** Complement the HLR.

6.4.2.2 Model coverage criteria

The model coverage criteria of Test Model Coverage were designed to satisfy the following objectives:

- Match DO-331 model coverage principles
- Fit the entire Scade language: data flow constructs as well as control-oriented constructs (state machines, clocked blocks)
- Provide a sound and accurate assessment of the fact that every model construct and flow is exercised by test and/or simulation

Model coverage criteria defined within SCADE Test Model Coverage are strongly linked to the characteristics of SCADE Suite models:

- SCADE Suite models describe the functionality of software, while a C program describes its implementation. It creates a major difference both in terms of abstraction level (feature coverage versus code coverage) and in terms of coverage of multiple occurrences.
- SCADE Suite models are based on functional data flows and state machines, while most programming languages and their criteria are sequential.

6. For deactivated functionality expressed by a design model that is not intended to be realized in any configuration used within an aircraft or engine, a combination of analysis, simulation, and testing should show that its realization is prevented, isolated, or eliminated. For deactivated functionality expressed by a design model that is only intended to be realized in certain approved configurations used within an aircraft or engine, the operational configuration needed for normal realization of these requirements should be established and additional verification cases and verification procedures developed to satisfy the required coverage objectives. See [DO-331], §MB.6.7.2.d.
They are based on the observation of operator features activation. They capture in particular:

- The activation of data influence through their data path, not limited to Boolean data (see Basic Coverage)
- The activation of the internal computation of a given operator (called ‘coverage criteria’)
- The activation of its instantiated/called operators (called ‘integration criteria’)

These criteria include:

1. **Basic Coverage (BC)**

   They evaluate the coverage of the model in terms of data and control flow. They measure the fact that every branch of the model is activated. The definition of Basic Coverage relies on the concept of **Data Path Activity** concept as illustrated in Figure 6.6.

2. **Decision Coverage (DC)**

   In addition to the Basic Coverage measurement, these criteria measure the outcome of every Boolean expression taking both true and false values while the outcome is on a path whose Activation Condition (AC) is true.

3. **Masking Modified Condition/Decision Coverage (Masking MC/DC)**

   In addition to the Basic Coverage measurement, these criteria measure the independent effect of each condition of every Boolean expression’s outcome. Each condition should take the true and false value while its effect on the expression’s outcome is active (i.e., not masked). MC/DC satisfaction ensures DC satisfaction.
Masking MC/DC can be seen as a particular case of Data Path Activity concept for Boolean flows/expressions. The Activation Condition (AC) for every Boolean primitive of the Scade language is presented in the following table.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Activation Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a→1→s</td>
<td>AC(a,s) = b</td>
</tr>
<tr>
<td>b→1→s</td>
<td>AC(a,s) = not b</td>
</tr>
<tr>
<td>a→true</td>
<td>AC(a,s) = true</td>
</tr>
<tr>
<td>a→b→s</td>
<td>AC(a,s) = b</td>
</tr>
</tbody>
</table>

MC/DC of a path is achieved when its flows are true and false while the activation condition of that path is true. Since the activation condition is the negation of the masking condition, this corresponds to the “masking” MC/DC criterion, as described in [NASA-MCDC].

The following example illustrates the Masking MC/DC coverage of a Boolean expression.

First, considering path (A, S), there is the activation condition $AC(A, S) = \neg (B \land C)$ which may be covered by these test cases:

<table>
<thead>
<tr>
<th>Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

Considering path (B, S), there is $AC(B, S) = \neg A \land C$ which is already covered by test case 2 plus an additional test case:

<table>
<thead>
<tr>
<th>Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

Considering path (C, S), there is $AC(C, S) = \neg A \land B$ which is already covered by test case 3 plus an additional test case:

<table>
<thead>
<tr>
<th>Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

Thus, the above four test cases fully cover the Boolean expression regarding the masking MC/DC coverage criterion.

Test Model Coverage also supports the assessment of data and control coupling activities as presented below:
4. CONTROL COUPLING COVERAGE

The following criteria are addressed by Test Model Coverage:

- Operator instance activation
- Operator instance reset
- Activation of branches in “activate if” and “activate when” constructs
- Iterators (total and partial/conditional)
- States and transitions of state machines

5. DATA COUPLING COVERAGE

The following criteria are addressed by Test Model Coverage:

- Flow use taking into account the selection of active paths
- Operator instances input change

Part of these data and control coupling criteria (operator activation and input data flow change) are selectable by users from a list of predefined integration coverage criteria. These predefined integration coverage criteria apply to any user-defined operator. They are the default criteria and should be used when users consider that there is nothing more specific to capture about the external behavior than the predefined criteria.

Figure 6.4 illustrates the predefined integration coverage criteria supported by Test Model Coverage:

- Control coverage of a given operator instance (blue lightning)
- Coverage of data connections (red lightning)

In some cases, users can define their own integration coverage criteria. These user-defined coverage criteria are relevant when users consider that there is something more specific to capture about the external behavior, than the predefined criteria such as the capture of some specific modes, singular points, and/or ranges of a given function.

The following table provides an example of user-defined criteria for a SCADE IntegrTrapez library component.

Table 6.3: User-defined integration coverage criteria

<table>
<thead>
<tr>
<th>Coverage cases examples:</th>
<th>Integrator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal integration</td>
<td>IntegrTrapez</td>
</tr>
<tr>
<td>Upper saturation limitation</td>
<td>R</td>
</tr>
<tr>
<td>Lower saturation limitation</td>
<td>T</td>
</tr>
<tr>
<td>Reset</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>

Figure 6.8: Predefined integration coverage criteria
Table 6.4 summarizes the coverage criteria used by SCADE Test Model Coverage.

### Table 6.4: Coverage criteria in SCADE Test Model Coverage

<table>
<thead>
<tr>
<th>Coverage Criterion</th>
<th>Applies to</th>
<th>Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Coverage (BC)</td>
<td>Any flow type</td>
<td>Every element was on an active path.</td>
</tr>
<tr>
<td>Decision Coverage (DC)</td>
<td>Boolean expressions</td>
<td>The outcome of every Boolean expression has taken the true and false values while it is on an active path.</td>
</tr>
<tr>
<td>Masking Modified Condition/Decision Coverage (MC/DC)</td>
<td>Boolean expressions</td>
<td>Show independent effect of each condition of every Boolean expression’s outcome. Each condition should take the true and false value while its effect on the expression’s outcome is active (i.e., not masked).</td>
</tr>
<tr>
<td>Control coupling</td>
<td>Clocked constructs</td>
<td>Every state/transition, clocked block, activate branch was active, every operator instance was called.</td>
</tr>
<tr>
<td>Data coupling</td>
<td>Connections</td>
<td>Every connection between operators was active. This is checked by verifying that every input flow to every user operator instance has changed value.</td>
</tr>
</tbody>
</table>

#### 6.4.3 Source code coverage analysis (from MB.A-7#5 to MB.A-7#8)

The coverage analysis of source code generated by SCADE Suite KCG is measured with SCADE Test Model Coverage as illustrated in Figure 6.1. It determines which code structure were not exercised by the requirements-based test cases and procedures.

Structural code coverage information is reported at code level with the following criteria:

- Entry and Exit points of both functions and linear statement blocks
- Linear block statements (i.e. without branching)
- Decisions
- MC/DC

Achieving model coverage is a prerequisite to structural code coverage. The primary objective is to address coverage analysis at model level and focus on the functional origin of coverage holes. Code coverage is measured to confirm the coverage measurement at model-level.

#### 6.4.4 Data and control coupling verification (MB.A-7#8)

##### 6.4.4.1 Definitions

DO-178C requires that test coverage of the data and control coupling is achieved and it defines:

- **Data coupling** as “The dependence of a software component on data not exclusively under the control of that software component.”
- **Control coupling** as “The manner or degree by which one software component influences the execution of another software component.”
6.4.4.2 Verification of data and control coupling in SCADE Suite models

The qualification of SCADE Suite KCG and SCADE Test Model Coverage ensures that data coupling and control coupling at model level are exactly reflected in the generated code. In SCADE Suite models:

- **Data coupling** is accurately and completely described in terms of operators’ interfaces and fully explicit operators’ connections.

- **Control coupling** is accurately and completely described in terms of operators’ activation, either at every cycle of the basic clock or subject to derived clocks (conditional activation).

Data and control coupling verifications are performed by a combination of activities:

- It is verified first by design review and analysis with SCADE Suite KCG Semantic Checker.
- SCADE Test Model Coverage analysis with integration coverage criteria (see Section 6.4.4.2) must confirm that 100% of the SCADE Suite components control and data coupling structures are exercised by the requirement-based test cases and procedures.

6.4.4.3 Verification of data and control coupling between model and external environment

This activity is performed in the traditional way via a combination of design and code reviews and requirement-based integration testing.

6.4.5 Verification of additional code untraceable to source code (MB.A-7#9)

This activity is required for level A software only. Source to object code traceability analysis must address the following issues:

- Identify object code that is not ‘directly traceable’ to the source code
- Perform additional verification of this untraceable object code (if any)

Source code to object code traceability analysis must confirm that the target C compiler does not generate additional code that cannot be traced to the source code, based on a representative sample of C code defined by the coding standard (see [DO-248C], DP #12).

SCADE Suite CVK provides a representative sample of KCG generated code that may be used for this analysis.

6.4.6 Verification of simulation cases, procedures and results (MB.A-7#10, #11 and #12)

Objectives MB.A-7#10, #11 and #12 are not applicable in the context of the verification process described in this paper. No verification credit is claimed from simulation to achieve the objectives of Table MB.A-6 (EOC verification).
6.5 Summary of Verification of Verification

Table 6.5 summarizes verification objectives and methods for the verification of verification process results.

Table 6.5: DO-331 Table MB.A-7 Objectives Achievement

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Activity Ref</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Test procedures are correct.</td>
<td>6.4.5</td>
<td>Peer review of SCADE test procedures (see Note 1)</td>
</tr>
<tr>
<td>2 Test results are correct and discrepancies are explained.</td>
<td>6.4.5</td>
<td>Analysis of test report generated by user target testing environment</td>
</tr>
<tr>
<td>3 Test coverage of high-level requirements is achieved.</td>
<td>6.4.4.1 MB.6.8.2.a</td>
<td>Peer review of HLRs/SCADE test cases and procedures traceability matrices generated by SCADE LifeCycle RM Gateway</td>
</tr>
<tr>
<td>4 Test coverage of low-level requirements is achieved.</td>
<td>6.4.4.1 MB.6.7</td>
<td>Analysis of SCADE Suite model coverage with SCADE Test Model Coverage</td>
</tr>
<tr>
<td>5 Test coverage of software structure (modified condition/decision coverage) is achieved.</td>
<td>6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3 MB.6.8.2.b</td>
<td>Analysis of SCADE Suite model/code coverage with SCADE Test Model Coverage (modified condition/decision coverage)</td>
</tr>
<tr>
<td>6 Test coverage of software structure (decision coverage) is achieved.</td>
<td>6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3 MB.6.8.2.b</td>
<td>Analysis of SCADE Suite model/code coverage with SCADE Test Model Coverage (decision coverage)</td>
</tr>
<tr>
<td>7 Test coverage of software structure (statement coverage) is achieved.</td>
<td>6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3 MB.6.8.2.b</td>
<td>Analysis of SCADE Suite model/code coverage with SCADE Test Model Coverage (decision coverage)</td>
</tr>
<tr>
<td>8 Test coverage of software structure (data coupling and control coupling) is achieved.</td>
<td>6.4.4.2.c 6.4.4.2.d 6.4.4.3 MB.6.8.2.b</td>
<td>Analysis of SCADE Suite model control and data coupling with SCADE Test Model Coverage</td>
</tr>
<tr>
<td>9 Verification of additional code, that cannot be traced to Source Code, is achieved</td>
<td>6.4.4.2.b</td>
<td>Source code to object code traceability analysis (see Note 2)</td>
</tr>
<tr>
<td>MB 10 Simulation cases are correct (see Item 2)</td>
<td>MB.6.8.3.2</td>
<td>N/A (see Note 3)</td>
</tr>
<tr>
<td>MB 11 Simulation procedures are correct (see Item 2)</td>
<td>MB.6.8.3.2</td>
<td>N/A (see Note 3)</td>
</tr>
</tbody>
</table>
Table 6.5: DO-331 Table MB.A-7 Objectives Achievement (Continued)

<table>
<thead>
<tr>
<th>Objective Description</th>
<th>Activity Ref</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB 12: Simulation results are correct and discrepancies explained (see Item 2)</td>
<td>MB.6.8.3.2</td>
<td>N/A (see Note 3)</td>
</tr>
</tbody>
</table>

**Note 1**: Both simulation cases and test cases are factorized with the support of SCADE Test Target Execution. In this context, the qualification of SCADE Test Target Execution (as DO-330/TQL-5) removes the need for reviewing target test procedures if SCADE simulation cases and procedures have already been reviewed during SCADE Suite design verification.

**Note 2**: Support of CVK sample may be considered for this activity.

**Note 3**: No verification credit is claimed from simulation to achieve the objectives of Table MB.A-6.
Appendixes and Index
## References

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[AC 20-115C]</td>
<td>Advisory Circular “Airborne Software Assurance”, Federal Aviation Administration, 07/19/2013</td>
</tr>
<tr>
<td>Reference</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
# B Acronyms and Glossary

## ACRONYMS

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Advisory Circular</td>
</tr>
<tr>
<td>A/C</td>
<td>Aircraft</td>
</tr>
<tr>
<td>ANAC</td>
<td>Agência Nacional de Aviação Civil</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ARMAK</td>
<td>Air Register of Interstate Aviation Committee (Russia)</td>
</tr>
<tr>
<td>ARP</td>
<td>Aerospace Recommended Practices</td>
</tr>
<tr>
<td>BC</td>
<td>Basic Coverage</td>
</tr>
<tr>
<td>CAAC</td>
<td>Civil Aviation Administration of China</td>
</tr>
<tr>
<td>CAST</td>
<td>Certification Authorities Software Team</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The-Shelf</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CTP</td>
<td>Combined Testing Process</td>
</tr>
<tr>
<td>CVK</td>
<td>Compiler Verification Kit</td>
</tr>
<tr>
<td>DAL</td>
<td>Development Assurance Level</td>
</tr>
<tr>
<td>DC</td>
<td>Decision Coverage</td>
</tr>
<tr>
<td>DP</td>
<td>Discussion Paper</td>
</tr>
<tr>
<td>EASA</td>
<td>European Aviation Safety Agency</td>
</tr>
<tr>
<td>EOC</td>
<td>Executable Object Code</td>
</tr>
<tr>
<td>EUROCAE</td>
<td>European Organization for Civil Aviation Equipment</td>
</tr>
<tr>
<td>FAA</td>
<td>Federal Aviation Administration</td>
</tr>
<tr>
<td>FHA</td>
<td>Functional Hazard Analysis</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>HLR</td>
<td>High-level Requirement</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>IIR</td>
<td>Infinite Impulse Response</td>
</tr>
<tr>
<td>KCG</td>
<td>Qualified Code Generator</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>LLR</td>
<td>Low-level Requirement</td>
</tr>
<tr>
<td>MB</td>
<td>Model-Based</td>
</tr>
<tr>
<td>MBD</td>
<td>Model-Based Development</td>
</tr>
<tr>
<td>MC/DC</td>
<td>Modified Condition/Decision Coverage</td>
</tr>
<tr>
<td>N/A</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>N.B.</td>
<td>Nota Bene</td>
</tr>
<tr>
<td>OOT</td>
<td>Object-Oriented Technology</td>
</tr>
<tr>
<td>PDI</td>
<td>Parameter Data Item</td>
</tr>
<tr>
<td>PSSA</td>
<td>Preliminary System Safety Assessment</td>
</tr>
<tr>
<td>RM</td>
<td>Requirements Management</td>
</tr>
<tr>
<td>ROI</td>
<td>Return On Investment</td>
</tr>
<tr>
<td>RT</td>
<td>Related Techniques</td>
</tr>
<tr>
<td>RTCA</td>
<td>Radio Technical Commission for Aeronautics, RTCA, Inc.</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating System</td>
</tr>
<tr>
<td>SAE</td>
<td>Society of Automotive Engineers</td>
</tr>
<tr>
<td>SCADE</td>
<td>Safety Critical Application Development Environment</td>
</tr>
<tr>
<td>SCCI</td>
<td>Source Code Control Interface</td>
</tr>
<tr>
<td>SDL</td>
<td>Specification and Description Language</td>
</tr>
<tr>
<td>SQA</td>
<td>Software Quality Assurance</td>
</tr>
<tr>
<td>SRATS</td>
<td>System requirements allocated to software</td>
</tr>
<tr>
<td>SSA</td>
<td>System Safety Assessment</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>TAS</td>
<td>Tool Accomplishment Summary</td>
</tr>
<tr>
<td>TCI</td>
<td>Tool Configuration Index</td>
</tr>
<tr>
<td>TOR</td>
<td>Tool Operational Requirements</td>
</tr>
<tr>
<td>TORD</td>
<td>Tool Operational Requirements Data</td>
</tr>
<tr>
<td>TQL</td>
<td>Tool Qualification Level</td>
</tr>
<tr>
<td>TQP</td>
<td>Tool Qualification Plan</td>
</tr>
<tr>
<td>TSO</td>
<td>Timing and Stack Optimizer</td>
</tr>
<tr>
<td>TSV</td>
<td>Timing and Stack Verifiers</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst Case Execution Time</td>
</tr>
</tbody>
</table>
GLOSSARY

Extracts from [DO-178C].

**Certification**
Legal recognition by the certification authority that a product, service, organization, or a person complies with the requirements. Such certification comprises the activity of technically checking the product, service, organization, or person, and the formal recognition of compliance with the applicable requirements by issue of a certificate, license, approval, or other documents as required by national laws and procedures. In particular, certification of a product involves: (a) the process of assessing the design of a product to ensure that it complies with a set of standards applicable to that type of product so as to demonstrate an acceptable level of safety; (b) the process of assessing an individual product to ensure that it conforms with the certified type design; (c) the issuance of a certificate required by national laws to declare that compliance or conformity was found with standards in accordance with items (a) or (b) above.

**Certification credit**
Acceptance by the certification authority that a process, product, or demonstration satisfies a certification requirement.

**Condition**
A Boolean expression containing no Boolean operators except for the unary operator (NOT).

**Coverage analysis**
The process of determining the degree to which a proposed software verification process activity satisfies its objective.

**Data coupling**
The dependence of a software component on data not exclusively under the control of that software component.

**Deactivated code**
Executable object code (or data) that, by design, is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component; or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options. [...]

**Dead code**
Executable object code (or data) which exists as a result of a software development error but cannot be executed (code) or used (data) in an operational configuration of the target computer environment. It is not traceable to a system or software requirement. [An exception is embedded identifiers.]

**Decision**
A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition.

**Error**
With respect to software, a mistake in requirements, design, or code.

**Extraneous code**
Code (or data) that is not traceable to any system or software requirement. An example of extraneous code is legacy code that was incorrectly retained although its requirements and test cases were removed. Another example of extraneous code is dead code.

**Failure**
The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered.

**Fault**
A manifestation of an error in software. A fault, if it occurs, may cause a failure.
Fault tolerance
The built-in capability of a system to provide continued correct execution in the presence of a limited number of hardware or software faults.

Formal methods
Descriptive notations and analytical methods used to construct, develop, and reason about mathematical models of system behavior. A formal method is a formal analysis carried out on a formal model.

Hardware/software integration
The process of combining the software into the target computer.

High-level requirements
Software requirements developed from analysis of system requirements, safety-related requirements, and system architecture.

Host computer
The computer on which the software is developed.

Independence
Separation of responsibilities, which ensures the accomplishment of objective evaluation. (1) For software verification process activities, independence is achieved when the verification activity is performed by a person(s) other than the developer of the item being verified, and a tool(s) may be used to achieve an equivalence to the human verification activity. (2) For the software quality assurance process, independence also includes the authority to ensure corrective action.

Integral process
A process which assists the software development, processes and other integral processes and, therefore, remains active throughout the software life cycle. The integral processes are the software verification process, the software quality assurance process, the software configuration management process, and the certification liaison process.

Low-level requirements
Software requirements derived from high-level requirements, derived requirements, and design constraints from which source code can be directly implemented without further information.

Modified Condition/Decision Coverage
Every point of entry and exit in the program was invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision was shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.

Parameter Data Item
A set of data that, when in the form of a Parameter Data Item File, influence the behavior of the software without modifying the Executable Object Code and that is managed as a separate configuration item. Examples include databases and configuration tables.

Robustness
The extent to which software can continue to operate correctly despite abnormal inputs and conditions.

Standard
A rule or basis of comparison used to provide both guidance in and assessment of the performance of a given activity or the content of a specified data item.

Test case
A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Test Procedure
Detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases.
Tool qualification
The process necessary to obtain certification credit for a software tool within the context of a specific airborne system.

Traceability
An association between items, such as between process outputs, between an output and its originating process, or between a requirement and its implementation.

Validation
The process of determining that the requirements are the correct requirements and that they are complete. The system life cycle process may use software requirements and derived requirements in system validation.

Verification
The evaluation of the results of a process to ensure correctness and consistency with respect to the inputs and standards provided to that process.
C DO-178C Qualification of SCADE Suite KCG

C-1 What Does Qualification Mean and Imply?

Qualification of a tool is needed when processes are eliminated, reduced, or automated by the use of the tool, without its output being otherwise verified. The entire qualification process is described in Section 12.2 of DO-178C and in the full contents of DO-330.

Within DO-178C, Criteria 1 tools are those whose output is part of the embedded software; thus, they can introduce errors in the embedded software. That is why SCADE Suite KCG is classified as a Criteria 1 tool. The way to achieve the qualification of a development tool is as follows:

- Using Table 12-1 of DO-178C, the Tool Qualification Level is identified. To be able to use KCG for generating source code for level A application software without verification of its output, KCG Tool Qualification Level is TQL-1, the most rigorous software level.
- DO-330 defines the activities, guidance, and life cycle data required by Tool Qualification Levels.

C-2 Development of SCADE Suite KCG

The SCADE Suite KCG code generator is developed as a TQL-1 tool to be able to use KCG for generating source code for level A application software without verification of its output. These objectives are described in the following documents, audited by Certification Authorities on a number of past projects:

- **Compliance Analysis**: Presents KCG compliance with DO-330 objectives at TQL-1
- **Tool Qualification Plan (TQP)**: Presents all provisions taken for KCG code generator qualification and references other project plans
- **Tool Operational Requirements (TOR)**: Describes KCG functionality and usage. It matches the Developer-TOR defined in DO-330.
- **Scade Language Reference Manual**: Contains the Scade language definition
- **Tool Accomplishment Summary (TAS)**: Gives compliance status with TQP, conditions of use, list of unresolved defects and tool limitations
- **Software Installation Procedure (SIP)**: Contains detailed instructions for installing KCG
- **Tool Configuration Index (TCI)**: Presents tool version and configuration
- **Tool Life Cycle Environment Configuration Index (TECI)**: Presents the software environment used for tool certification
C-3  SCADE Suite KCG Life-Cycle Documentation

Table C.1 lists the documents that are delivered to SCADE Suite users for the qualification of SCADE Suite KCG.

Table C.1: Documents delivered for KCG qualification audit by Certification Authorities

<table>
<thead>
<tr>
<th>Data</th>
<th>DO-330 Ref.</th>
<th>Certification Kit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool Qualification Plan</td>
<td>10.1.2</td>
<td>SCADE Suite KCG Tool Qualification Plan</td>
</tr>
<tr>
<td>Tool Operational Requirements</td>
<td>10.3.1</td>
<td>Software requirements data of SCADE Suite KCG</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scade 6 Language Reference Manual</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCADE Suite KCG Tool Installation Procedure</td>
</tr>
<tr>
<td>Tool Accomplishment Summary</td>
<td>10.1.15</td>
<td>SCADE Suite KCG Tool Accomplishment Summary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCADE Suite KCG Compliance Analysis to DO-330 (all levels)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCADE Suite KCG Compliance Analysis to DO-178B (all levels)</td>
</tr>
<tr>
<td>Tool Configuration Index</td>
<td>10.1.11</td>
<td>SCADE Suite KCG Tool Configuration Index</td>
</tr>
<tr>
<td>Tool Life Cycle Environment Configuration Index</td>
<td>10.1.10</td>
<td>SCADE Suite KCG Tool Life Cycle Environment Configuration Index</td>
</tr>
</tbody>
</table>

All other lifecycle data (e.g., plans and standards, design data, source code, or test cases) are available and can be audited by the Certification Authorities at Esterel Technologies.
**SCADE Suite Compiler Verification Kit (CVK)**

---

**D-1 CVK Product Overview**

**WHAT SCADE SUITE CVK IS**

While SCADE Suite KCG qualification ensures that source code conforms to LLRs developed with SCADE Suite, CVK is a test suite that can be used to verify that the type of code generated by SCADE Suite KCG is correctly compiled/executed with a given cross-compiler on a given target.

CVK can be used for the following purposes:

- to support early verification of the correctness and consistency between the development tool chain and the target platform
- to address the verification of target

**WHAT SCADE SUITE CVK IS NOT**

1. CVK is NOT a validation suite of the C compiler. Such validation suites are generally available on the market. They rely on running large numbers of test cases covering all programming language constructs, the right amount of combinations, and various compiling options. It is expected that the applicant requires evidence of this activity from the compiler provider (or other source).

2. CVK is NOT an executable software.

3. CVK is NOT a hardware test suite.

Since CVK is not a tool (it is a set of test cases and procedures), the concept of qualification is not relevant. Instead, CVK is verified with the same objectives as any other set of test cases and procedure, including review, requirements coverage analysis, and structural coverage analysis (MC/DC).

**ROLE OF SCADE SUITE CVK**

CVK is a test suite: it is part of verification means of the SCADE Suite KCG users. Figure D.1 shows the complementary roles of KCG and CVK in the verification of the development environment of the users.

---

*Figure D.1: Role of KCG and CVK in verification of user development environment*
1 **Objective MB.A-4.3**: Low-level requirements are compatible with target computer. CVK allows compatibility analysis of the cross-compiler and target regarding:

- Complexity of expressions
- Complexity of control structures
- Rounding to zero

2 **Objective MB.A-4.10**: Architecture is compatible with target computer. CVK allows compatibility analysis of the cross-compiler and target regarding:

- Complexity of data structure nesting
- Number of arguments in a function call

**SCADE SUITE CVK CONTENTS**

The CVK product is made of the following:

   - Installation and user instructions
   - Description of the underlying methodology
   - Scade models description
   - C sample description
   - Test cases and procedures description
   - Coverage matrices
   - C code complexity metrics description
2. The SCADE Suite-generated C sample to verify the C compiler.
3. A representative SCADE Sample covering the set of Scade language primitive operators and enabling the generation of C sample with KCG in your own environment.

4. Requirements-based test cases to exercise the Scade C sample with 100 percent MC/DC coverage [NASA-MCDC] for all KCG settings.

5. Automated test procedures for Windows platform.

**C SAMPLE CHARACTERISTICS**

The C sample is generated from a Scade models database by SCADE Suite KCG and it exhibits the following characteristics:

- It contains an exhaustive set of elementary C constructs that can ever be generated from a SCADE Suite model by the SCADE Suite KCG Code Generator.
- It contains a set of combinations of these elementary C constructs.

**D-2 CVK Representativity**

The source code generated by SCADE Suite KCG is a subset of C with several relevant safety properties in term of statements, data structures and control structures such as:

- No recursion or unbounded loop.
- No code with side effects (no a += b, no side effect in function calls).
- Communication between operators only goes through explicit data flows.
- No functions passed as arguments.
- No arithmetic on pointers.
- No pointer on function.
- No jump statement such as “goto” or “continue”
• Memory allocation is fully static (no dynamic memory allocation).
• Expressions are explicitly parenthesized.
• There are no implicit conversions.

CVK contains a representative sample of the generated code. This sample covers a subset of elementary C constructs as well as deeply nested constructs identified from C code complexity metrics.

The C code complexity metrics provided by CVK are relevant in the context of C compiler verification. These metrics, selected by analyzing compiler limits defined in C standards and cross-compilers documentation, address complexity both in depth and in width.

Each complexity metric has a limit defined by CVK to cover a certain degree of complexity. Therefore, CVK users must check that the complexity of the code generated by KCG from their SCADE Suite application is in the scope of the limits covered by CVK. SCADE Suite KCG provides most values for these metrics in a dedicated generated file. Some other metrics are computed by scripts.

This approach addresses the concerns expressed by certification authorities in [DO-330] (see FAQ D.8 Scenario 3, section 1) for compiler verification activities in the case of automatically generated code.

---

**D-3 Strategy for Developing SCADE Suite CVK**

Figure D.2 summarizes the strategy for developing and verifying CVK.

![Figure D.2: Strategy for developing and verifying CVK](image)

CVK is built in the following way:

1. Identify the C elementary constructs generated with KCG by analyzing the KCG software requirements (HLR and LLR). These C constructs are identified by a name and defined in terms of the C-ISO standard.
2. Define relevant complexity metrics for KCG-generated code by analyzing compilers limits defined in C standards and compilers documentation. These metrics address parameters such as the number of nested structures or the number of nesting levels of control structures.
3 Identify the combination of elementary C constructs generated by KCG that make sense in the compiler verification (in particular, focus on the risky events for a cross-compiler). These combinations are directly based on complexity metrics previously identified. Their usage limits and generation conditions are defined at this step.

4 Build the C sample:
   a A suite of Scade samples, covering all SCADE constructs, is built as material for code generation.
   b Each elementary C construct and their combination are generated from Scade samples root nodes with appropriate KCG options.
   c Coverage of the C subset (elementary C constructs and combination) by the C sample is required and verified.

5 Develop a test harness, exercising the C sample with a set of input vectors and verifying that the output vectors conform to the expected output vectors.

6 Tests execution on a host platform to verify:
   a Conformance of outputs to expected outputs.
   b MC/DC coverage at C code level.

7 Tests execution for each selected target platform to verify:
   a The adaptation to a specific cross-environment capabilities of CVK (portability).
   b The correctness of effective output vectors on the platform.

**D-4 Use of SCADE Suite CVK**

CVK is used as follows (Figure D.3):

- The CVK User’s Manual [CVK_UM] is an appendix of the customer’s verification plan, more precisely in the qualification plan of the user’s development environment.
- The CVK test suite is instantiated for the customer’s verification process, more precisely in the qualification process of one’s development environment, for the verification of the compiler. Users must verify that the complexity of their model (depth of expressions, data structures, and call tree) is lower than the one of the model in CVK. Otherwise, they shall either upgrade CVK accordingly or decompose the SCADE model.

**Figure D.3:** Use of CVK items in user processes

**Figure D.4** details the role of CVK items (highlighted by shadowed boxes) in the verification of the compiler:

- The C sample is regenerated by KCG from the SCADE sample, with specified KCG options and is compared to the provided Reference C sample.
• From the C sample, the C compiler/linker generates an executable. Note that the C sample is always taken from the delivered reference sample, not from the regenerated C sample.

• The executable reads input vectors (from its static memory) and computes output vectors. It compares the actual output vectors to reference vectors (from its static memory). Comparison is performed directly in the C test harness. The C primitive "==" is used for boolean, integer and character data and a specific C function is used for floating point comparison with tolerance. Unit tests of these comparison C functions are provided in the CVK test suite to ensure that the C compiler compiles correctly these functions. The reference vectors were developed and verified when developing CVK, and are based on the requirements (i.e., semantics of Scade model).

The cross compiler/linker has to be run with the same options as for the manual code and as for the rest of the KCG-generated code. If there is a discrepancy (beyond a relative tolerance parameter, named epsilon for floating point data) between collected and reference results, an analysis has to be conducted to find the origin of the difference. If it is an error in the use or contents of CVK (e.g., error in adapting the compiling procedure), this has to be fixed. If it is due to an error in the compiler, then the usage of this compiler should be seriously reconsidered.

To be able to share the verification of Source to Object code traceability analysis between the KCG-generated code and manual code, it is recommended to use the same environment (cross-compiler/linker with same options) for the manual code and the KCG code.
Index

A
Accuracy 60
ANAC 5
Architecture 35
ARP 4754A
  overview 5

B
Basic Coverage 83
BC 83

C
C elementary constructs 103
C sample 102
C subset 102
CAAC 5
Causality 25
Clock 29
Coding process 42
Combined Testing Process 72
Compiler Verification Kit 101
Concurrency 26, 29
Control Engineering 22
Coverage 78, 80
Coverage analysis
  test coverage 14
  with Test Model Coverage 78
Coverage criteria
  structural coverage 14
Coverage resolution
  structural coverage 14
CPU 62
CVK 101
CVK test execution and MC/DC 102

D
Data typing 28
Dependency 26
Design process 35
Design Verifier 59
Development assurance levels 7
Development processes 9, 10
Discrete control 27
DO-178C
  overview 5
  processes 9

E
EASA 5
Equations 24
EUROCAE 5

F
FAA 5
Filtering 37
Formal verification 59

H
High-level requirements 10, 12
HLR
  see High-level requirements

I
Initialization 26
Integral processes 9
Integration process 47
Interface 24

K
KCG 30, 64, 69
  acronym 95
DO-178C qualification 99
SCADE Suite and CVK 101

L
LLR
  see Low-level requirements
Local variables 24
Logic 38, 39
Low-level requirements 10, 12
  development in SCADE 37

M
MC/DC 14
Model 80
Model-based 30
Model-Level Coverage 80
Modified Condition/Decision Coverage
  see MC/DC
Modular 25
Multitasking 51

O
Operator 24

P
Parameter Data Item 70
Partitioning 66
Planning processes 9

Q
Qualification 69, 99

R
Regulation 37
Requirements Management Gateway 34
Requirements process 34
RTCA 5
RTOS 49
Index

S
SCADE State Machines 27
Scheduling 48
Simulation 56
Software architecture 10
Software Design Standards 12
Source code 12
Standards 70

T
TAS
    acronym 95
Task integration
    RTOS 49
Tasking 48
TCI
    acronym 95
Teamwork 52
Test procedures 78
Test results 78
Testing 13, 53
TORD
    acronym 95
Traceability 43

U
Unintended functions 79

V
Validation 53
Verification 53
Verification processes 11

W
WCET analysis 61
Contact Information

Submit questions to Technical Support at
scade-support@esterel-technologies.com

Contact one of our Sales representatives at
scade-sales@esterel-technologies.com

Direct general questions about Esterel Technologies to
scade-info@esterel-technologies.com

Discover the latest news on our products and technology at
http://www.esterel-technologies.com

Copyright © 2015 Esterel Technologies - A wholly-owned subsidiary of ANSYS, Inc. All rights reserved Esterel Technologies SAS. [SC-HB-D0178C-KG64]