Ansys

Methodology Handbook

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

Second Edition

CONTACTS

Legal Contact

ANSYS France S.A.S.
15 place Georges Pompidou
78180 Montigny-le-Bretonneux FRANCE
Phone: +33 1 30 68 61 60
Fax: +33 1 30 68 61 61

Technical Support

ANSYS France S.A.S.
Parc Avenue, 9 rue Michel Labrousse
31100 Toulouse FRANCE
Phone: +33 5 34 60 90 50
Fax: +33 5 34 60 90 41

Submit questions to SCADE Products Technical Support at scade-support@ansys.com.

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

Direct general questions about SCADE products to scade-info@ansys.com.

Discover latest news on our products at www.ansys.com/products/embedded-software.

Shipping date: April 2021 Revision:SC-HB-DO178C-KCG66 - DOC/rev/74520-04 SecondEd

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 guidance 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 the 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

Table of Contents

Table of Contents

Table of Contents

List of Figures

List of Figures

List of Figures

List of Tables

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 control laws as described above and to automatically generate the code from this model using a code generator qualified with respect to [DO-330]. This method 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 software engineers by supporting the accurate definition of the software requirements and by providing efficient automatic code generation of software having the qualities 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 guidance (see [DO-178C] and [DO-331]). It also presents a quick overview of an 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/ introduces the regulatory guidance of ARP 4754A, DO-178C, and DO-331 used when developing embedded

aeronautics systems and software. It also addresses Tool Qualification considerations according to the DO-330 guidance.

Section 3/ presents an overview of SCADE Suite methodology and tools, including how our solutions achieve 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.
- Early identification of design errors makes it possible to fix them in requirements/design phase rather than in testing or integration phases.
- Qualified code generation not only saves writing the code by hand, but also the cost of verifying it.
- Automation of verification activities relies on a set of SCADE testing and life-cycle management tools qualified as verification tools.

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

Section 5/ and Section 6/ present the verification activities to be performed when using SCADE tools. 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.

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 qualification process of SCADE Suite KCG code generator.

Appendix D/ details the Compiler Verification Kit (SCADE Suite CVK).

The concepts and methodology described in this document are applicable starting from the following product configuration (and onwards):

- SCADE Suite 19.2 with SCADE Suite KCG 6.6
- SCADE Test Model Coverage 19.2 for SCADE Suite
- SCADE Test Environment for Host and SCADE Test Target Execution 19.2
- SCADE LifeCycle 19.2

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 the 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 generally have 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 retargetable 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 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 Guidance

2.1.1 Introduction

The certification authorities[1] require from the aeronautics industry means of compliance to safety standards for any safety-critical software that may be used on a commercial aircraft. [ARP 4754A] and [DO-178C][2] provide guidance used both by the companies developing airborne equipment and by the certification authorities.

2.1.2 ARP 4754A/ED-79

The Aerospace Recommended Practice ARP 4754A is the Guidelines For Development Of Civil Aircraft and Systems. It is published by 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 released 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-HB], 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 current 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 AR MAK will approve all commercial software-based aerospace systems. The new document is called DO-178C/ED-12C and it was completed in November 2011 and approved by RTCA in December 2011. This

1. For example, the United States Federal Aviation Administration (FAA), the European Aviation Safety Agency (EASA), AR MAK (Russia), Transport Canada, Brazil's Agência Nacional de Aviação Civil (ANAC), Civil Aviation Administration of China (CAAC).
2. This section contains many quotations from DO-178C standard guidance. Some figures are directly reproduced from the standard.

document was approved in July 2013 by the FAA (see [AC 20-115C]) and in September 2013 by the EASA (see [AMC 20-115]), making it recognized as an acceptable "*means of compliance with the applicable airworthiness regulations for the software aspects of airborne systems*".

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

The standard guidance specifies:

- 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, ARP 4761, and DO-178C

ARP 4754A and DO-178C provide complementary guidance:

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

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

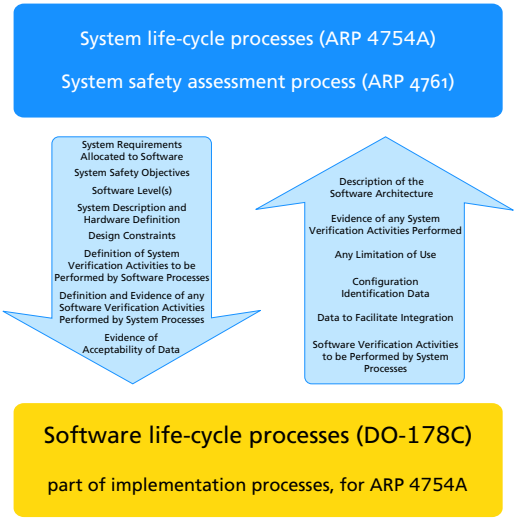


Figure 2.1: Relation between ARP 4754A, ARP 4761, and DO-178C processes

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

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.*

h *Evidence of software verification activities performed by the system life cycle processes, if any.*"

On the other hand, DO-178C provides the information flow from Software Processes to System Processes (see §2.2.2 in [DO-178C]):

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 *Problems or documentation changes, 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 below.

Table 2.1: Top-Level function DAL assignment

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

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.

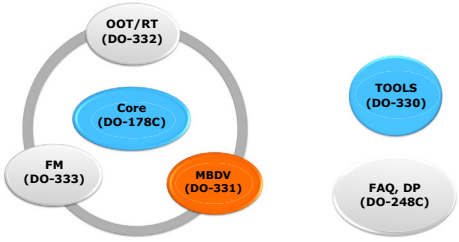


Figure 2.2: DO-178C documents structure

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 Section 2.1.7 and Section 2.1.8 for further information on DO-178C objectives and processes).

This core document is completed by supplements to be considered, depending on the techniques used for the production of 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.

DO-178C DOCUMENTS COMMON STRUCTURE

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

- DO-248C "*Supporting Information for DO-178C*" addresses the questions of 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 are achieved. The DO-178C authors acknowledged that objectives are more essential and stable than specific procedures. The ways of achieving an objective may vary between companies, 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.

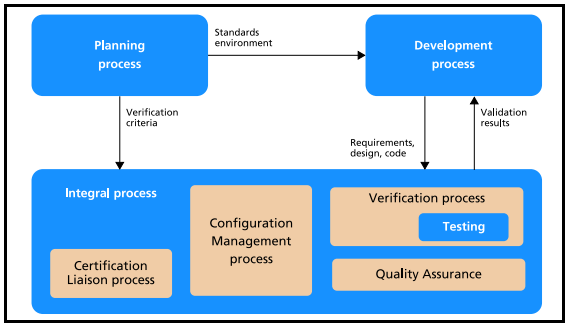


Figure 2.3: DO-178C life-cycle processes structure

In the remainder of this document, we focus on the development and verification processes.

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

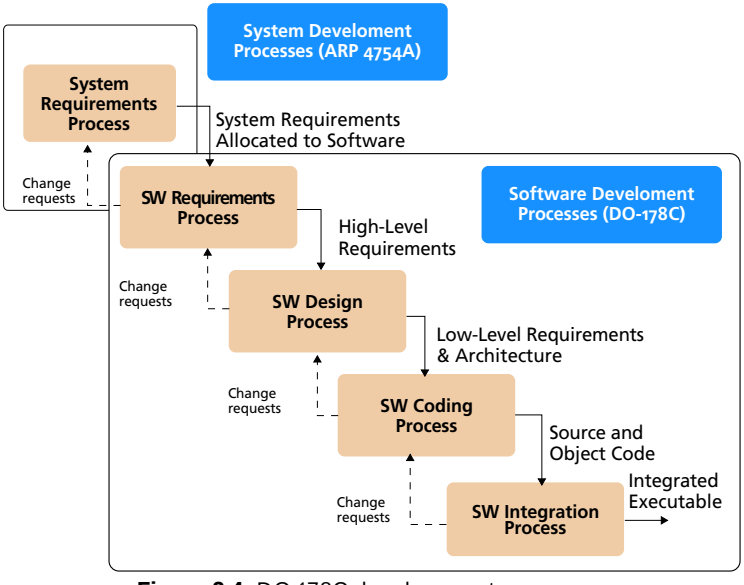


Figure 2.4: DO-178C development processes

The 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 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 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 processes.*" 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 HLR

The objective of reviews and analyses is to confirm that the HLR 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 LLR 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 LLR and architecture satisfy the following:

- Compliance with high-level requirements: the software LLR satisfy the software HLR
- 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
- Traceability: the objective is to ensure that all HLR were taken into account in the development of the LLR
- 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 LLR and the software architecture, and compliance with the Software Code Standards. The reviews should include:

- Compliance with low-level requirements: the source code is accurate and complete with respect to the software LLR; 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 LLR
- 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.

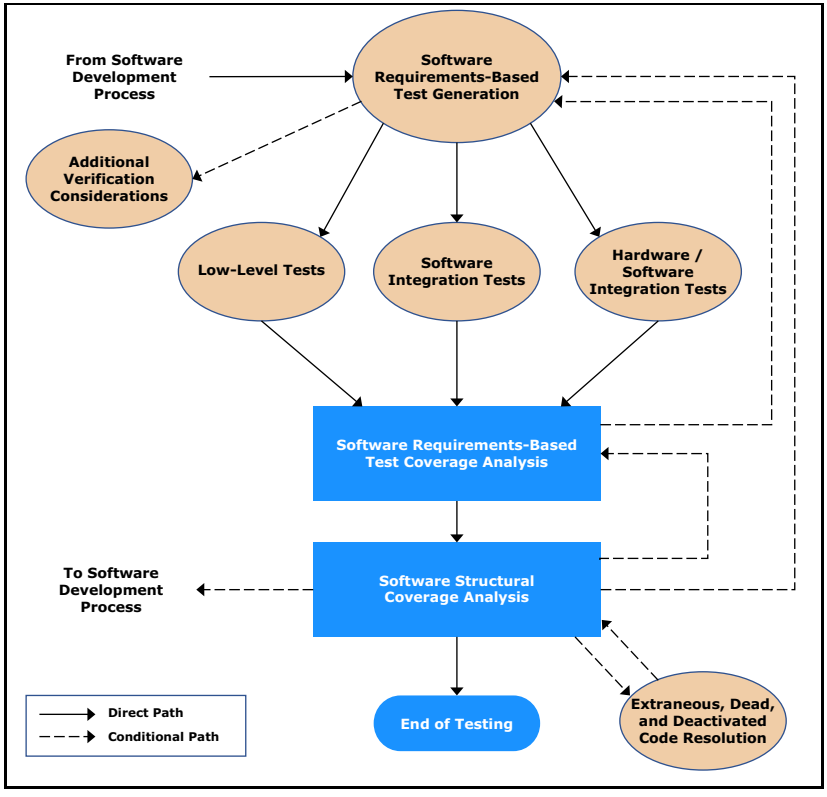


Figure 2.5: DO-178C testing process

There are three types of testing activities:

- Low-level testing: to verify that each software component complies with its LLR
- 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 HLR and LLR.
2. Structural coverage analysis determines which code structures including interfaces between components, are exercised by requirements-based test procedures. Its purposes are:
   - Ensures all code structures, including interfaces, was executed at least once
   - Detects untested functions which could be unintentional
   - 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 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.
- 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 C: Statement Coverage is required, meaning every statement in the source code was exercised.

For instance, the following fragment requires four test cases for Level A, 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 satisfying MC/DC

Case	A	B	C	Outcome
1	FALSE	FALSE	ANY	FALSE
2	TRUE	ANY	ANY	TRUE
3	FALSE	TRUE	TRUE	TRUE
4	FALSE	TRUE	FALSE	FALSE

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 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 examples of Model usage in the context of industrial projects that illustrate different strategies for Model-Based Development (MBD).

Table 2.3: Model usage examples (DO-331 Table MB.1-1)

Process generating life-cycle data	MB Example 1	MB Example 2	MB Example 3	MB Example 4	MB Example 5
System Requirements and System Design Processes	Requirements allocated to software	Requirements from which the model is developed	Requirements from which the model is developed	Requirements from which the model is developed	Requirements from which the model is developed
Software Requirements and Software Design Processes	Requirements from which the model is developed	Specification Model	Specification Model	Design model	Design model
	Design model	Design model	Textual description		
Software Coding Processes	Source code	Source code	Source code	Source code	Source code

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 follows:

- Software requirements are usually textual requirements supplemented by pictures, when appropriate, that are derived from System Requirement Allocated to Software (SRATS)
- A design model is developed for LLR 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 using model-based development. 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 representative of the

target code. Simulation is different from testing which is the execution of the "real" Executable Object Code (EOC) on target.

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

- Compliance with HLR for models containing LLR
- Accuracy and consistency
- Verifiability
- Algorithm aspects

On the other hand, 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 LLR. In particular, the following objectives are considered in addition to the existing objectives described in Section 2.3.3:

- "*Simulation cases are correct*" (Table MB.A-4 objective MB14)
- "*Simulation procedures are correct*" (Table MB.A-4 objective MB15)
- "*Simulation results are correct and discrepancies explained*" (Table MB.A-4 objective MB16)

2.4.5 Model coverage analysis for design models

Model coverage analysis concerns the 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 from structural coverage analysis. Both model coverage and structural code coverage shall be achieved to satisfy objectives of DO-331 Table MB.A-7. As stated in DO-331 FAQ-11, the implication of model level coverage to the generated code level allows to satisfy structural code coverage objectives.

DO-331 objective MB.A-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 stated in DO-331 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:

- The modeling notations range from non-formal (*e.g.*, UML/SysML) to formal (*e.g.*, Scade, event-B)
- 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.*, UML)

Even if it is not possible to impose specific detailed model coverage criteria due to various modeling notations, [DO-331], Table MB.6-1 provides an example of criteria that are relevant to assess model coverage according to the objectives defined in [DO-178C], see §6.4.2.1 and §6.4.2.2.

General principles such as coverage of all characteristics of the requirements (from which the design model is developed) functionality, coverage of equivalence classes and boundary/singular values for numeric data, and coverage of all derived requirements are highlighted in the example from [DO-331], Table MB.A-6.

The applicant may use any alternative coverage criteria provided these criteria comply with the objectives defined in [DO-178C], see § 6.4.2.1 and §6.4.2.2.

These criteria should be defined in the Software Verification Plan (SVP) of the applicant.

2.5 DO-330 Software Tools Qualification Considerations

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

The 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 the airborne domain, 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.

Software Level	Criteria		
	1	2	3
A	TQL-1	TQL-4	TQL-5

Software Level	Criteria		
	1	2	3
B	TQL-2	TQL-4	TQL-5
C	TQL-3	TQL-5	TQL-5
D	TQL-3	TQL-5	TQL-5

2.5.4 Tool Stakeholders

One major improvement of 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 Architect for the analysis and design of software architecture in synchronization with software subsystem design;
- SCADE Suite for the design of embedded control applications;
- SCADE Display for the design of embedded displays;
- 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 critical applications.

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

SCADE Suite addresses the applicative part of software as illustrated in Figure 3.1. This is usually the most complex and changeable aspect of software. It typically represents 60 percent to 90 percent of the embedded software.

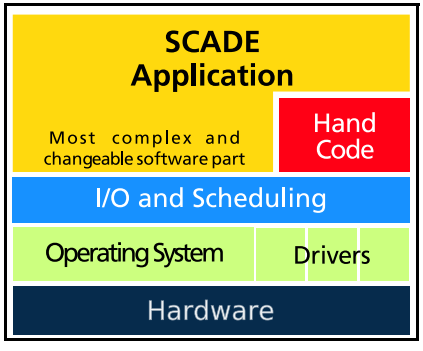


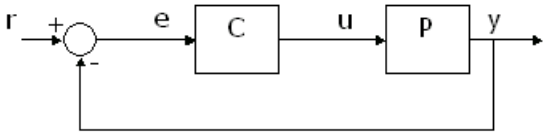
Figure 3.1: Applicative part of software

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 3.2.



$$X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n} \quad \text{(bilateral z transform)}$$

Figure 3.2: Control engineering view of a Controller

- Software engineers describe their programs in terms of tasks, flow charts, and algorithms, as shown below in Figure 3.3.

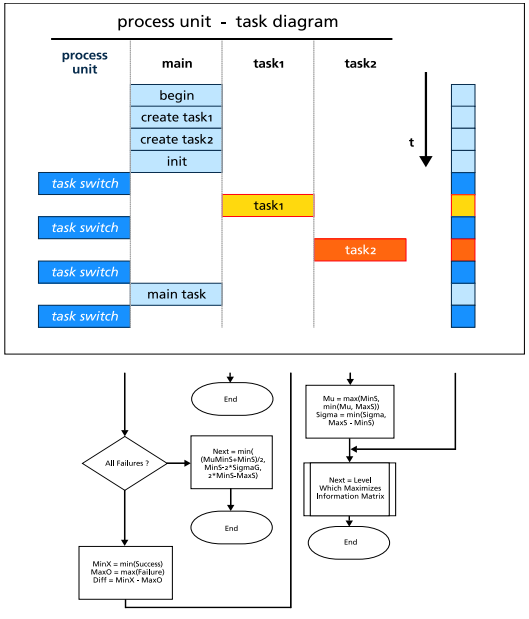


Figure 3.3: A software engineering view

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

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 sampling 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. The product allows for automatic generation of C/Ada code from the above formalism.

3.2.1.2 Scade operator

The basic Scade building block is called an operator. It is either a pre-defined operator (*e.g.*, +, delay) or a user-defined operator that decomposes itself using other operators. This allows to build complex applications in a structured way. An operator can be represented graphically (see Figure 3.4) or textually (see Table 3.1).

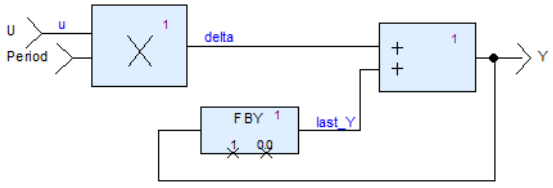


Figure 3.4: Graphical notation for an integrator operator

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

Table 3.1: Components of Scade functional modules: operators

Component	Textual Notation for an Integrator Operator [1]	Graphical Notation
Formal interface	`node IntegrFwd(U: real ; hidden Period: real)` `  returns (Y: real) ;`	Arrows and rectangles

Local variables declarations	`var`
	`delta : real ;`
	`last_Y : real;`
		Named wires

Equations	`delta = u * Period ;`
	`Y = delta + last_Y ;`
	`last_Y = fby(Y , 1 , 0.0) ;`
		Network of operator calls

1. In the Scade language, 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 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.

In SCADE Suite, a user-friendly editing mode supports 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 the Scade language, continuous control is graphically specified using data flow diagrams, such as the one illustrated in Figure 3.5 below.

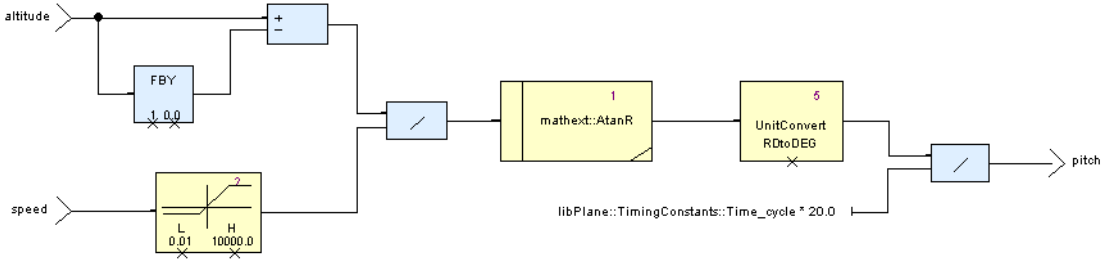


Figure 3.5: Sample of model data flows from a Flight Control system

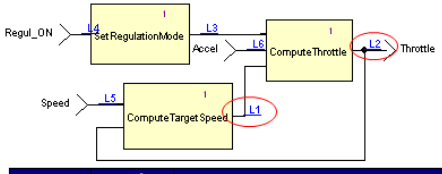
Operators compute mathematical functions, filters, and delays, while arrows denote data flowing between operator instances. Operator instances that have no functional dependency are computed concurrently. Flows may carry numeric, Boolean, enumeration, or structured values used or produced by operators.

Operators are fully hierarchical: operators at a description level can themselves be composed of smaller operators interconnected by local flows. In models, 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.

The Scade language 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 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 3.6 below, where the “Throttle” output depends on itself via the ComputeTargetSpeed and ComputeThrottle operators. With SCADE Suite Checker, semantic checks[3] detect this error and signal that this output has a recursive definition.



Category	Code	Message
Semantic Error	ERR_400	**Causality error at** Package2::Operator1/ L1= the definition of flow _L1 depends on flow _L2 ; (Package2::Operator1/ L2=) the definition of flow _L2 depends on flow _L1 ;

Figure 3.6: Detection of a causality problem

3. SCADE Suite Checker is provided with SCADE Suite for running syntactic and semantic checks during software modeling.

As shown in Figure 3.7, 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.

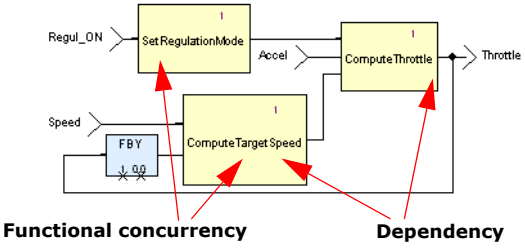


Figure 3.7: Functional expression of concurrency

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 the Scade language, 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 language is related to the initialization of flows. In the absence of explicit initialization for instance by using the -> (Init) operator, SCADE Suite semantic check emits errors, as illustrated in Figure 3.8 for a counter model.

Category	Code	Message
Semantic Error	ERR_300	**Initialization error at** Package3::Operator4/_L2= The operator "pre" expects a well-initialized argument (Package3::Operator4/_L2=) The pre operator caused a delay

Figure 3.8: Detection of a flow initialization problem

As shown in Figure 3.9, 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 critical software.

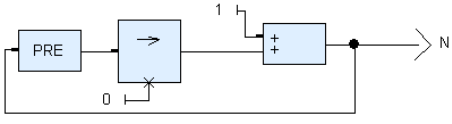


Figure 3.9: Initialization of flows

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 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 3.10.

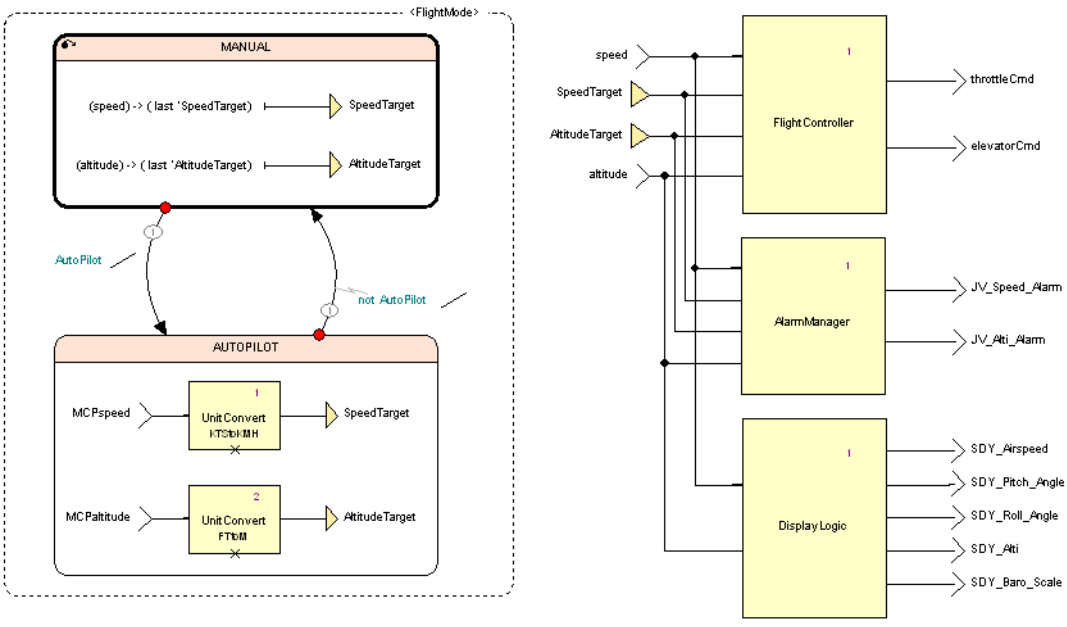


Figure 3.10: Mixed data and control flows from Flight Control

3.2.1.6 SCADE data typing

The Scade language is strongly typed.

The following data types are supported;

- Predefined types: Boolean, Integer (int 8, uint8, int16 uint16, int 32, uint32, int64, uint64), Real (float 32, float 64), Enumeration, Character.
- Structured types:
  - Structures make it possible to group data of different types. Example:

```
Ts = [x: int, y: real];
```

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

```
tab = 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 SCADE Suite semantic checks.

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 3.11 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 functions are *blind to environment changes*.[4] 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.

The external environment shall ensure that the cyclic function of the whole system is blind to environment changes.

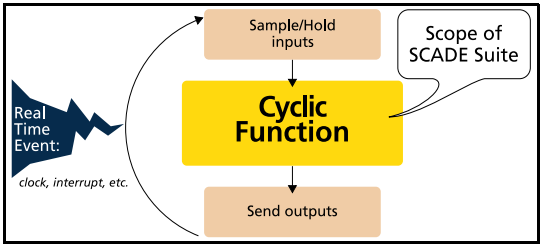


Figure 3.11: Cycle-based execution model of SCADE

CONCEPT OF CYCLE IN SCADE SUITE

In a Scade specification, each operator and flow has a so-called clock (the event triggering its cycles) and all operators that do not exhibit data flow dependencies act concurrently (see Figure 3.7). Operators can all have the same clock, or they can have different clocks, which subdivide a master clock. At each of its clock cycle, an operator reads its inputs and generates its outputs. If an output of operator A is connected to an input of operator B, and A and B have the same cycle, the outputs of A are used by B in the same cycle, unless an explicit delay is added between A and B. This is the essence of the semantics of the Scade language.

State machines share the same notion of 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 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.

BENEFITS OF CYCLE-BASED COMPUTATION MODEL

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/Ada 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. There is no overhead for communication, which is internally implemented using well-controlled shared variables without any context switching.

3.2.3 SCADE Suite as a model-based development environment

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

4. It is still possible for interrupt service routines or other task to run, as long as they do not interfere with the cyclic function.

- SCADE Suite models are usually considered as design model. They mainly represent the software low-level requirements and software architecture. Such models rely on a formally defined notation.
- Models can be managed under configuration control.
- Documentation is automatically and directly generated from models: it is correct and up-to-date by construction.
- Syntactic and semantic checking can be performed to check that the models follow the rules of the Scade notation syntax and semantics.[5]
- Models can be exercised by simulation to verify dynamically their 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 models 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 any model and the target platform.
- Target compatibility with SCADE Suite KCG- generated code can also be verified on a representative code sample (using SCADE Suite Compiler Verification Kit) in particular to anticipate potential issues with the cross-compiler used to generate the target EOC.
- Code is automatically and directly generated from models with the KCG qualified Code Generator: the source code complies with the semantics of the input model.
- 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 at DO-330/ TQL-1 (see Appendix C/).

SCADE Suite applies these “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 design, between simulation and target testing, between module software design and code.
- Do things right at the outset: Detect errors in the early stages of a project.

BENEFITS OF DESIGN-VERIFY-GENERATE PRINCIPLE

SCADE Suite allows saving 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.

BENEFITS OF “DO THINGS ONCE” PRINCIPLE

SCADE Suite models formalize a significant part of the software architecture and design. The model is

5. In SCADE Suite, use SCADE Suite Checker.

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 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 modeling and safety benefits

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

- The Scade 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 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 Suite 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 language contains specific features oriented towards safety: strong typing, mandatory initialization of flows, and so on.
- SCADE Suite models are 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 language provides a simple and clean expression of concurrency at functional level (data flows express dependencies between operators). Within a model, this avoids the traditional problems of deadlocks and race conditions.
- SCADE Suite performs the complete verification of language syntactic and semantic rules, such as type and clock consistency, initialization of data flows, or causality in models.

4 / Software Development Activities with SCADE Suite

4.1 Overview of Software Development Activities

A typical SCADE Suite software development process is a combination of Model-Based Development flow with an integration step of generated code.

Figure 4.1 shows the software development processes and where SCADE Suite is used.

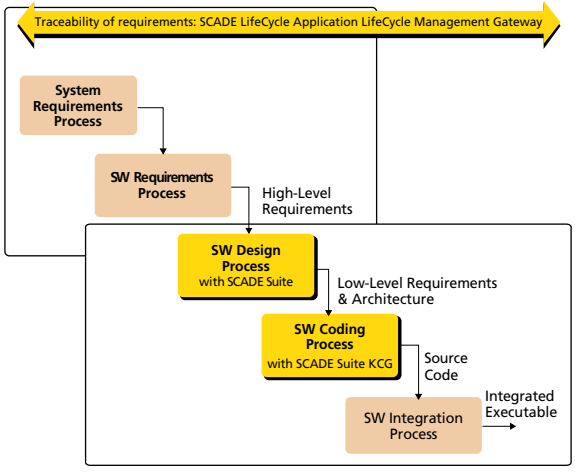


Figure 4.1: Software development processes with SCADE Suite

Some companies start using SCADE Suite to define control laws during the system definition phase.

The simulation of 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 models are extensively used in the software design process to develop major parts of the architecture and the low-level requirements. Such models are design models 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 HLR
- HLR and LLR (*i.e.*, SCADE Suite model)
- LLR and source code

The traceability process between HLR and SCADE Suite models can be easily supported by SCADE LifeCycle Application Lifecycle Management (ALM) Gateway as mentioned in Figure 4.1 above and as illustrated in Section 4.3.

The SCADE Suite KCG-generated code must be integrated with respect to integration constraints specified in [KCG-TOR].

4.2 Software Requirements Process

In DO-178C terminology, the inputs to the Software Requirements Process are the System Requirements allocated to Software (SRATS). The software requirements process produces the HLR. These HLR usually include functional, performance, interface, and safety-related requirements.

The logics requirements (logics HLR) are usually in textual form. SCADE Suite modeling capabilities can be efficiently used to refine, improve, and validate the logics HLR defined as input of the software design process.

In this context, a prototype may be developed in SCADE Suite for all functional HLR 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.

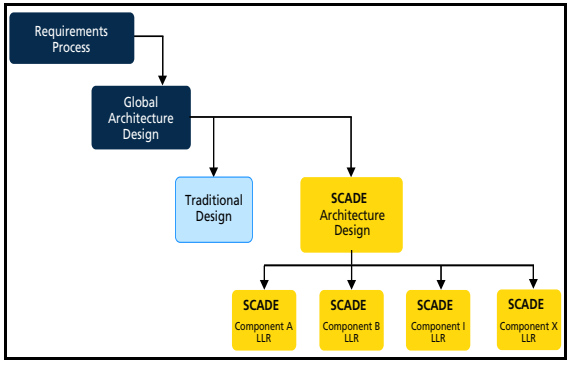


Figure 4.2: Software design process with SCADE Suite

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 SCADE Suite 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 SUITE ARCHITECTURE DESIGN

An architecture design model can be developed in SCADE Suite as shown in the next figure. It is also possible to use SCADE Architect for designing the software architecture and synchronizing this model with SCADE Suite.

The purpose of the software architecture design model is to:

- Identify high-level functions: typically one develops 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 SCADE Suite semantic checks
- Prepare the framework for the detailed design process: define the top-level functions while ensuring consistency of their interfaces

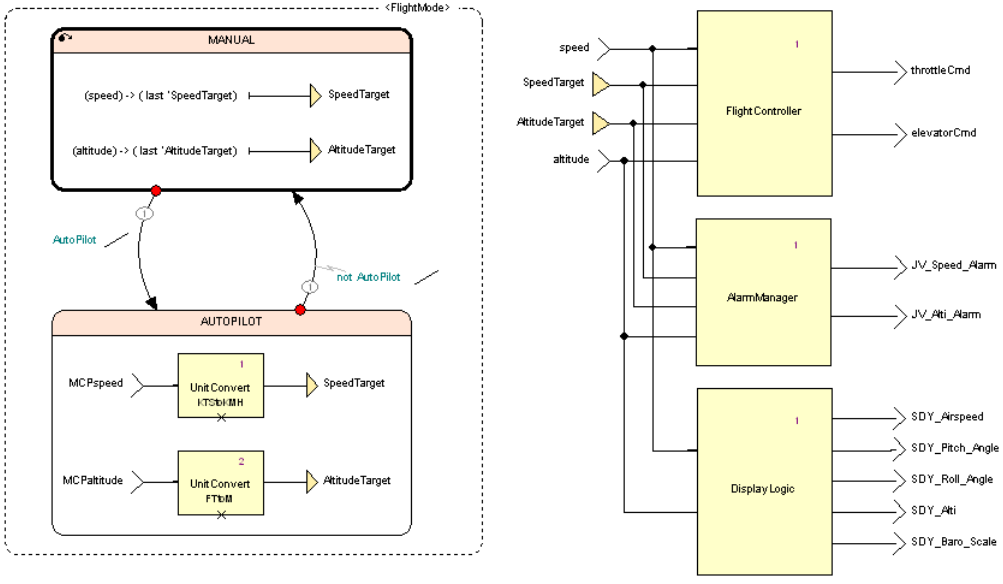


Figure 4.3: Top-level view of a simple Flight Control System

This architecture design model is extremely important because it lays the foundations for the Logics LLR.

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 the Scade language, but also requires to minimize the complexity of the model.
- Efficiency: There is no magic recipe for achieving a good model architecture with SCADE products, 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 the first draft, but build two or three architecture variants, then analyze and compare them. You may otherwise have to live with a bad architecture for a long time.
  - Review and discuss the architecture with peers.
  - Select the architecture that minimizes connection complexity 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 and with a manual coding approach. Inputs/Outputs of a 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, the logics architecture models are refined to design the low-level requirements (LLR). The objective of this activity is to produce a set of complete and consistent SCADE Suite design models.

LOGICS LLR DEFINITION

The definition and granularity of an LLR in SCADE Suite models are determined by the user itself.

For instance, LLRs can be mapped to:

- user-defined operators (nodes or functions declared by users to define operators with/without memory, imported operators, or operators specialized by other operators)
- diagrams (graphical or textual representation of dataflow and states)
- or equation sets (grouping design elements graphically in diagrams to allowing global commenting, annotating or tracing)

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

4.3.2.1 Logics LLR development with SCADE Suite

The Scade 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 Suite 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 Suite graphical notation allows representing block diagrams exactly in 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 the Scade language. For example, if a control engineer has written an equation such as s=K1*u - K2* z-1*s, which means s(k)=K1*u(k) - K2* s(k-1), this can be expressed textually as s=K1*u-K2*pre(s) or graphically, as shown in Figure 4.4 below.

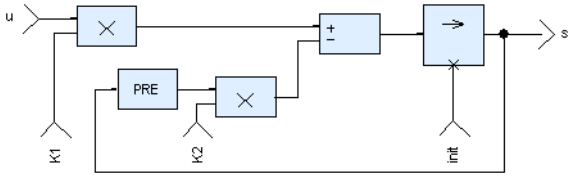


Figure 4.4: A first order filter

It is possible to 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

In SCADE Suite, a variety of techniques are available 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 Suite 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.

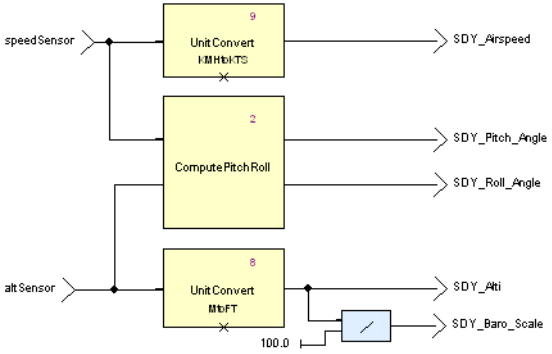


Figure 4.5: Complex display logic and simple functions

- 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.

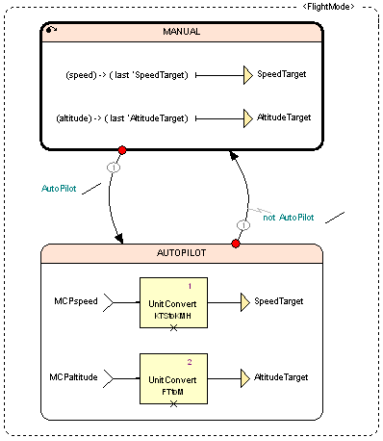


Figure 4.6: State machine in Flight mode management

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 degraded 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 Reusable components and library management

4.3.3.1 SCADE library software life cycle

A SCADE Suite library[6] object must be developed as any other SCADE Suite software 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 (HLR) of the application.
- Good practices consist in defining functional requirements (derived HLR) for these library components as a separate document and in developing and verifying the components from its derived HLR.
- 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.

Section 4.3.3.2 below describes several examples where the use of reusable components is relevant for logics.

4.3.3.2 Re-usability with SCADE Suite library components

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 RE-USABILITY

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

- Library
- Genericity/Polymorphism
- Parameterization by size

6. Libraries distributed with SCADE Suite product are provided as examples; they were not developed following the process described in this section.

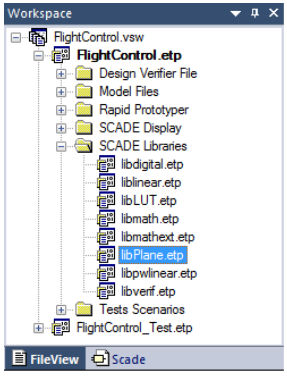


Figure 4.7: Concept of SCADE Suite library

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.

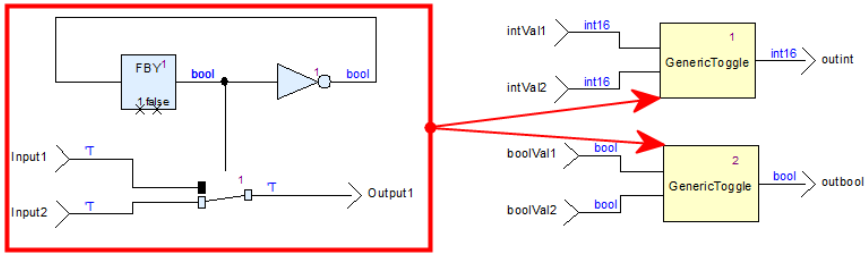


Figure 4.8: Example of generic operator instantiated with int and bool types

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).

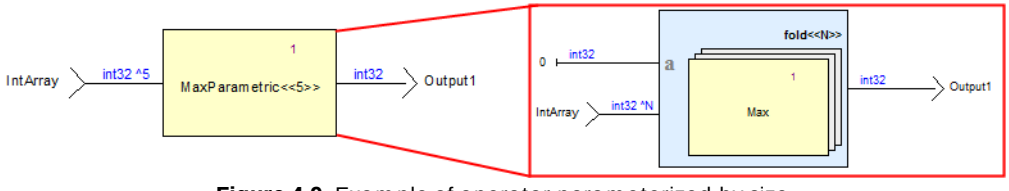


Figure 4.9: Example of operator parameterized by size

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 HLR (including derived HLR 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

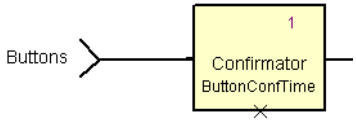


Figure 4.10: Inserting Confirmator in Boolean input flow

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.

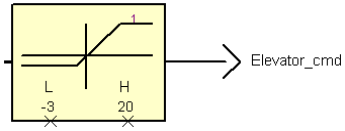


Figure 4.11: Inserting Limiter in output flow

Since the data flow is very explicit in SCADE Suite models, 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 SCADE 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 SCADE 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.

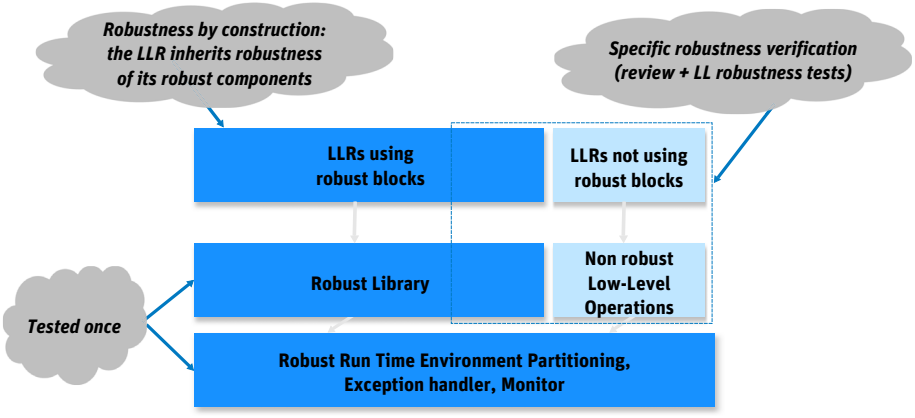


Figure 4.12: Example of robust architecture

On the left part, the robustness of the design relies on a set of low-level robust operators. Two benefits can be highlighted in this context:

1 The corresponding software application inherits robustness from its low-level robust components.

2 The verification strategy of such robust components is optimized because the library operator is tested once according to its robustness requirements.

On the right part, the approach is not optimal because the low-level operations are not systematically robust: a specific and integral robustness analysis is required to ensure the robustness of the

whole software application and the corresponding verification effort should be higher.

See Section 5.5.5 for more information about the verification strategy regarding the robustness of a SCADE Suite application.

4.4 Software Coding Process

The SCADE Suite KCG code generator automatically generates the complete code that implements the software design defined in formal notation for both data flows and state machines (see Figure 4.13). It is not just a generation of skeletons; the complete dynamic behavior is implemented.

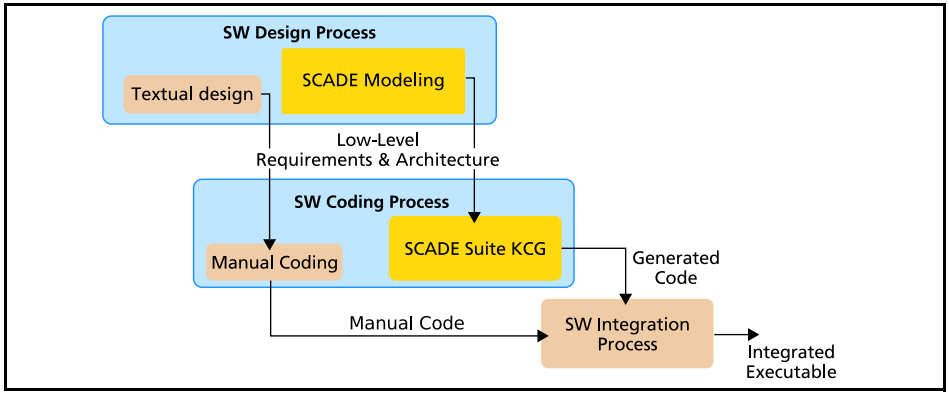


Figure 4.13: Software coding process with SCADE Suite

4.4.1 Code generation from SCADE Suite models

The 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.

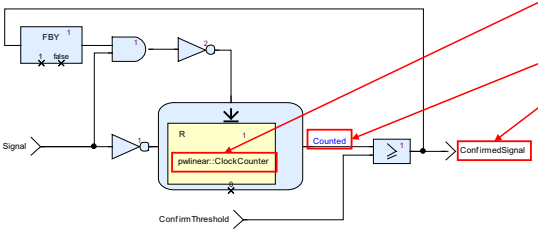
PROPERTIES OF THE GENERATED CODE

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

- The code is portable: it is [ISO-C] and [ISO-Ada] 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/Ada code is ensured.
- The code is readable and traceable to the input 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 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 from the generated code to a SCADE Suite data flow is illustrated in Figure 4.14.




```
/* FlightControl::Confirmator/ */
void Confirmator_FlightControl(
  inC_Confirmator_FlightControl *inC,
  outC_Confirmator_FlightControl *outC)
{
  /* _L7/ */
  kcg_bool _L7;

  _L7 = !(outC->_L4 & inC->Signal);
  if (_L7) {
    /* Counted=(pwlinear::ClockCounter#1)/ */
    ClockCounter_pwlinear
      (kcg_bool) !inC->Signal,
      &outC->Context_ClockCounter_1);
    outC->Counted = outC->Context_ClockCounter_1.Count;
  }
  outC->ConfirmedSignal = outC->Counted >= inC->ConfirmThreshold;
  outC->_L4 = outC->ConfirmedSignal;
}

#ifndef KCG_USER_DEFINED_INIT
void Confirmator_init_FlightControl(outC_Confirmator_FlightControl *outC)
{
  outC->_L4 = kcg_false;
  outC->ConfirmedSignal = kcg_true;
  /* Counted=(pwlinear::ClockCounter#1)/ */
  ClockCounter_init_pwlinear(&outC->Context_ClockCounter_1);
  outC->Counted = kcg_lit_int32(0);
}
#endif /* KCG_USER_DEFINED_INIT */


#ifndef KCG_NO_EXTERN_CALL_TO_RESET
void Confirmator_reset_FlightControl(outC_Confirmator_FlightControl *outC)
{
  outC->_L4 = kcg_false;
  /* Counted=(pwlinear::ClockCounter#1)/ */
  ClockCounter_reset_pwlinear(&outC->Context_ClockCounter_1);
  outC->Counted = kcg_lit_int32(0);
}
#endif /* KCG_NO_EXTERN_CALL_TO_RESET */
```

Figure 4.14: SCADE Suite data flow to generated C source code traceability

Traceability from the generated code to a SCADE Suite state machine is illustrated in Figure 4.15.

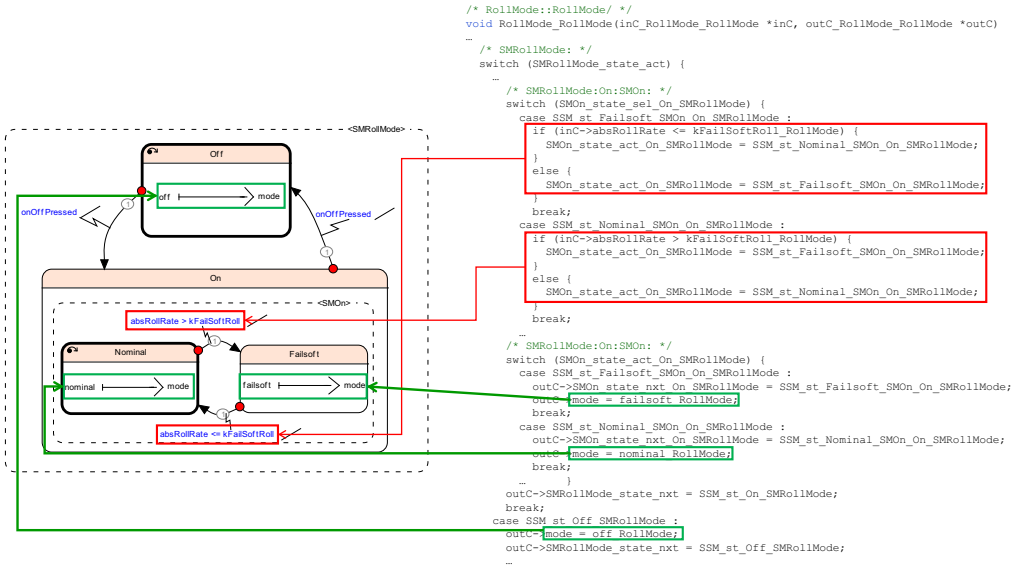


Figure 4.15: SCADE Suite state machine to generated C source code traceability

To further support automated analysis of traceability between model constructs and code, a traceability file (`mapping.xml`) is generated by SCADE Suite KCG. A Python API allowing to access this file content is provided with SCADE Suite.

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 using SCADE Suite Timing and Stack Verifiers. Specified as a SCADE Suite 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 an operator:

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

This is illustrated in Figure 4.16.

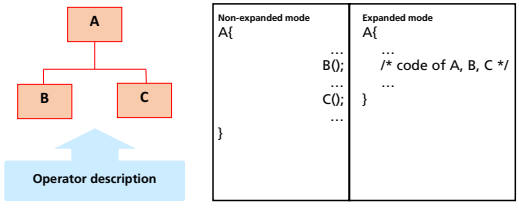


Figure 4.16: Non-expanded and Expanded modes

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/pragma) may have an impact on the structure of the generated code, on the integration of the 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 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/Ada global variables and not passed as arguments of the root C/Ada functions. This change on the signature of root C/Ada functions impacts the integration of KCG generated code.
- The separate_io SCADE Suite KCG option and/or pragma applies to an 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 generated code.

4.4.2 Code generation from multiple components

CODE GENERATION FOR MULTIPLE LOGICS COMPONENTS

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

This process is usually sufficient because it ensures 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 components (interacting or not together) where each component corresponds to a single 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 model.

As shown in Figure 4.17, there are two alternatives for generating code:

1. Generating all code in one run, using the “multi-root operators” SCADE Suite KCG option (see [SUITE-UM] for further information on options). This applies whether root operators are defined in the same model or not. When 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 C/Ada 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 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.

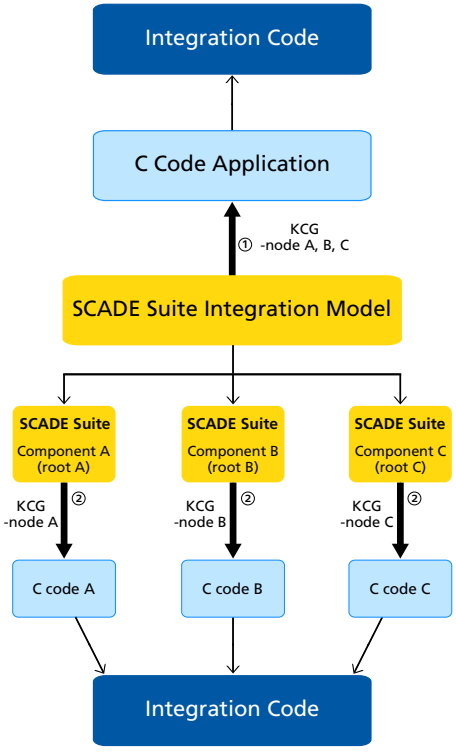


Figure 4.17: Code generation and multiple components

- The manifest pragma is used to control the type names generated by KCG. It ensures better stability of the code between two code generation sessions.
- The global prefix KCG option is used to prevent name conflicts 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

The integration of a SCADE Application is about:

- Interface with the external environment (Inputs/Outputs)
- 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 Suite functions are not active, then a driver may pass its data directly to SCADE Suite inputs. 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 Suite function remain stable, while the function is computing the current cycle. This can be ensured by separating the internal buffer of the driver from the input vector and by performing a transfer (or address swap) before each computation cycle starts. These drivers are usually not developed in the Scade

4.5.3 SCADE Suite module integration

A module refers here to the C/Ada 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 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 predefined Scade types (see Section 3.2.1.6) which must be mapped to C/Ada 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 code

SCADE Suite allows to reference external code in models.

On the logics side, the Scade language includes the concept of imported constants, types, and functions (a tag "imported" is set at the declaration level). The declaration of these external data is performed at model level in Scade language whereas their definition is given in host language (implementation in C code). A typical example for SCADE Suite is the usage of imported functions such as trigonometric functions or byte encoding and checksum functions. At integration time, these functions have to be compiled and linked to the SCADE Suite-generated code.

For model simulation purposes, SCADE Test automatically compiles and links external code when the path names of the source files are given in the project settings.

4.5.5 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 Suite models, and then examines how to implement scheduling of a model in single or multirate mode, while in single tasking or multitasking mode.

SCADE SUITE EXECUTION SEMANTICS

The SCADE Suite execution semantics is based on a cycle-based execution model as described in Section 3.2.2. This model can be represented with Figure 4.18.

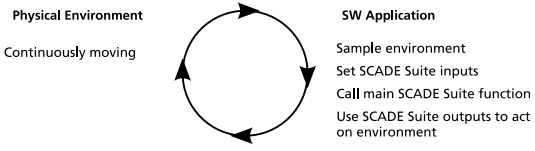


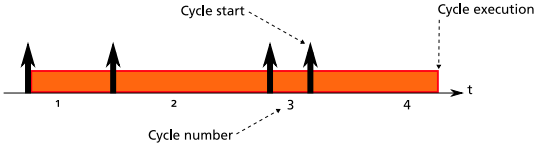
Figure 4.18: Execution semantics of SCADE Suite

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

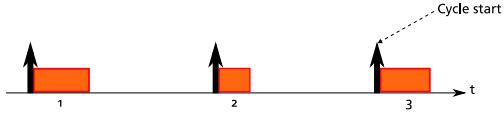
BARE SYSTEM IMPLEMENTATION

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

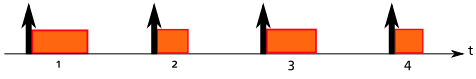
- Polling: a new cycle is started immediately after the end of the previous one in an infinite loop.



- 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 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:

```
begin_loop
waiting for an event (usually clock signal)
setting SCADE Suite inputs
calling SCADE Suite generated main functions
using SCADE Suite 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 design, which can be a periodic alarm or a user activation.

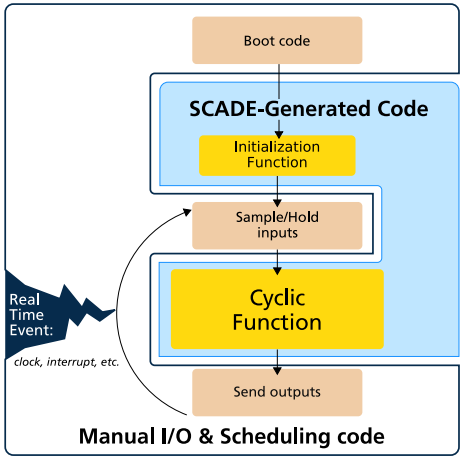


Figure 4.19: SCADE Suite code integration

This architecture can be designed by hand for any RTOS. SCADE Suite provides automation of this code production through the SCADE Code Integration Toolbox allowing to develop user-specific Adaptors for VxWorks® 653 from Wind River®, for Integrity®-178B from Green Hills® Software, for PikeOS from SYSGO, for Deos™ from DDC-I, and for many platforms at major suppliers and integrators.

Note that concurrency is expressed functionally in SCADE Suite models and that SCADE Suite KCG 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 Suite generated code.

MULTIRATE, SINGLE-TASK APPLICATIONS

SCADE Suite can be used to design multirate applications in a single OS task. Some parts of the design can be executed at a slower rate than the top-level loop. Putting a slow part inside an *activate*[7] 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.

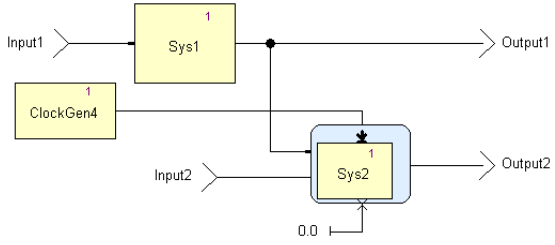


Figure 4.20: Modeling a bi-rate system

The schedule of this application is as shown in Figure 4.21 below:

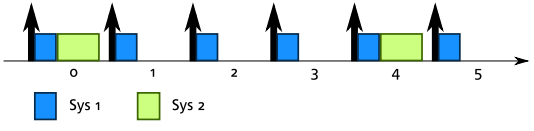


Figure 4.21: Timing diagram of a bi-rate system

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:

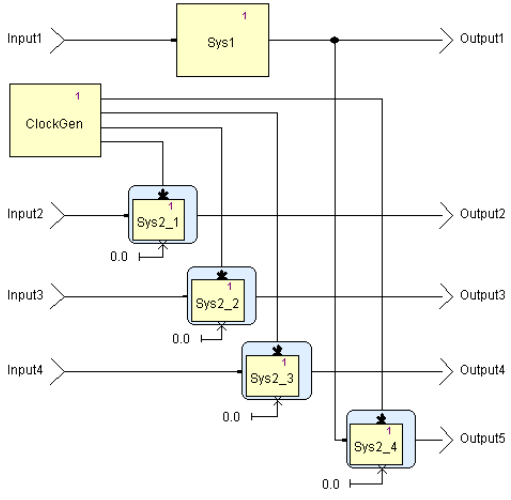


Figure 4.22: Modeling slow system over four cycles

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:

7. The Boolean Activate operator has an input condition (on top) used to trigger the execution of the computation that is described inside the block, thus allowing the introduction of various rates of execution for different parts of a model. The operator execution only occurs when a given activation condition is true.

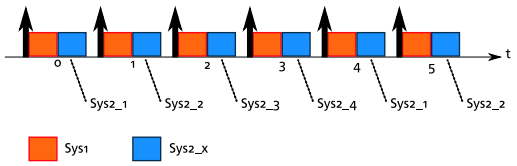


Figure 4.23: Timing diagram of distributed computations

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

The multirate aspect of a SCADE Suite design is achieved using standard constructs. This has no effect on the external interface of the 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 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 Suite 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 software pieces developed by each team. The concept of SCADE project (etp file) supported by SCADE Suite makes easier collaborative work and re-usability. A SCADE project has no semantic meaning: it is a pure organizational entity.

Whatever the architecture, we usually consider several categories of projects:

- A component project that provides a complete functional view of a given SCADE macro-component

- A set of library projects that contains shared objects such as types, constants, and functions intentionally located in a dedicated project for re-usability purposes or due to Intellectual Properties (IP) constraints. Such library projects are referenced in a component project and/or top-level project.
- A top-level project for the integration of the different SCADE macro-components. This project is also called “integration project” or “architecture project”.

In a typical project organization:

- A software architect is in charge of managing the top-level project, defining in particular the macro-components, their interfaces, and connections.
- A library manager is in charge of defining the different library projects and their content.
- Each macro-component or library is developed by a specific engineering team. The interface of such macro-components or library components defines a framework for these teams, that maintain the consistency of the design.

Figure 4.24 below describes a typical teamwork organization for logics.

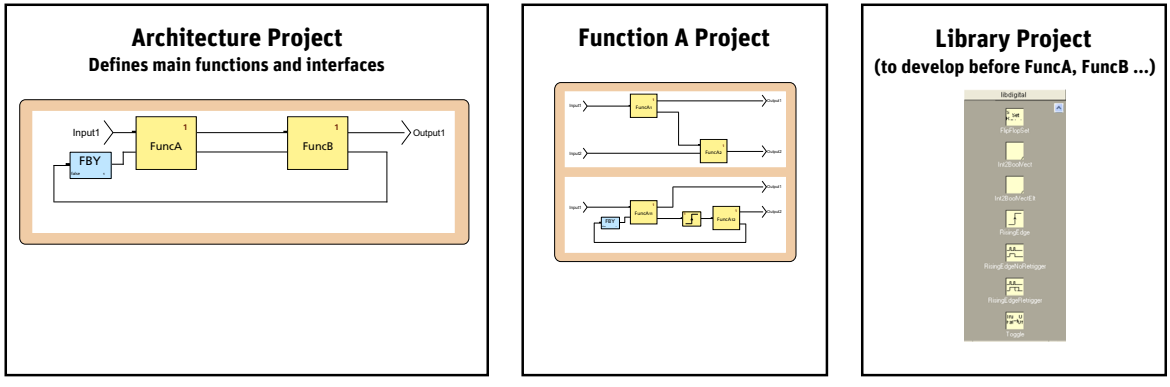


Figure 4.24: Typical teamwork organization

The best organization is to consider one single engineer working on one separate etp file. This etp file groups XSCADE files (*.xscade) or SCADE files (*.scade) corresponding to the definition of a macro-component (see “Function A project” in Figure 4.24) or a library (see “Library project” in Figure 4.24).

If several engineers are required for the development of a macro-component or a library, the finest modularity is to consider no more than one engineer for one XSCADE (resp SCADE) file.

At each step of the software integration, the team can verify in a mouse click that a SCADE Suite component remains consistent with its interface thanks to semantic checks using SCADE Suite.

Later, the integration of these parts into a larger model can be achieved by linking the “projects” to the larger one and the integration consistency is also verified by semantic checks using SCADE Suite.

All development data (etp, [X]SCADE files) have to be kept under strict version and configuration management control by using any commercial Configuration Management System (CMS).

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 5.1: DO-178C Table A-3

	Objective Description	Activity Ref
1	High-level requirements comply with system requirements	6.3.1
2	High-level requirements are accurate and consistent	6.3.1
3	High-level requirements are compatible with target computer	6.3.1
4	High-level requirements are verifiable	6.3.1
5	High-level requirements conform to standards	6.3.1
6	High-level requirements are traceable to system requirements	6.3.1
7	Algorithms are accurate	6.3.1

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 as detailed in Section 4.2. These requirements must be verified against the objectives of DO-178C Table A-3.

When the requirements from which a 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 described textually.

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

	Objective Description	Activity Ref	Verification Method
1	High-level requirements comply with system requirements	6.3.1	Peer review
2	High-level requirements are accurate and consistent	6.3.1	Peer review
3	High-level requirements are compatible with target computer	6.3.1	Peer review
4	High-level requirements are verifiable	6.3.1	Peer review
5	High-level requirements conform to standards	6.3.1	Peer review
6	High-level requirements are traceable to system requirements	6.3.1	Peer review
7	Algorithms are accurate	6.3.1	Peer review

5.3 Verification of SCADE Low-Level Requirements and Architecture

5.3.1 Verification objectives for the LLR and architecture

The SCADE Suite design models (see Section 4.3) have 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

	Objective Description	Activity Ref
1	Low-level requirements comply with high-level requirements	MB.6.3.2 MB.6.7 MB.6.8.1 (see Item 1)
2	Low-level requirements are accurate and consistent	MB.6.3.2 MB.6.8.1 (see Item 1)
3	Low-level requirements are compatible with target computer	MB.6.3.2
4	Low-level requirements are verifiable	MB.6.3.2 MB.6.8.1 (see Item 1)
5	Low-level requirements conform to standards	MB.6.3.2
6	Low-level requirements are traceable to high-level requirements	MB.6.3.2
7	Algorithms are accurate	MB.6.3.2 MB.6.8.1 (see Item 1)
8	Software architecture is compatible with high-level requirements	MB.6.3.3 MB.6.8.1 (see Item 1)
9	Software architecture is consistent	MB.6.3.3 MB.6.8.1 (see Item 1)

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

	Objective Description	Activity Ref
10	Software architecture is compatible with target computer	MB.6.3.3
11	Software architecture is verifiable	MB.6.3.3 MB.6.8.1 (see Item 1)
12	Software architecture conforms to standards	MB.6.3.3
13	Software partitioning integrity is confirmed	MB.6.3.3
MB 14	Simulation cases are correct (see Item 1)	MB.6.8.1 MB.6.8.3.2
MB 15	Simulation procedures are correct (see Item 1)	MB.6.8.1 MB.6.8.3.2
MB 16	Simulation results are correct and discrepancies explained (see Item 1)	MB.6.8.1 MB.6.8.3.2

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

Compliance with HLR is verified through a combination of techniques:

- Model simulation
- Peer review
- Formal verification

5.3.2.1 Model simulation

Model simulation allows exercising the behavior of a model. As stated in [DO-331], MB.6.8.1, its main purpose is to provide repeatable evidence of compliance of the model to the requirements from which the model was developed.

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 models requires the following activities:

- SCADE verification cases and procedures are developed from the requirements from which the SCADE model was developed (HLR)
- SCADE verification cases and procedures shall 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)
- HLR are covered by SCADE verification cases and procedures.
- SCADE verification cases and procedures are reviewed to confirm that they are correct (see objectives MB.14 and MB.15).
- SCADE models are exercised by HLR-based 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)

Note: “SCADE verification cases and procedures” is a generic term to designate both

- The simulation cases and procedures used for SCADE model simulation on host during design verification ([DO-331] Table MB.A-4)
- The test cases and procedures used for Executable Object Code (EOC) Testing on target ([DO-331] Table MB.A-6)

Model simulation, when it is supported by a qualified tool, may be used to formally satisfy some objectives of Table A-4 as it is shown below during the verification of logics architecture and LLR. On the other hand, some peer reviews and/or analysis are still required to fully address the design verification objectives as illustrated below.

VERIFICATION OF LOGICS ARCHITECTURE AND LLR

SCADE Test Environment fully supports the dynamic verification of SCADE Suite models with regard to the logics HLR.

1 SCADE Test Environment for Host provides an integrated environment that allows validation and verification engineers to both automate the creation and management of simulation cases (see Figure 5.1) and then to run on host the verification cases created from the HLR (see Figure 5.2).

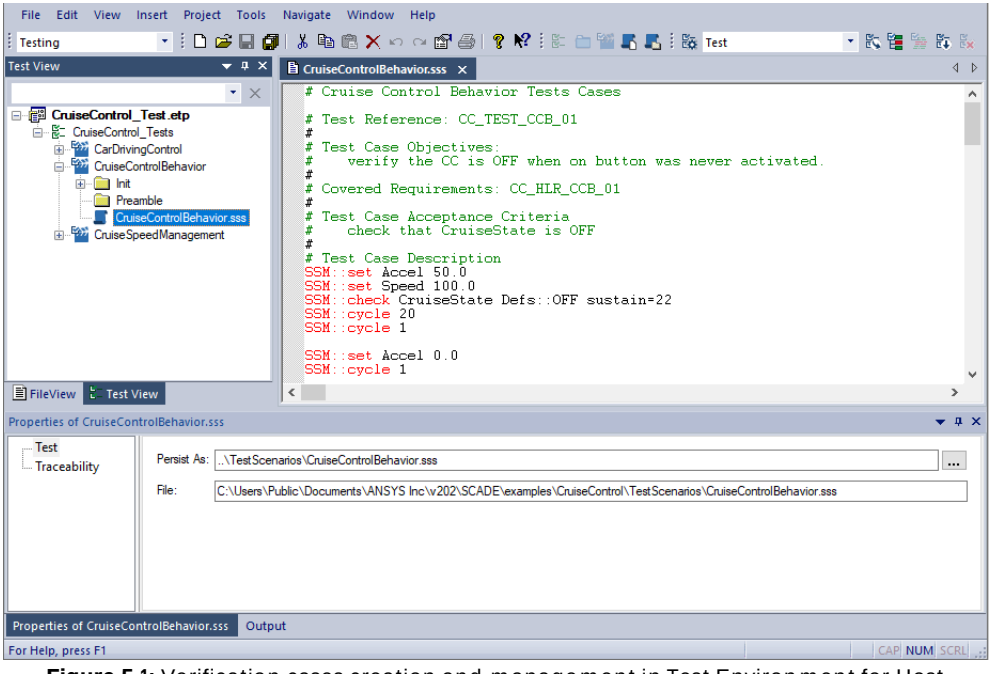


Figure 5.1: Verification cases creation and management in Test Environment for Host

With SCADE Test Environment for Host, the generation of test conformance reports (containing simulation results) 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 HLR.

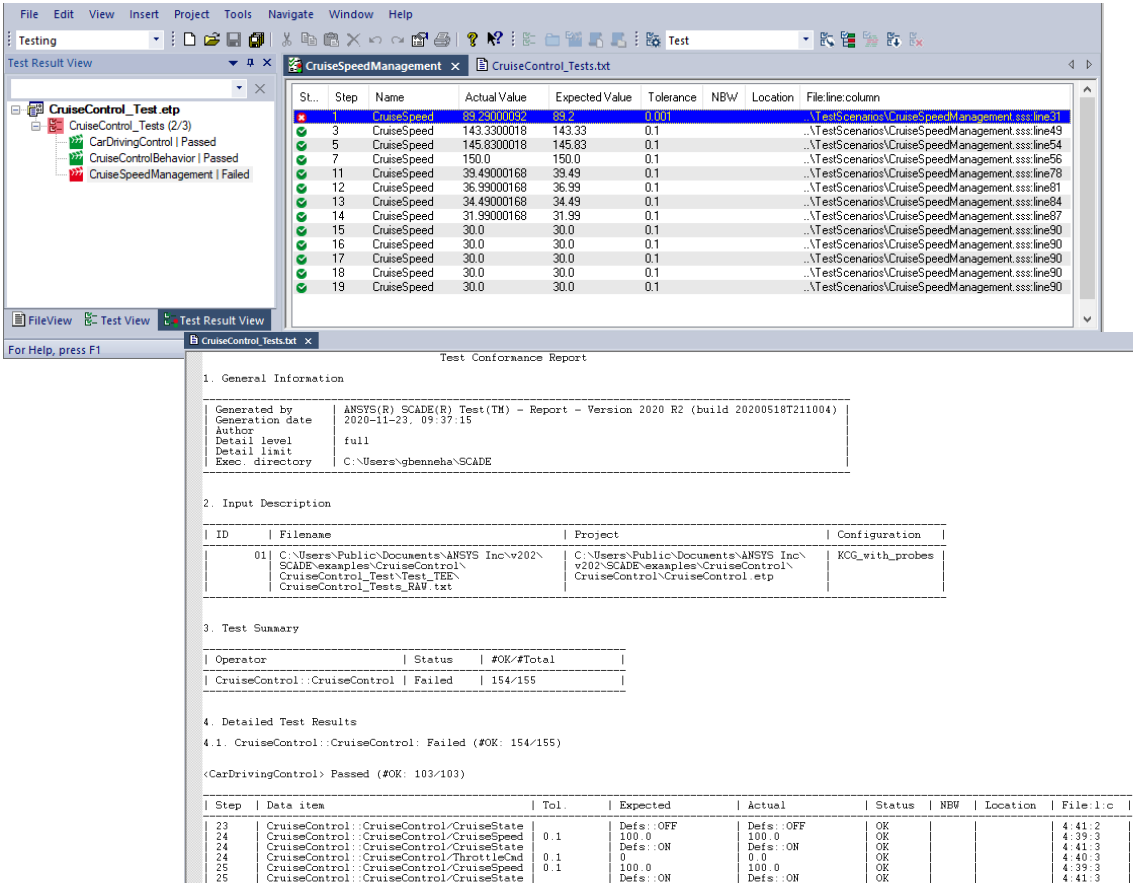


Figure 5.2: Simulation results from running verification cases on host

2 SCADE Test Model Coverage[8] is a coverage analysis tool that executes and reports coverage from HLR-based SCADE Suite verification cases and procedures at model level. Model coverage analysis (see Figure 5.3) is required during design verification with the objective to assess completeness of the verification cases. SCADE Test Model Coverage is qualified as a verification tool for DO-178C/DO-330 at TQL-4. This qualification evidence allows applicants to claim credit from model coverage measurement. For further information about Test Model Coverage concepts and usage, refer to Chapter 6/

8. Support for Ada code available from SCADE 2021 R1 onward.

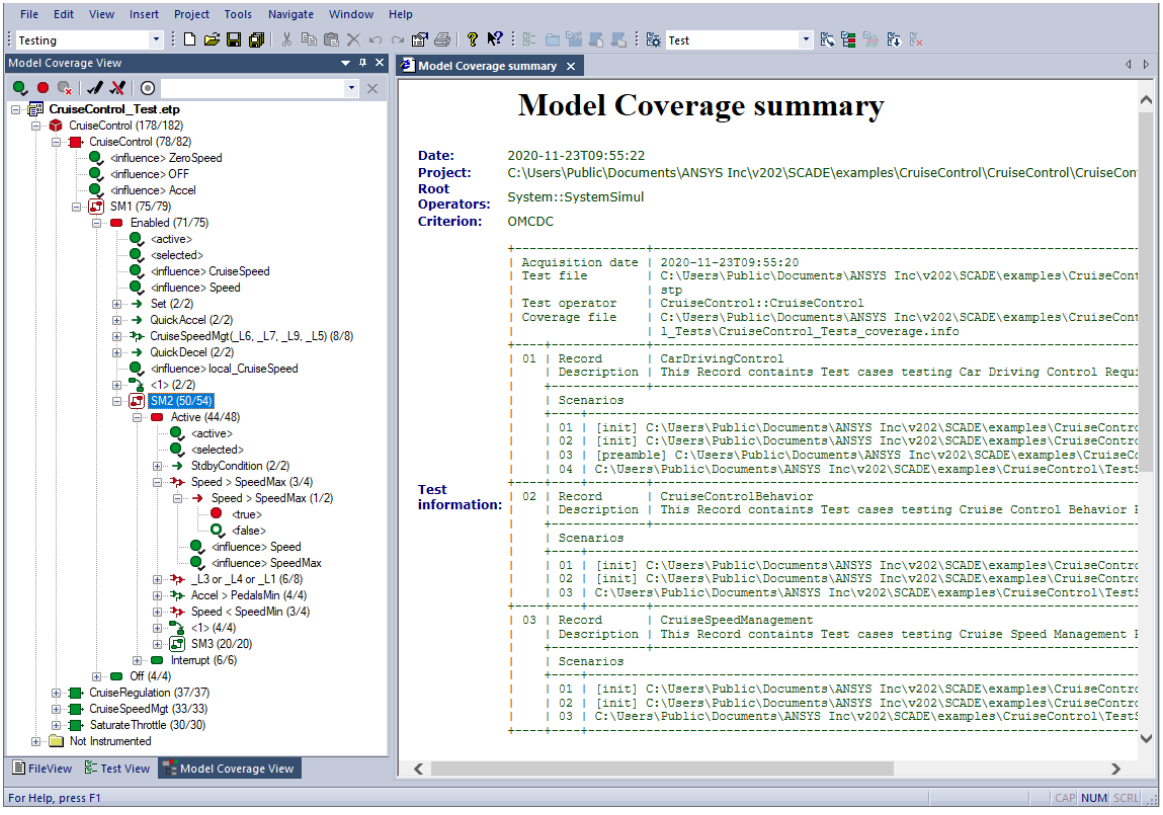


Figure 5.3: Model coverage analysis with SCADE Test Model Coverage

5.3.2.2 Peer reviews with SCADE LifeCycle Reporter

Additional peer reviews focused on HLR/LLR traceability analysis and design robustness analysis only can be performed based on the report 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 model. The notation used for SCADE Suite models 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

5.3.2.3 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[9] provides an original and powerful verification technique based on formal verification technologies.

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 LLR 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 Suite operator one would express the safety property shown in Figure 5.4 below, reflecting the property above. This operator is called an observer.

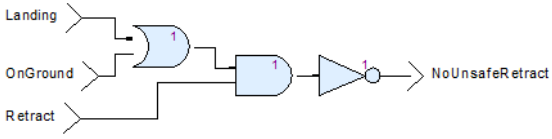


Figure 5.4: Observer operator containing landing gear safety property

Then, one would connect the observer operator to the controller in a verification context operator, as in Figure 5.5 below.

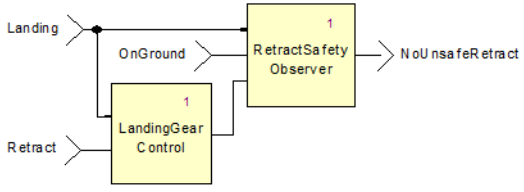


Figure 5.5: Connecting the observer operator to the landing gear controller

In 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.

9. SCADE Suite Design Verifier is powered by Prover® PSL from Prover Technology. Prover, Prover Technology, Prover Plug-in, and the Prover logo are trademarks or registered trademarks of Prover Technology AB in European Union, the United States, China, and in other countries.

5.3.3 Model accuracy and consistency

Syntactic and semantic checks using SCADE Suite Checker perform an in-depth analysis of logics consistency, including:

- Detection of missing definitions
- Warnings on unused definitions
- Detection of dependency to an uninitialized flow
- Type consistency check of operator instance actual parameters with operator interface
- Detection of causality issues *i.e.*, immediate dependency of a flow definition with the flow itself
- Clock consistency check to ensure that flows are produced and consumed at the same rate

5.3.4 Compatibility with target computer

The objective is to ensure that no conflict exists between the low-level requirements, the architecture and the hardware/ software features of the target platform.

In the context of SCADE models, the following aspects shall be considered:

- Models complexity
- Execution time and memory size
- Compatibility of generated code with target platform

MODEL COMPLEXITY ANALYSIS

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

It is strongly recommended to define the rules related to the management of SCADE models complexity in the Software Model Standards document (see DO-331, MB.11.23).

Regarding SCADE Suite models, typical complexity metrics such as "the maximum number of diagrams for an operator", "the maximum number of user-operators within a diagram", or "the maximum number of nested levels of conditional operators" are defined in the SCADE Suite 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 rules by using SCADE Suite Rules Checker[10] scripting capabilities. This tool is not qualified: qualification must be done by the user for ones' specific rules. For further information on scripting capabilities, refer to SCADE Suite User Manual [SUITE-UM].

EXECUTION TIME AND MEMORY SIZE ANALYSIS IN SCADE SUITE MODELS

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.

10.Available from SCADE 2019 R1 onwards.

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 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 model to determine the most efficient design. SCADE Suite users can use it to monitor the performances of their design with respect to WCET and stack usage. This tool is relevant, in particular, for early verification of the compatibility between the model and the target platform.

Timing and Stack Verifiers (TSV) compute precise WCET and stack size for a 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 and hyperlinks are available for direct reference to the model constructs matching each WCET and/or stack size results.

Figure 5.6 illustrates global visualization results.

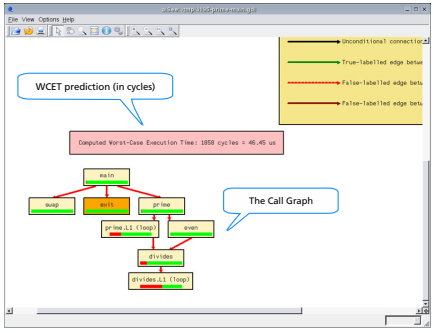


Figure 5.6: Timing and Stack analysis global visualization

Figure 5.7 illustrates global and detailed results for Timing analysis.

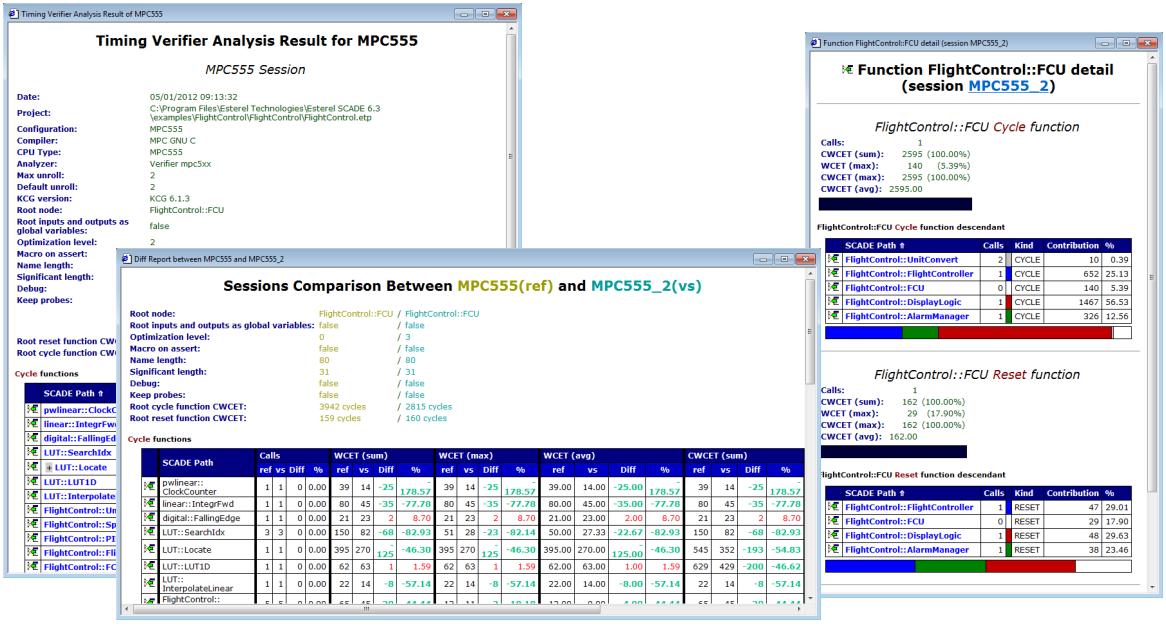


Figure 5.7: Timing Verifier analysis reports

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

COMPATIBILITY OF GENERATED CODE WITH TARGET PLATFORM

SCADE Suite includes a Compiler Verification Kit (CVK) with the objective of verifying that the type of code generated by SCADE Suite KCG is correctly compiled/executed with a given cross-compiler on target platform.

CVK supports early verification of the correctness and consistency of the development environment with the development standards and the 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 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/.

5.3.5 Verifiability

Since SCADE Suite has a formal notation, the corresponding models are formally verifiable.

Such verifiability is confirmed by SCADE Suite syntactic and semantic checks (see Section 5.3.3) when no errors or warnings (that cannot be justified) are raised by the respective tools.

SCADE Suite model complexity must also be monitored to ensure design verifiability according to the procedure described in Section 5.3.4.

5.3.6 Conformity to standards

Two levels of rules must be considered for SCADE models:

- SCADE Suite built-in rules: they are predefined rules directly from the definition of SCADE Suite formal notation. Regarding the logics, the Scade Language Reference Manual [SCS-KCG-LRM] defines what a correct Scade model is, and what a correct Scade model defines. The former is called “static semantics” as formally defined in [SCS-KCG-LRM], the later is also defined in the same document in a semi-formal way (text and mathematics). KCG front-end first implements all the static checks defined in [SCS-KCG-LRM] and stops whenever the defined static discipline is not satisfied; then it generates a code that implements the dynamic semantics. Using SCADE Suite Checker, it is possible to invoke KCG front-end to check the static semantics only.
- User design rules related to SCADE models: they are additional rules defined by the user in its Software Model Standards (DO-331, MB.11.23) for readability, verifiability, and maintainability purposes. These rules must be checked either automatically or manually. In the context of automatic verification, users are able to develop their own design rules by using the Python capabilities (see 5.3.3 for details about SCADE products scripting capabilities).

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 Logics LLR within a model, please refer to Section 4.3.2.

Trace data must confirm that:

•All HLR are covered by SCADE LLR[11];

•All SCADE LLR are correctly traced to HLR;

•All SCADE LLR that are not traced to HLR are explicitly identified as derived SCADE LLR by design choice.

11. SCADE LLR is used as a generic term to designate Logics LLR .

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

This traceability analysis is efficiently supported by SCADE LifeCycle Application Lifecycle Management Gateway that allows connection to ALM tools for the creation of HLR/LLR traceability links from the model-based design environment (see Figure 5.8).

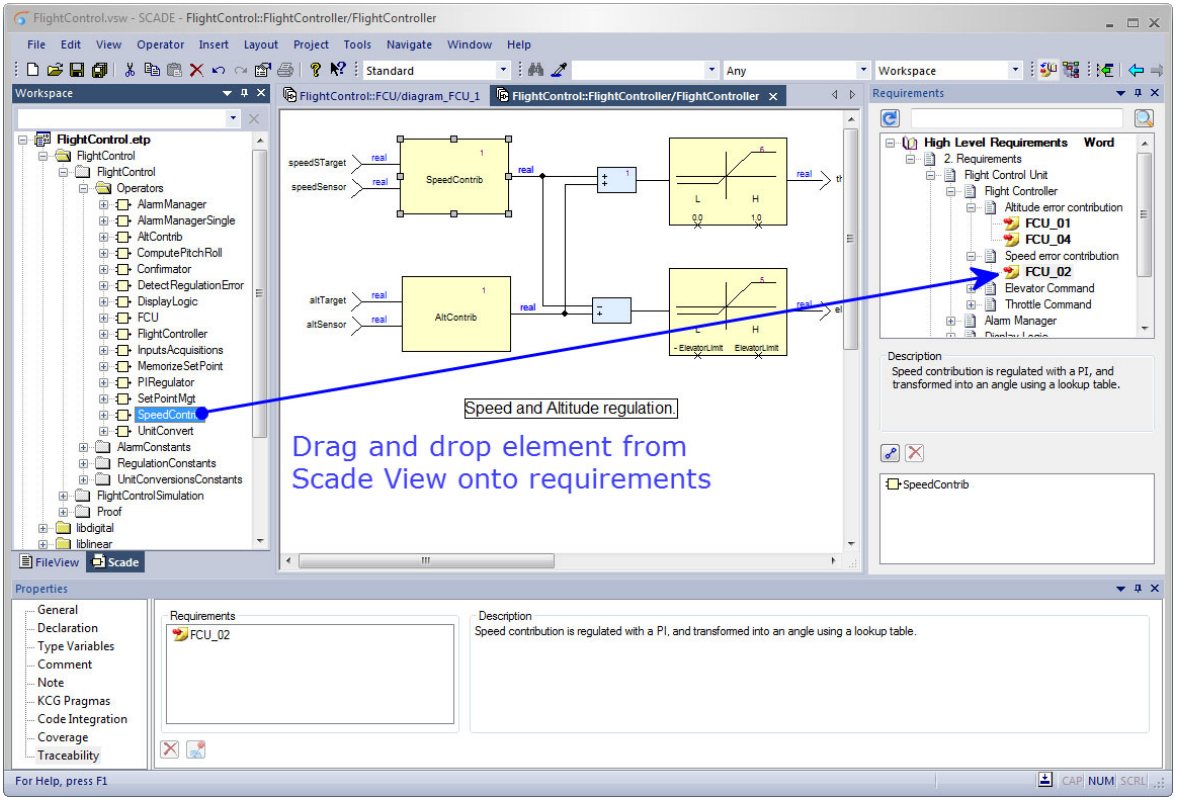


Figure 5.8: Creating LLR/HLR traceability links within ALM Gateway

5.3.8 Algorithms accuracy

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

The review of SCADE 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 the SCADE model-based design environments. 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 ARINC 653 compliant operating systems.

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). This is the case for the logics part of the application.

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

The review of verification cases must confirm that:

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

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

The review of simulation results must confirm that:

- Simulation results are expected results
- Discrepancies between actual and expected results generate problem reports

Simulation results generated by SCADE Test Environment for Host and reported in test conformance reports 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 and correctly reported.

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 5.4: DO-331 Table MB.A-4 Objectives Achievement

	Objective Description	Ref	Activity Ref	Verification Method
1	Low-level requirements comply with high-level requirements	MB.6.3.2.a	MB.6.3.2 MB.6.7 MB.6.8.1 (see Item 1)	Peer review with SCADE LifeCycle Reporter Model Simulation with SCADE Test Environment for Host Model Coverage with SCADE Test Model Coverage
2	Low-level requirements are accurate and consistent	MB.6.3.2.b	MB.6.3.2 MB.6.8.1 (see Item 1)	Automated by SCADE Suite Checker (syntactic and semantic)

Table 5.4: DO-331 Table MB.A-4 Objectives Achievement (Continued)

	Objective Description	Ref	Activity Ref	Verification Method
3	Low-level requirements are compatible with target computer	MB.6.3.2.c	MB.6.3.2	Analysis of SCADE Suite models complexity Analysis of SCADE models execution time and memory size with SCADE Suite TSO/TSV CVK execution on target
4	Low-level requirements are verifiable	MB.6.3.2.d	MB.6.3.2 MB.6.8.1 (see Item 1)	Analysis of SCADE Suite models complexity Automated by SCADE Suite Checker (syntactic and semantic)
5	Low-level requirements conform to standards	MB.6.3.2.e	MB.6.3.2	SCADE built-in rules: Automated by SCADE Suite Checker User design rules: peer review or automated by SCADE Suite Checker (if any design rules existing)
6	Low-level requirements are traceable to high-level requirements	MB.6.3.2.f	MB.6.3.2	Traceability analysis with SCADE LifeCycle ALM Gateway
7	Algorithms are accurate	MB.6.3.2.g	MB.6.3.2 MB.6.8.1 (see Item 1)	Peer review Model Simulation of numerical algorithms with SCADE Test Environment for Host
8	Software architecture is compatible with high-level requirements	MB.6.3.3.a	MB.6.3.3 MB.6.8.1 (see Item 1)	Peer review with SCADE LifeCycle Reporter
9	Software architecture is consistent	MB.6.3.3.b	MB.6.3.3 MB.6.8.1 (see Item 1)	Automated by SCADE Suite Checker (syntactic and semantic)
10	Software architecture is compatible with target computer	MB.6.3.3.c	MB.6.3.3	Analysis of SCADE Suite architecture models complexity CVK execution on target
11	Software architecture is verifiable	MB.6.3.3.d	MB.6.3.3 MB.6.8.1 (see Item 1)	Analysis of SCADE Suite architecture models complexity Automated by SCADE Suite Checker (syntactic and semantic)
12	Software architecture conforms to standards	MB.6.3.3.e	MB.6.3.3	SCADE built-in rules: Automated by SCADE Suite Checker (syntactic and semantic) User design rules: peer review or automated by SCADE Suite Checker (if any design rules existing)
13	Software partitioning integrity is confirmed	MB.6.3.3.f	MB.6.3.3	SCADE Suite introduces no specific risk, but provides no partitioning mechanism; traditional method has to be used
MB 14	Simulation cases are correct (see Item 1)	MB.6.8.3.2.a	MB.6.8.1 MB.6.8.3.2	Peer review of SCADE Suite verification cases
MB 15	Simulation procedures are correct (see Item 1)	MB.6.8.3.2.b	MB.6.8.1 MB.6.8.3.2	Peer review of SCADE Suite verification procedures
MB 16	Simulation results are correct and discrepancies explained (see Item 1)	MB.6.8.3.2.c	MB.6.8.1 MB.6.8.3.2	Analysis of test conformance report generated by SCADE Test Environment for Host

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 5.5: DO-331 Table MB.A-5

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

5.4.2 Impact of code generator qualification

The KCG Code Generator is qualified as a Criteria 1 tool because it was developed 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 LLR contained in these models.

Note that if the models are 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 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 models have a 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 models complies with its coding standards. Coding rules for SCADE Suite KCG are 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 models is traceable to Logics LLR 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 models reflects these models accurately and consistently. This evidence is based on the requirements (see TOR document) of KCG that include:

- The verification that the model complies with the syntactic/semantic rules of the input language;
- A code generation scheme ensuring that the generated code reflects the model.

Additional user integration activities are needed to evaluate if the properties of the code are met in the target execution environment such as stack usage, WCET analysis, mathematical analysis for overflow prevention. Such analysis can be confirmed by actual measurements for timing, memory usage, etc.

The objectives listed above are met thanks to KCG qualification, provided that the code was successfully generated by 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 generated code or not.

5.4.3 Verification of parameter data items

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 Achievement

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

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

	Objective Description	Activity Ref
1	Executable Object Code complies with high-level requirements	6.4.2 6.4.2.1 6.4.3 6.5 MB.6.8.2.a (see Item 1)
2	Executable Object Code is robust with high-level requirements	6.4.2 6.4.2.1 6.4.3 6.5 MB.6.8.2.a (see Item 1)
3	Executable Object Code complies with low-level requirements	6.4.2 6.4.2.1 6.4.3 6.5
4	Executable Object Code is robust with low-level requirements	6.4.2 6.4.2.2 6.4.3 6.5
5	Executable Object Code is compatible with target computer	6.4.1.a 6.4.3.a

Item1: As described in section MB.6.8.2.a of the [DO-331] Supplement, 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: test cases and procedures are primarily developed from HLR. 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 test 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:

- Regarding the logics (with SCADE Suite), the same 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 5.9.

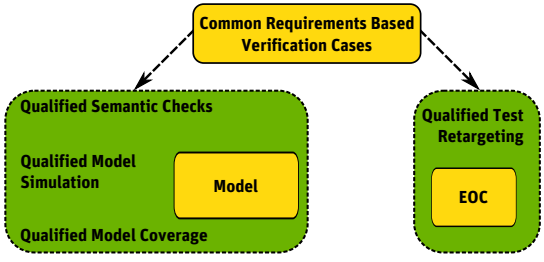


Figure 5.9: Factor simulation and test cases with SCADE Test

- There is no need to develop additional test cases and procedures for Logics LLR that are already covered by HLR-based 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 5.10 provides an overview of the Combined Testing Process.

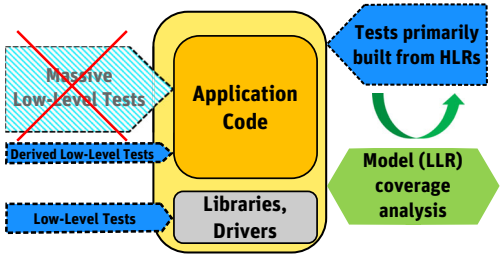


Figure 5.10: Combined Testing Process

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 hand, 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 HLR (MB.A-6 #1) and robustness with HLR (MB.A-6 #2)

Test cases and procedures are developed firstly on the basis of HLR and executed in the target environment. They should include normal range test cases and robustness test cases.

In the context of SCADE Suite, users can reuse existing simulation cases developed for design verification (see Section 5.3.2) with the support of SCADE Test Target Execution as in Figure 5.11.

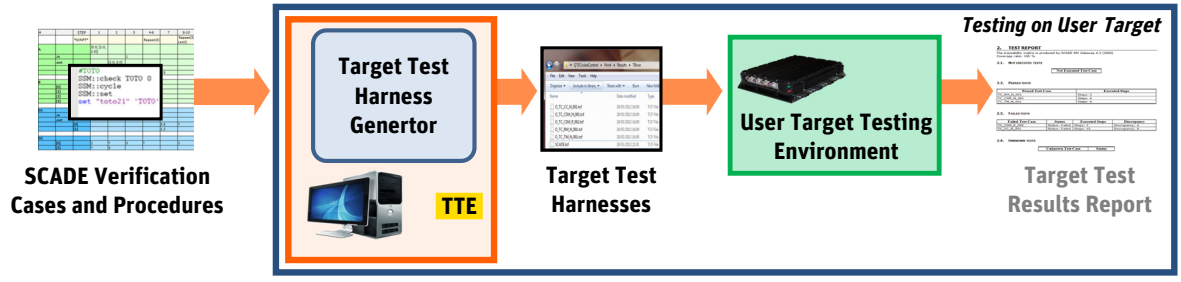


Figure 5.11: Factor simulation and test cases with SCADE Test Target Execution for logics

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

TRACEABLE LLR

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

The assessment of Logics LLR coverage is supported by SCADE Test Model Coverage. Full model coverage with Test Model Coverage is required to ensure compliance of the EOC to traceable LLR as highlighted in Figure 5.10.

This approach is valid and complete in the scope of sequential logic algorithms. In the specific context where the SCADE Suite model includes numerical computations, additional verification activities must be considered:

- For library-based numerical algorithms: SCADE Test Model Coverage additional coverage points can be used to capture user-selected numerical equivalence classes.
- For non-library-based numerical algorithms: There is no efficient way to capture numerical equivalence classes with SCADE Test Model Coverage. Additional equivalence classes analysis (by manual means) is required to ensure the full coverage of numeric computations by the tests.

DERIVED LLR

Additionally, verification objectives of Table MB.A-6 #3 require specific derived-LLR 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 the logics, there are two cases:

1 The derived LLR was implemented as a library operator: the applicant must test the implementation of this operator from the HLR 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 coverage criteria and for coverage of equivalence classes other than those addressed by SCADE Test Model Coverage. For further information on 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 LLR-based testing must be considered in this case.

The objective is to minimize as much as possible the number of non-library-based derived LLR to maximize the benefit of model-based verification.

5.5.5 Robustness of EOC with LLR (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.

Regarding the logics, Figure 4.12 provides a typical example of SCADE Suite robust architecture where low-level robustness can be managed with different non-exclusive techniques for the same application.

The strategy of EOC verification with respect to the robustness aspects depends on this architecture choice as follow:

1 Use of robust library operators: Each library operator is unit-tested according to its associated robust requirements. Then, the verification of the integration of such robust operator 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)

2 Use of non-robust library operators: in this context specific robustness verification activities must be considered at the application level including low-level robustness tests of the generated code.

The usage of robust library operators (solution 1) is highly recommended to apply an optimized model-based verification strategy where the robust components are tested once according their respective HLR.

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 Suite or manually coded) and its scope can be beyond the SCADE 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

	Objective Description	Activity Ref	Verification Method
1	Executable object code complies with high-level requirements	6.4.2 6.4.2.1 6.4.3 6.5 MB.6.8.2.a (see Item 1)	HLR-based testing in the target environment with SCADE Test Target Execution
2	Executable object code is robust with high-level requirements	6.4.2 6.4.2. 6.4.3 6.5 MB.6.8.2.a (see Item 1)	HLR-based robustness testing in the target environment with SCADE Test Target Execution
3	Executable object code complies with low-level requirements	6.4.2 6.4.2.1 6.4.3 6.5	HLR-based testing in the target environment with full model coverage with SCADE Test Target Execution Analysis of model coverage and equivalence classes for non-library-based numerical computations
4	Executable object code is robust with low-level requirements	6.4.2 6.4.2.2 6.4.3 6.5	Library-based robustness testing with SCADE Test Target Execution Complementary robustness low-level testing for functions not based on robust libraries
5	Executable object code is compatible with target computer	6.4.1.a 6.4.3.a	HW/SW integration testing of the whole application

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 6.1: DO-331 Table MB.A-7

	Objective Description	Activity Ref
1	Test procedures are correct	6.4.5
2	Test results are correct and discrepancies are explained	6.4.5
3	Test coverage of high-level requirements is achieved	6.4.4.1 MB.6.8.2.a
4	Test coverage of low-level requirements is achieved	6.4.4.1 MB.6.7
5	Test coverage of software structure (modified condition/decision coverage) is achieved	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)

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

	Objective Description	Activity Ref
6	Test coverage of software structure (decision coverage) is achieved	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)
7	Test coverage of software structure (statement coverage) is achieved	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)
8	Test coverage of software structure (data coupling and control coupling) is achieved	6.4.4.2.c 6.4.4.2.d 6.4.4.3 MB.6.8.2.b
9	Verification of additional code, that cannot be traced to Source Code, is achieved	6.4.4.2.b
MB 10	Simulation cases are correct (see Item 2)	MB.6.8.3.2
MB 11	Simulation procedures are correct (see Item 2)	MB.6.8.3.2
MB 12	Simulation results are correct and discrepancies explained (see Item 2)	MB.6.8.3.2

Item 1: As described in section MB.6.8.2.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:

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

The review of SCADE test procedures must confirm that test cases, including expected results, are correctly developed into test procedures. SCADE LifeCycle Reporter for SCADE Test[12] supports this activity.

As illustrated by Figure 5.9 and Figure 5.11, both simulation and test cases are factorized for the verification of logics 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 simulation cases and procedures were already reviewed during design verification.

The review of test results must confirm:

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

VERIFICATION OF LOGIC TEST RESULTS

Regarding the logics, test results are usually generated by the user target testing environments and include a pass/fail status. Typical testing environment such as RTRT from IBM, TestBed® from LDRA, or VectorCAST® from Vector can be used for the verification of logics on target. These tools support a qualification process at DO-330/TQL-5 and ensure that test results are correctly evaluated.

12.Available from SCADE 2020 R2 onwards.

6.3 HLR Coverage Analysis

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

This is achieved by peer review of HLR test cases traceability matrices.

Regarding the logics, if common requirement-based verification cases and procedures are used for both model simulation on host and testing on target, HLR 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 5.3.2).

6.4 LLR Coverage Analysis

6.4.1 SCADE Test Model Coverage overview

SCADE Test Model Coverage performs model coverage analysis of SCADE Suite models.

SCADE Test Model Coverage takes as inputs a SCADE Suite model and a set of HLR-based test cases and procedures and supports model coverage analysis with so-called code coverage implication with respect to MB.B.11 FAQ#11 [DO-331]. Such implication means that reaching 100% model coverage guarantees 100% code coverage of the SCADE Suite-KCG generated code. The HLR-based test cases

and procedures used for coverage measurement are those previously developed to satisfy Table MB.A-6 objectives #1, #2, #3, and #4. SCADE Test Model Coverage generates a model coverage report and evidence for structural coverage.

Model coverage analysis focuses 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 Logics LLR coverage analysis (MB.A-7#4)

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

In the context of SCADE development, Logics 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 model (representing a LLR) was fully exercised when requirements-based tests are exercised. It supports in particular the detection of unintended functions in the model (see Section 2.4.4 and Section 5.3.2).

6.4.2.1 Logics LLR coverage analysis with Model Coverage

Model Coverage measures the coverage of a model by high-level requirements-based test cases. The purpose of this measure is to assess how thoroughly the model was exercised.

For SCADE Suite models, Model Coverage criteria are based on the observation of how a flow of values is used. For details about criteria, see "Model coverage criteria".

Figure 6.1 shows the position of SCADE Test Model Coverage within the software verification flow.

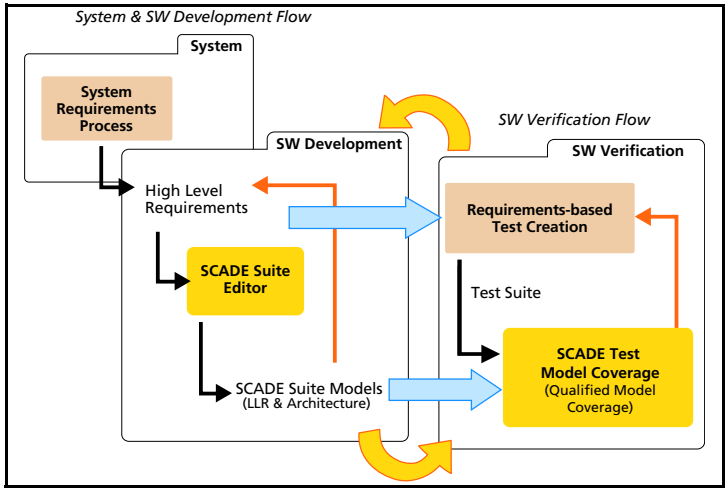
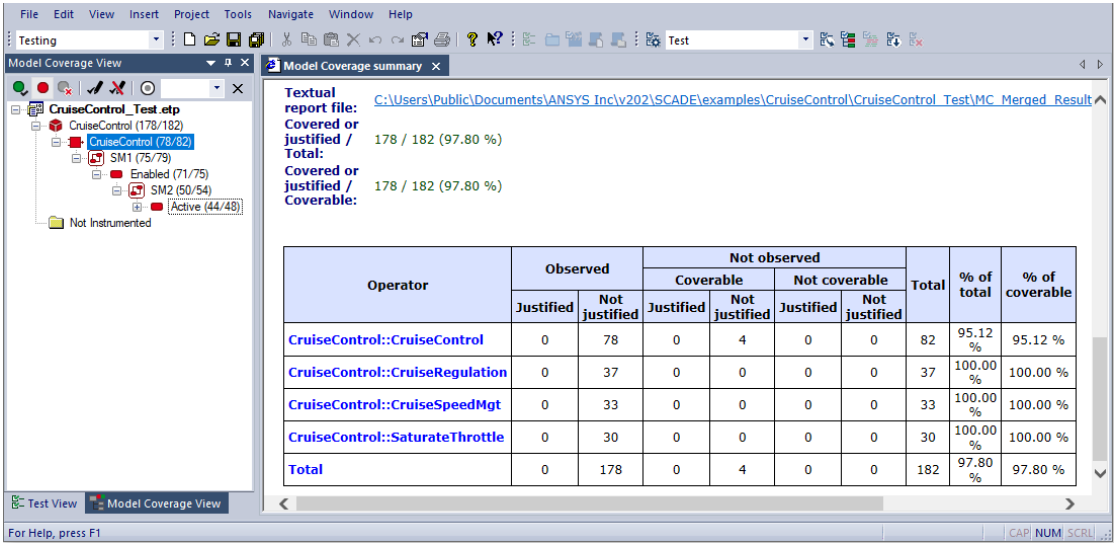


Figure 6.1: Position of SCADE Test Model Coverage within the verification flow

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 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 model, or both.

Figure 6.2 illustrates the use of SCADE Test Model Coverage. The coverage result for each operator and child elements is indicated via colors and coverage ratios about observed coverage points. The tool provides also detailed explanations about operator features that are not fully covered.



Operator	Observed		Not observed				Total	% of total	% of coverable
			Coverable		Not coverable				
	Justified	Not justified	Justified	Not justified	Justified	Not justified			
CruiseControl::CruiseControl	0	78	0	4	0	0	82	95.12 %	95.12 %
CruiseControl::CruiseRegulation	0	37	0	0	0	0	37	100.00 %	100.00 %
CruiseControl::CruiseSpeedMgt	0	33	0	0	0	0	33	100.00 %	100.00 %
CruiseControl::SaturateThrottle	0	30	0	0	0	0	30	100.00 %	100.00 %
Total	0	178	0	4	0	0	182	97.80 %	97.80 %

Figure 6.2: Model coverage analysis/resolution with SCADE Test Model Coverage

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 HLR 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 model: Resolution must be done according to DO-331 MB.6.7.2.d.[13] Moreover, the deactivated functionality should be identified as such in the design.

5 Unintended functionality in model: In that case, resolution consists in removing the functionality and assessing the effects and needs for re-verification

13. 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.

EXAMPLE 1: INSUFFICIENT TESTING

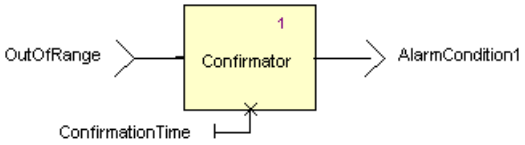


Figure 6.3: Non activated Confirmator

- Analysis: The Confirmator in Figure 6.3 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 6.4 was never reset during the tests. Is the "reset" behavior an unintended function?

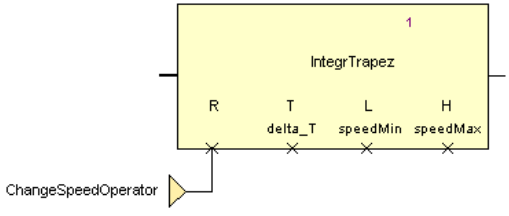


Figure 6.4: Uncovered "reset" activation

- 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 SCADE 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 are exercised by test and/or simulation

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

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

For SCADE Suite models, we use tags to represent coverage points. Model coverage criteria are based on tag propagation and observation through observable outputs of the model. Setting coverage criteria amounts to defining where tags are introduced in the model and what is the semantic of tag propagation to be used for Boolean primitives. For criteria that distinguish Boolean flows (see ODC and OMC/DC), two tags are introduced by the "bool_tag" primitive: one when the flow takes value true and the other when it is false. Each tag introduced in the model is expected to reach an observation point (red circle on output in Figure 6.5). A point is covered if the model is stimulated by an input sequence leading to the observation of the

corresponding tag. The overall coverage measure is the ratio of observed tags to introduced tags.

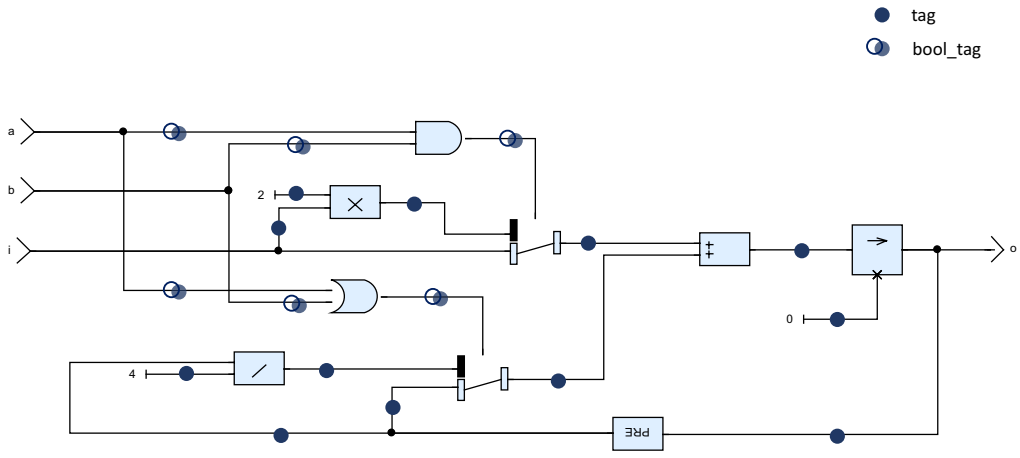


Figure 6.5: Tag propagation and output observation for SCADE Suite model coverage

The model coverage criteria for SCADE Suite are:

1. INFLUENCE

This criterion measures coverage based on tags attached to data flows of the model and on tags related to the activation of scopes introduced by control structures (state machines and conditional activation operators). With this criterion, Boolean primitives behave as any combinatorial primitive by always propagating the tags present on the inputs to the outputs regardless of the actual Boolean value of the streams.

This criterion is the least demanding one: a test suite that covers a model for Influence criterion does not necessarily covers this model for other criteria (ODC or OMC/DC).

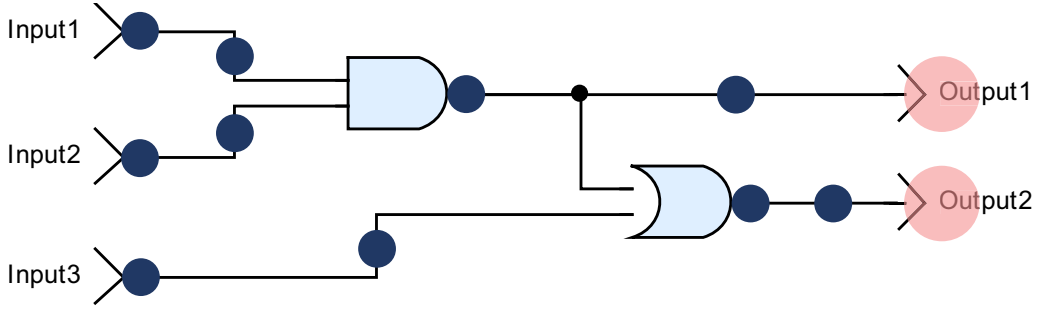


Figure 6.6: Tags and observation for Influence

2. **OBSERVABLE DECISION COVERAGE (ODC)**

This criterion measures coverage based on tags able to distinguish between the influence of True and the influence of False for the monitoring of Boolean flows. With this criterion, the propagation rules for Boolean primitives are the same as for Influence. The semantics of tag propagation of this criteria ignores the MC/DC masking effect of Boolean flows on coverage measurements.

This criterion is intermediary between Influence and OMC/DC: a test suite that covers a model for ODC criterion also covers this model for Influence but does not necessarily cover it for OMC/DC.

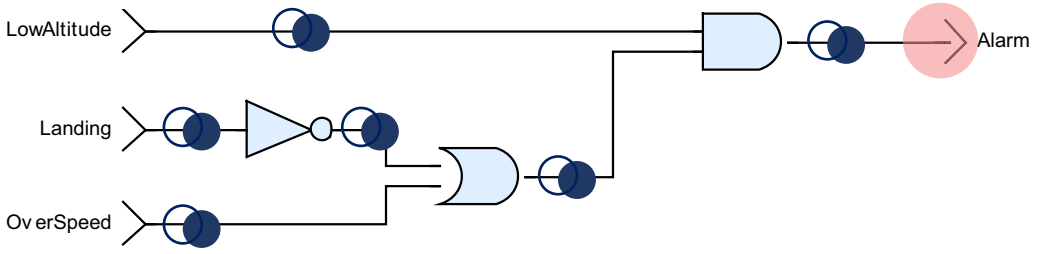


Figure 6.7: Tags and observation for ODC

3. OBSERVABLE MODIFIED CONDITION/DECISION COVERAGE (OMC/DC)

This criterion measures coverage based on the same tags as ODC (see figure above) and a semantics of tag propagation that takes into account the masking effect over coverage measurements.

This criterion is the most demanding one: a test suite that covers a model for OMC/DC also covers this model for both ODC and Influence.

Table 6.2 summarizes all coverage criteria used by SCADE Test Model Coverage.

Table 6.2: Coverage criteria in SCADE Test Model Coverage for SCADE Suite models

Coverage Criterion	Applies to	Synopsis
Influence	Any flow type	All connection points were tested as able to influence an output.
Observable Decision Coverage	Boolean expressions	All connection points were tested as able to influence an output and all Boolean flows have taken both True/False values while influencing an output without taking into account the masking effect of Boolean operators.
Observable Modified Condition/Decision Coverage	Boolean expressions	All connection points were tested as able to influence an output, and all Boolean flows have taken both True/False values while influencing an output by taking into account the masking effect of Boolean operators.

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

6.4.3.1 SCADE Suite generated code coverage analysis

With respect to DO-331 FAQ#11 [DO-331], it is possible to use model coverage to achieve structural code coverage. SCADE Test Model Coverage guarantees implication for structural coverage under certain conditions [MC-FAQ11].

With regard to MB.B.11 FAQ#11 [DO-331], whenever reaching 100% model coverage with SCADE Test Model Coverage and OMC/DC criterion, users can claim 100% code coverage of the SCADE Suite-KCG generated code in the MC/DC sense. This property also holds for other criteria; 100% model coverage with ODC (resp. Influence) criterion guarantees 100% coverage of the code in the DC (resp. Statement Coverage) sense. Model coverage analysis must be performed with the same options as those used to generate the target code with SCADE Suite KCG.

In addition, SCADE Test Model Coverage produces warnings about exceptions in the model that produce unreachable parts in the SCADE Suite KCG-generated code. In such cases, users have to provide justifications or perform complementary activities to achieve structural coverage objectives as detailed in [MC-FAQ11].

DESIGN REVIEW AND ANALYSIS

Data coupling and control coupling is verified first by design review and analysis

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 within models

The qualification of SCADE Suite KCG ensures that data coupling and control coupling at model level are exactly reflected in the generated code.

Regarding the logics with SCADE Suite:

- 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.

with semantic checks using SCADE Suite Checker.

MODEL COVERAGE ANALYSIS

SCADE Test Model Coverage analysis must confirm that 100% of the components control and data coupling structures are exercised by the requirement-based test cases and procedures. Since data and control coupling effects are part of the influence effects (yet not limited to them), the model coverage criteria take into account data and control coupling as part of the assessment of influence. For any C component integrating C1 and C2, data and control coupling of C1 and C2 are assessed by SCADE Test Model Coverage if model coverage is measured at C level. This holds for any of the three criteria: Influence, ODC, or OMC/DC.

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).

Note: For logics, 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 6.3 summarizes verification objectives and methods for the verification of verification process results.

Table 6.3: DO-331 Table MB.A-7 Objectives Achievement

	Objective Description	Activity Ref	Verification Method
1	Test procedures are correct.	6.4.5	Peer review of SCADE Suite test procedures (see Note 1)
2	Test results are correct and discrepancies are explained.	6.4.5	Analysis of test report generated by user target testing environment (report not generated by SCADE tools but by external user targets)
3	Test coverage of high-level requirements is achieved.	6.4.4.1 MB.6.8.2.a	Peer review of HLR test cases and procedures traceability matrices generated by SCADE LifeCycle ALM Gateway
4	Test coverage of low-level requirements is achieved.	6.4.4.1 MB.6.7	Analysis of SCADE Suite model coverage with SCADE Test Model Coverage
5	Test coverage of software structure (modified condition/decision coverage) is achieved.	6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3 MB.6.8.2.b	Analysis of SCADE Suite model/code coverage with SCADE Test Model Coverage (observable modified condition/decision coverage)
6	Test coverage of software structure (decision coverage) is achieved.	6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3 MB.6.8.2.b	Analysis of SCADE Suite model/code coverage with SCADE Test Model Coverage (observable decision coverage)
7	Test coverage of software structure (statement coverage) is achieved.	6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3 MB.6.8.2.b	Analysis of SCADE Suite model/code coverage with SCADE Test Model Coverage (influence)
8	Test coverage of software structure (data coupling and control coupling) is achieved.	6.4.4.2.c 6.4.4.2.d 6.4.4.3 MB.6.8.2.b	Analysis of SCADE Suite model control and data coupling with SCADE Test Model Coverage (influence)
9	Verification of additional code, that cannot be traced to Source Code, is achieved	6.4.4.2.b	Source code to object code traceability analysis Logics: (see Note 2)
MB10	Simulation cases are correct (see Item 2)	MB.6.8.3.2	N/A (see Note 3)

Table 6.3: DO-331 Table MB.A-7 Objectives Achievement (Continued)

	Objective Description	Activity Ref	Verification Method
MB11	Simulation procedures are correct (see Item 2)	MB.6.8.3.2	N/A (see Note 3)
MB12	Simulation results are correct and discrepancies explained (see Item 2)	MB.6.8.3.2	N/A (see Note 3)

Item 1: As described in section MB.6.8.2.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.

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 simulation cases and procedures have already been reviewed during 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 (see Item 2)

Ansys

Appendixes and Index

A / References

[AC 20-115C] Advisory Circular “Airborne Software Assurance”, Federal Aviation Administration, 07/19/2013

[AMC-20-115] "Software Considerations for Certification of Airborne Systems and Equipment", ED Decision 2013/026/R, EASA, 12/09/2013

[ARP-HB] Methodology Handbook, “Efficient Avionics Systems Engineering with ARP-4754A Objectives Using SCADE System®”, Esterel Technologies, Second Ed., 2015.

[ARP 4754A] “Guidelines for Development of Civil Aircraft and Systems”, Society of Automotive Engineers, 2010-12.

[Camus] “A verifiable architecture for multi-task, multi-rate synchronous software”, J. L. Camus, P. Vincent, O. Graff, and S. Poussard. Embedded Real Time Software Conference ERTS 2008, Toulouse, France.

[Caspi] “Integrating model-based design and preemptive scheduling in mixed time and event-triggered systems”, N. Scaife and P. Caspi, Verimag Report Nr. TR-2004-12, June 1, 2004, (see www-verimag.imag.fr).

[CVK-RM] “SCADE Suite CVK Reference Manual,” Esterel Technologies, CVK 6.6, September 2016.

[CVK-UM] “SCADE Suite CVK User Manual,” Esterel Technologies, CVK 6.6, September 2016.

[DO-178C] “Software Considerations in Airborne Systems and Equipment Certification”, RTCA Inc., December 2011.

[DO-330] “Software Tool Qualification Considerations”, RTCA Inc., December 2011.

[DO-331] “Model-Based Development and Verification Supplement to DO-178C and DO-278A”, RTCA Inc., December 2011.

[DO-333] “Formal Methods Supplement to DO-178C and DO-278A”, RTCA Inc., December 2011.

[DO-248C] “Supporting Information for DO-178C and DO-278A”, RTCA Inc., December 2011.

[DO-254] “Design Assurance Guidance for Airborne Electronic Hardware”, RTCA Inc., April 2000.

[ED–79] “Guidelines for Development of Civil Aircraft and Systems”, EUROCAE, December 2010.

[Esterel] “The Foundations of Esterel”, Gérard Berry. In “Proofs, Languages, and Interaction, Essays in Honour of R. Milner,” G. Plotkin, C. Stirling, and M. Tofteed, MIT Press (2000).

[Harel] “Statecharts: a Visual Approach to Complex Systems”, D. Harel. In Science of Computer Programming, vol. 8, pp. 231-274 (1987).

[ISO-Ada] Ada Reference Manual - ISO 8652:1995

[ISO-C] Programming languages - C (ISO/IEC 9899:1990)

[KCG-TOR] “SCADE Suite KCG Tool Operational Requirements”, KCG-SRS-011, Esterel Technologies, March 14, 2016.

[Lustre] “The Synchronous Dataflow Programming Language Lustre”, N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, Proceedings of the IEEE, 79(9):1305-1320, September 1991.

[MC-FAQ11] “Model Coverage for SCADE Suite - FAQ#11: Application Conditions and Property”, MC-SRS-004, 2020-04-06.

[NASA-MCDC] “A Practical Tutorial on Modified Condition/Decision Coverage”, K. J. Hayhurst (NASA), D. Veerhusen (Rockwell Collins), J. J. Chilenski (Boeing), L. K. Rierson (FAA).

[SC-SDVST] "SCADE Suite® Application Software Development Standard", Esterel Technologies, 2017-09-13.

[SCS-KCG-LRM] "The Scade 6 Language", KCG-SRS-007, Esterel Technologies, March 17, 2016.

[SLC-UM] "SCADE LifeCycle User Manual", Version R19.2 release, Esterel Technologies, June 2018.

[SUITE-UM] "SCADE Suite User Manual", Version R19.2 release, Esterel Technologies, June 2018.

[SUITE-TM] "SCADE Suite Technical Manual", Version R19.2 release, Esterel Technologies, June 2018.

[TEST-UM] "SCADE Test User Manual", Version R19.2 release, Esterel Technologies, June 2018.

B / Acronyms and Glossary

ACRONYMS

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

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 and SCADE Verification Tools

C-1 What Does SCADE Suite KCG 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 qualification process is described entirely in §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. Therefore SCADE Suite KCG is classified as a Criteria 1 tool. Achieving 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-1.1 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): shows 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-1.2 SCADE Suite KCG Life-Cycle Documentation

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

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 Ansys.

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

Data	DO-330 Ref.	Certification Kit
Tool Qualification Plan (TQP)	10.1.2	SCADE Suite KCG Tool Qualification Plan
Tool Operational Requirements (TOR)	10.3.1	• Software requirements data of SCADE Suite KCG • Scade 6 Language Reference Manual • SCADE Suite KCG Software Installation Procedure (SIP)
Tool Accomplishment Summary (TAS)	*10.1.15*	SCADE Suite KCG Tool Accomplishment Summary SCADE Suite KCG Compliance Analysis to DO-330 (all levels) SCADE Suite KCG Compliance Analysis to DO-178B (all levels)
Tool Configuration Index (TCI)	10.1.11	SCADE Suite KCG Tool Configuration Index
Tool Life Cycle Environment Configuration Index	10.1.10	SCADE Suite KCG Tool Life Cycle Environment Configuration Index

C-2 SCADE Test Model Coverage at TQL-4 Level

SCADE Test Model Coverage for SCADE Suite allows to measure the coverage of the SCADE Suite model by test cases without the need to verify the tool outputs. Model coverage analysis also allows to assess the thoroughness of simulation of the Low-Level Requirements contained in the model when simulation is used for verification of model compliance to the High-Level Requirements of the application.

Model Coverage is used as a tool supporting the model verification activity. Yet, a malfunction of the tool such as reporting positive coverage for a part of the model that is not covered may lead to not testing parts of the model. Therefore, Model Coverage automates the verification activity and may lead to failure in detecting an error.

While the certification credit of the Model Coverage tool covers the model coverage objective, it also extends to SCADE Suite KCG-generated code structural coverage objective, provided some conditions on models are met excluding some exceptions [MC_FAQ11]. This is worth explaining in details.

As stated in DO-331 FAQ11, model coverage analysis does not eliminate the need to achieve the objectives of structural coverage analysis per DO-178C §6.4.4.2. However, model coverage analysis can be used as a means for achieving structural code coverage analysis under appropriate conditions.

DO-331 FAQ 11 further states the conditions, the first most important one being: "Model coverage analysis criteria hold the same properties as the applicable structural code coverage analysis criteria hold for the level of the software being developed, for example, MC/DC coverage for the level A."

The coverage criteria of Model Coverage (OMC/DC, ODC, Influence) are defined as a correspondence to code coverage criteria (MC/DC, DC, Statement Coverage) in such a way that, when model coverage is achieved for a matching criterion, say OMC/DC, then structural coverage of SCADE Suite KCG 6.6- generated code holds for the corresponding criterion, say MC/DC. In other words, SCADE Suite KCG preserves model coverage, meaning that achieving model coverage is enough to ensure that structural coverage of the generated code is also achieved for matching coverage criteria.

This enables SCADE Test Model Coverage and SCADE Suite KCG to meet the DO-331 FAQ11 condition to use Model coverage as a means to also ensure structural coverage of the SCADE Suite KCG-generated code.

As a consequence, Model Coverage is a Criteria 2 tool as defined in DO-178C §12.2.2, since Model Coverage automates the verification process (*i.e.,* model coverage), and Model Coverage output (*i.e.,* coverage objectives achievement measure) is used to justify the elimination of the verification process other than that automated by the tool (*i.e.,* structural coverage). DO-178C Table 12-1 provides the required Tool Qualification Level (TQL) according to the application software level. TQL-4 is required for applicability to DAL A projects, therefore SCADE Test Model Coverage is qualified to TQL-4 Tool Qualification requirements of DO-330.

C-3 SCADE Test Environment for Host and SCADE Test Target Execution at TQL-5 Level

SCADE Test Execution for Host and SCADE Test Target Execution are used to automate test execution and perform automatic checks to determine if tests are passed.

An error in these tools may result in reporting a test as passed when it should not, which can result in failure to detect an error in SCADE models. Therefore these tools are Criteria 3 tools as defined in DO-178C § 12.2.2, since they automate a verification process and could fail to detect an error. Table 12-1 of DO-178C provides the required Tool Qualification Level (TQL) according to the application software level. TQL-5 is required for applicability to DAL A projects, therefore SCADE Test Execution for Host and SCADE Test Target Execution are qualified to TQL-5 Tool Qualification requirements of DO-330.

C-1 SCADE LifeCycle Reporter at TQL-5 Level

SCADE Lifecycle Reporter is not designed as a tool to directly detect an error in SCADE models, but it is used to support the SCADE model review activity. Since the review activity is performed to detect errors in the model being developed, a malfunction of SCADE Lifecycle Reporter like for example failing to report some SCADE operators in the report, may lead to the reviewer not reviewing part of the model and, as a consequence, failing to detect an error in the resulting software. This is why, although indirectly, we consider that SCADE Lifecycle Reporter may “fail to detect” an error. Table 12-1 of DO-178C provides the required Tool Qualification Level (TQL) according to the application software level. TQL-5 is required for applicability to DAL A projects, therefore SCADE Lifecycle Reporter is qualified to TQL-5 Tool Qualification requirements of DO-330.

D / 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 LLR 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.

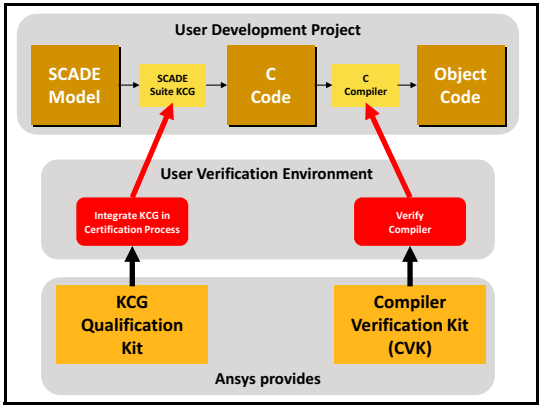


Figure D.1: Role of KCG and CVK in verification of user development environment

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:

1. A CVK User's Manual [CVK_UM] and a Reference Manual [CVK_RM] containing:
   - Installation and user instructions
   - Description of the underlying methodology
   - 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 Suite 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 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 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.

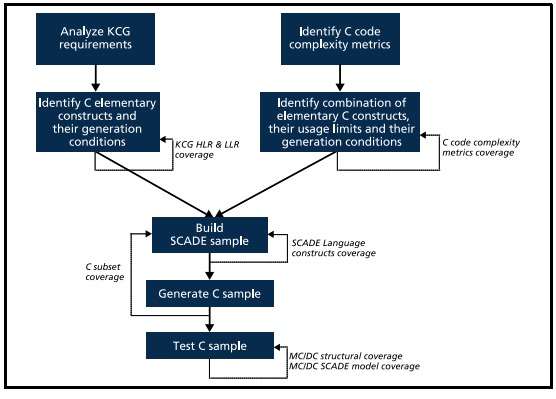


Figure D.2: Strategy for developing and verifying CVK

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 level 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 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 model.

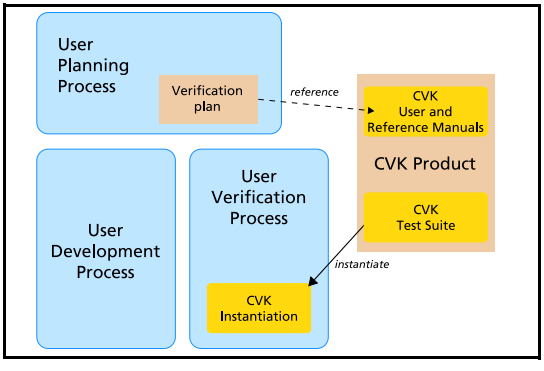


Figure D.3: Use of CVK items in user processes

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 Suite 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 model).

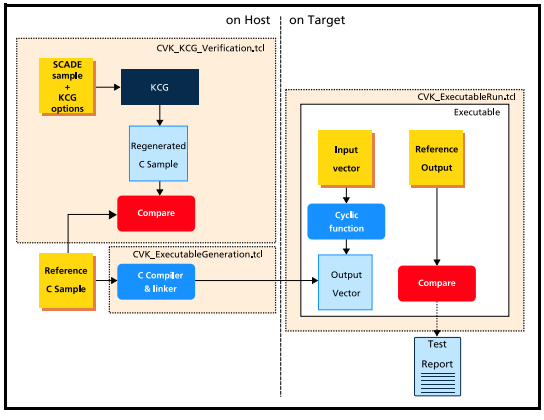


Figure D.4: Position of CVK items in the Compiler Verification Process

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

Index

Contact Information

Submit questions to SCADE Products Technical Support at
scade-support@ansys.com

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

Direct general questions about SCADE products to
scade-info@ansys.com

Discover the latest news on our products at
http://www.ansys.com

[SC-HB-DO178C-KCG66 - SecondEd]