Methodology Handbook

Efficient Development of Safe Avionics Display Software with DO-178B Objectives Using Esterel SCADE®

Third Edition (Revision 1)
CONTACTS

Headquarters

Esterel Technologies SA
Parc Euclide - 8, rue Blaise Pascal
78990 Elancourt
FRANCE
Phone: +33 1 30 68 61 60
Fax: +33 1 30 68 61 61

Submit questions to Technical Support at support@esterel-technologies.com.
Contact one of our Sales representatives at sales@esterel-technologies.com.
Direct general questions about Esterel Technologies to info@esterel-technologies.com.
Discover latest news on our products and technology at www.esterel-technologies.com.

LOCAL SUPPORT SITES

Northern Europe

Esterel Technologies
8 Hester Road B45
SW11 4AP London
UNITED KINGDOM
Phone: +44 7917 114 451

Southern Europe

Esterel Technologies SA
Park Avenue - 9, rue Michel Labrousse
31100 Toulouse
FRANCE
Phone: +33 5 34 60 90 50

Central Europe

Esterel Technologies GmbH
Otto-Hahn-Strasse 13b
Ottobrunn - Riemerling
D- 85521 München
GERMANY
Phone: +49 89 608 75530

China

Esterel Technologies
1303, Jiaxing Mansion
877, Dongfang Road
200122, Shanghai
P.R. CHINA
Phone: +86 21 61050287

Americas

Phone: +1 203 910 3993
Contact us at infoUS@esterel-technologies.com.

Shipping date: February 2012 Revision: SDY-HB-DO178B - SDY/u1-KCG62-Rev1
Abstract

This handbook addresses the issue of cost and productivity in the development of safe embedded software for avionics display applications. Such projects, driven by the DO-178B guidelines, traditionally require very difficult and precise development tasks, incurring high verification efforts. The handbook reviews the regulatory guidelines and then presents the optimization of the development and verification processes that can be achieved with the SCADE methodology and tools. Esterel SCADE supports the automated production of a large part of the development life-cycle elements. The effect of using SCADE Display® and SCADE Suite® together with their qualified SCADE Suite KCG™ 6.1.3 and SCADE Display KCG™ 6.2 Code Generators is presented in terms of savings in the development and verification activities, following a step-by-step approach and considering the objectives that have to be met at each step.
Table of Contents

1. Document Background, Objectives, and Scope 1
   1.1 Background 1
   1.2 Objectives and Scope 1

2. Development of Safety-Critical Airborne Software 3
   2.1 ARP 4754/5056 and DO-178B Guidelines 3
      2.1.1 Introduction 3
      2.1.2 ARP 4754 3
      2.1.3 ARP 5056 3
      2.1.4 DO-178B 4
      2.1.5 Relationship between ARP 4754 and DO-178B 4
      2.1.6 Development assurance levels 5
      2.1.7 Objective-oriented approach 5
      2.1.8 DO-178B processes overview 6
   2.2 DO-178B Verification Processes 8
      2.2.1 Objectives of software verification 8
      2.2.2 Reviews and analyses of the high-level requirements 8
      2.2.3 Reviews and analyses of the low-level requirements 9
      2.2.4 Reviews and analyses of the source code 9
      2.2.5 Software testing process 10
   2.3 What Are the Main Challenges in the Development of Airborne Display Software? 12
      2.3.1 Avoiding multiple descriptions of the software 12
      2.3.2 Preventing ambiguity and lack of accuracy in specifications 12
      2.3.3 Avoiding low-level requirements and coding errors 12
      2.3.4 Allowing for an efficient implementation of code on target 13
      2.3.5 Finding specification and design errors as early as possible 13
      2.3.6 Lowering the complexity of updates 13
      2.3.7 Improving verification efficiency 13
      2.3.8 Providing an efficient way to store Intellectual Property (IP) 14

3. Model-Based Development of Displays with Esterel SCADE 15
   3.1 What is Esterel SCADE? 15
   3.2 Esterel SCADE Modeling Techniques 18
      3.2.1 Modeling the graphics with SCADE Display 18
      3.2.2 Modeling the behavior with SCADE Suite 18
# Table of Contents

3.2.2.1 Familiarity and accuracy reconciled ........................................ 18  
3.2.2.2 Scade operator ............................................................................. 19  
3.2.2.3 Data flow diagrams for continuous control .............................. 20  
3.2.2.4 State Machines for discrete control ........................................ 22  
3.2.2.5 Combining data and control flows ......................................... 22  
3.2.2.6 Scade data typing ...................................................................... 23  
3.2.3 Esterel SCADE cycle-based intuitive computation model .......... 24  
3.2.4 Esterel SCADE as a model-based development environment .... 25  
3.2.5 SCADE Display and SCADE Suite modeling and safety benefits 27  

4. Software Development Activities with Esterel SCADE .................. 29  
4.1 Overview of Software Development Activities .............................. 29  
4.2 Software Requirements Process with Esterel SCADE ................. 31  
4.2.1 Graphics high-level requirements ............................................. 31  
4.2.2 Logics high-level requirements .................................................. 32  
4.3 Software Design Process with Esterel SCADE ............................ 33  
4.3.1 Architecture design ..................................................................... 34  
4.3.2 SCADE low-level requirements development ......................... 35  
4.3.3 Code generation from SCADE Display model ......................... 39  
4.3.4 Code generation from SCADE Suite models ............................ 40  
4.3.5 Integration aspects ...................................................................... 43  
4.3.6 Input/output ................................................................................ 43  
4.3.7 Integration of SCADE Suite and SCADE Display generated code 43  
4.3.8 Integration of external data and code ....................................... 43  
4.3.9 SCADE scheduling and tasking .................................................. 44  
4.3.10 Integration of the SCADE Display generated code with the embedded graphics library 47  
4.4 Teamwork ....................................................................................... 48  

5. Software Verification Activities ....................................................... 51  
5.1 Overview ........................................................................................ 51  
5.2 Verification of the SCADE High-Level Requirements ............... 51  
5.2.1 Verification objectives for HLR ................................................. 51  
5.2.2 Verification methods for HLR ..................................................... 52  
5.2.3 Verification summary for HLR .................................................... 53  
5.3 Verification of the SCADE Low-Level Requirements and Architecture 54  

Methodology Handbook  
ii
Table of Contents

5.3.1 Verification objectives for the LLR and architecture 54
5.3.2 Scade model accuracy and consistency 54
5.3.3 Compliance with design standard 55
5.3.4 Traceability from SCADE LLR to HLR 55
5.3.5 Verifiability 56
5.3.6 Compliance with high-level requirements 57
5.3.7 Compatibility with target computer 61
5.3.8 Partitioning 66
5.3.9 Verification summary for LLR and architecture 66
5.3.10 Verification objectives for coding output and integration process 68
5.3.11 Impact of code generators qualification 68
5.3.12 Verification summary for coding output and integration process 69

5.4 Verification of Outputs of the Integration Process 69
5.4.1 Verification objectives of the outputs of the integration process 69
5.4.2 Combined Testing Process 70
5.4.3 Verification summary for the outputs of the integration process 72

6. Verification of the Verification Activities 73
6.1 Verification Objectives 73
6.2 Verification of Test Procedures and Test Results 73
6.3 HLR Coverage Analysis 73
6.4 LLR Coverage Analysis with SCADE Suite MTC 74
   6.4.1 Objectives of model coverage 74
   6.4.2 LLR coverage analysis with SCADE Suite MTC 75
   6.4.3 Model coverage criteria 79
6.5 Structural Coverage of the Source Code 84
   6.5.1 Control structure coverage 84
6.6 Data and Control Coupling 84
   6.6.1 Definitions 84
   6.6.2 Control Coupling 85
      6.6.2.1 Control coupling between SCADE Modules 85
   6.6.3 Data Coupling 85
      6.6.3.1 Data coupling between SCADE Modules 85
      6.6.3.2 Data coupling between SCADE Suite and SCADE Display-generated code 86
      6.6.3.3 Data coupling between SCADE Suite generated code and manual code 86
# Table of Contents

6.6.3.4 Control coupling between SCADE Suite and SCADE Display-generated code 86  
6.6.3.5 Control coupling between SCADE Suite-generated code and manual code 86  
6.7 Summary of Verification of Verification 87  

## Appendixes and Index

A References 91  
B Acronyms and Glossary 93  
C DO-178B Qualification of SCADE Display KCG and SCADE Suite KCG 97  
\- What Does Qualification Mean and Imply? 97  
\- Development of SCADE Display KCG and SCADE Suite KCG 97  
\- SCADE Display KCG and SCADE Suite KCG Life-Cycle Documentation 98  
\- The OGLX library for OpenGL SC 99  
D SCADE Suite Compiler Verification Kit (CVK) and User Context Verification Kit (UCVK) 101  
\- Compiler Verification Kit (CVK) 101  
\- CVK Product Overview 101  
\- Motivation for Sample-Based Testing 102  
\- Strategy for Developing SCADE Suite CVK 103  
\- Use of SCADE Suite CVK 104  
\- User Context Verification Kit (UCVK) 105  
\- UCVK Overview 105  
\- Motivation for Sample-Based Testing 106  
\- Use of UCVK 106  
INDEX 109
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>Relationship between ARP 4754 and DO-178B processes</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>DO-178B life-cycle processes structure</td>
<td>6</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>DO-178B development processes</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>DO-178B testing processes</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>SCADE addresses the applicative part of software</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>A software engineering view</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Graphics in SCADE Display Editor</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Plugged objects in a symbology layer</td>
<td>18</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Graphical notation for an integrator operator</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Sample of SCADE Suite model data flows from a Flight Control system</td>
<td>20</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Detection of a causality problem</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Functional expression of concurrency in SCADE Suite</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.9</td>
<td>Detection of a flow initialization problem</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>Initialization of flows</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>SCADE State Machine</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.12</td>
<td>Mixed data and control flows from Flight Control</td>
<td>23</td>
</tr>
<tr>
<td>Figure 3.13</td>
<td>Cycle-based execution model of Esterel SCADE</td>
<td>24</td>
</tr>
<tr>
<td>Figure 3.14</td>
<td>Model-based development with SCADE Display and SCADE Suite</td>
<td>26</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Software development processes with SCADE Display and SCADE Suite</td>
<td>29</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Development of high and low-level requirements with SCADE Display/SCADE Suite</td>
<td>30</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>The software design process with SCADE Display</td>
<td>31</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Layer structure in SCADE Display</td>
<td>32</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Hierarchy structure in SCADE Display</td>
<td>32</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Top-level view of a simple Flight Control System</td>
<td>33</td>
</tr>
</tbody>
</table>
List of Figures

Figure 4.7: The software design process with SCADE Display and SCADE Suite 34
Figure 4.8: Inserting a Confirmator in a Boolean input flow 36
Figure 4.9: Inserting a Limiter in an output flow 36
Figure 4.10: A first order filter 36
Figure 4.11: Complex display logic based on simple functions 37
Figure 4.12: SCADE State Machine in Flight mode management 37
Figure 4.13: The software coding process with SCADE Display and SCADE Suite 39
Figure 4.14: SCADE Display graphics to generated C source code traceability 40
Figure 4.15: Scade data flow to generated C source code traceability 41
Figure 4.16: SCADE State Machine to generated C source code traceability 41
Figure 4.17: Sample of kcg_trace.xml file for traceability between Scade model and source code 42
Figure 4.18: Comparing Call and Inline modes 42
Figure 4.19: SCADE execution semantics 44
Figure 4.20: SCADE code integration 45
Figure 4.21: Modeling a bi-rate system 46
Figure 4.22: Timing diagram of a bi-rate system 46
Figure 4.23: Modeling distribution of the slow system over four cycles 46
Figure 4.24: Timing diagram of distributed computations 46
Figure 4.25: A typical architecture for the graphics platform 48
Figure 4.26: Typical teamwork organization 49
Figure 5.1: Traceability between SCADE LLR and HLR using SCADE LifeCycle RM Gateway 56
Figure 5.2: Animation of a SCADE Display Specification 57
Figure 5.3: Co-Simulation enables run-time visualization of the software specification 59
Figure 5.4: Observer operator containing landing gear safety property 60
List of Figures

Figure 5.5: Connecting the observer operator to the landing gear controller 60
Figure 5.6: Design Verifier workflow 61
Figure 5.7: Timing Verifier integration in SCADE Suite 62
Figure 5.8: Timing and Stack Verifiers analysis stages 63
Figure 5.9: Timing and Stack Verifiers analysis visualization results 64
Figure 5.10: Timing Verifier analysis reports 65
Figure 5.11: The Combined Testing Process with SCADE Display KCG and SCADE Suite KCG 71
Figure 6.1: Position of SCADE Suite Model Test Coverage (MTC) 76
Figure 6.2: Using SCADE Suite Model Test Coverage (MTC) 77
Figure 6.3: Non activated Confirmator 78
Figure 6.4: Uncovered “reset” activation 78
Figure 6.5: Sources of unintended functions in a traditional process 78
Figure 6.6: Elimination of unintended functions with SCADE Suite MTC and KCG 79
Figure 6.7: Data flow diagram with clocks 80
Figure 6.8: Paths in a SCADE data flow diagram 81
Figure 6.9: Covering a state machine 81
Figure 6.10: Activation Conditions with selectors 82
Figure 6.11: MC/DC Coverage of a Boolean expression 83
Figure D.1: Role of KCG and CVK in verification of user development environment 101
Figure D.2: Strategy for developing and verifying CVK 103
Figure D.3: Use of CVK items in user processes 104
Figure D.4: Position of CVK items in the Compiler Verification Process 105
Figure D.5: Role of KCG and UCVK in verification of user development environment 106
List of Tables

Table 2.1: Example of test cases 11
Table 3.1: Components of Scade functional modules: operators 19
Table 5.1: DO-178B Table A-3 51
Table 5.2: DO-178B Table A-3 Objectives Achievement 53
Table 5.3: DO-178B Table A-4 54
Table 5.4: DO-178B Table A-4 Objectives Achievement 66
Table 5.5: DO-178B Table A-5 68
Table 5.6: DO-178B Table A-5 Objectives Achievement 69
Table 5.7: DO-178B Table A-6 69
Table 5.8: DO-178B Table A-6 Objectives Achievement 72
Table 6.1: DO-178B Table A-7 73
Table 6.2: MC/DC Activation Conditions for Boolean Operators 83
Table 6.3: DO-178B Table A-7 Objectives Achievement 87
Table C.1: Documents required for KCG qualification audit by Certification Authorities 98
1. Document Background, Objectives, and Scope

1.1 Background

Currently, numerous people play a role in defining and creating safety-critical systems for the avionics display industry. The function and architecture of a system are defined by system engineers using some informal notation for the graphics and the logic associated with the displays. The embedded production software is then specified textually and hand-coded by software engineers in the coding language augmented by a graphical library.

In this context, qualified code generation from formal models is a technology that may carry strong Return On Investment (ROI), while preserving the safety of the application. Basically, the idea is to describe the application through a software model, including the graphics and the associated logic, and to automatically generate the code from this model using a qualified code generator, in the sense of DO-178B, resulting in the following advantages to the development life cycle:

- When a proper modeling approach is defined:
  - It fulfills the needs of the system engineers by providing notations to formally define the system including its behavior and graphical layout.
  - It fulfills the needs of the software engineers by supporting the accurate definition of the software requirements and by providing efficient automatic code generation of software having the qualities that are expected for such applications (i.e., efficiency, determinism, static memory allocation, etc.).
- It allows for establishing efficient new processes to ensure that safety criteria 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 models.
- 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 the system and software life cycles as described in the ARP 4754 [ARP 4754] and DO-178B [DO-178B] guidelines. It then explains how the use of both proper modeling techniques and qualified code generation from models can drastically improve productivity. It is organized as follows:

**Section 2.** This section provides an introduction to the regulatory guidelines of ARP 4754 and DO-178B used when developing embedded avionics systems and software. It then describes
the main challenges in the development of safety-critical display applications, in terms of specification, verification, and efficiency of the resulting software.

Section 3. This section presents an overview of Esterel SCADE’s methodology and tools, including how Esterel SCADE achieves the highest-quality standards while reducing costs thanks to model-level verification and the use of qualified automatic code generators, 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, including pilots.
• The early identification of most remaining design errors makes it possible to fix them in the requirements/design phase rather than in the code testing or integration phase.
• Qualified code generation not only saves writing the code by hand, but also the cost of verifying it.

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

Section 5, and Section 6. These sections present the verification activities that take place when using Esterel SCADE, including model-level verification with SCADE Suite Simulator and SCADE Display Animator, Design Verifier (DV), and Model Test Coverage (MTC) tools, Timing and Stack Verifiers, as well as specific verification activities aimed at detecting compiler errors and errors of the graphical environment.

This document also contains in appendix:

Appendix A provides a reference list.
Appendix B lists all acronyms used in this document and explains key terminology in a glossary.
Appendix C details the KCG Code Generators qualification process.
Appendix D details the Compiler Verification Kit (CVK) and the User Context Verification Kit (UCVK) for the graphics part.

Note that the content of this document applies to the following products:

• SCADE Suite 6.x, SCADE Suite KCG 6.1.3, and SCADE Suite CVK 6.1.3
• SCADE Display 6.x and SCADE Display KCG 6.2

1. Esterel SCADE is made of the SCADE Suite, SCADE System, SCADE Display, and SCADE LifeCycle toolsets.
2. Development of Safety-Critical Airborne Software

2.1 ARP 4754/5056 and DO-178B Guidelines

2.1.1 Introduction

The avionics industry requires that safety-critical software be assessed according to strict certification authority guidelines before it may be used on any commercial airliner. ARP 4754 and DO-178B are guidelines used both by the companies developing airborne equipment and by the certification authorities. ARP 5056 provides guidelines on the specific processes used in the design of flight decks.

2.1.2 ARP 4754

ARP 4754 was defined in 1996 by the Society of Automotive Engineers (SAE) that also produces aerospace standards.

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.

ARP 4754 addresses the total life cycle for systems that implement aircraft-level functions. It excludes specific coverage of detailed systems, 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 DO-178B (RTCA)/ED-12B (EUROCAE) document [DO-178B/ED-12B]. Coverage of complex hardware aspects of design are dealt with in document DO-254 (RTCA) / ED-80 (EUROCAE) [DO-254/ED-80].

2.1.3 ARP 5056

ARP 5056 was defined in 2006 by the SAE. This document discusses “the recommended flight crew interface design processes and methods for new flight deck designs as well as modifications to the flight crew interface of transport category aircraft”.

2. For example, the United States Federal Aviation Administration (FAA), the European Aviation Safety Agency (EASA), Transport Canada, Brazil’s Agência Nacional de Aviação Civil (ANAC), Civil Aviation Administration of China (CAAC), etc.
The processes and methods that are described here address the integration of human factors/ergonomics, engineering and flight operations in these designs in a way that human engineering principles are "designed in" rather than "reviewed in".

Advisory Circular AC 25-11A - Electronic Flight Deck Displays was defined in 2007 by the FAA and it provides further guidance for the design, installation, integration, and approval of electronic flight displays, components, and systems installed in transport category aircraft in order to meet requirements from the FAA.

2.1.4 DO-178B

DO-178B/ED-12B was published in 1992 by RTCA (Radio Technical Commission for Aeronautics) and EUROCAE (a non-profit organization addressing aeronautic technical problems). It was written by a group of experts from aircraft and aircraft equipment manufacturing companies and from certification authorities. It provides guidelines for the production of software for airborne systems and equipment. The objective of the guidelines is to ensure that software performs its intended function with a level of confidence in safety that complies with airworthiness requirements.

These guidelines specify:

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

2.1.5 Relationship between ARP 4754 and DO-178B

ARP 4754 and DO-178B are complementary guidelines:

- ARP 4754 provides guidelines for the system-level processes.
- DO-178B provides guidelines for the software life-cycle processes.

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

![Figure 2.1: Relationship between ARP 4754 and DO-178B processes](image)

ARP 4754 (§A.2.1) identifies the relationships with DO-178B in the following terms:
“The point where requirements are allocated to hardware and software is also the point where the guidelines of this document transition to the guidelines of DO-178B (for airborne software), DO-254 (for airborne electronic hardware), and other existing industry guidelines. The following data is passed to the software and hardware processes as part of the requirements allocation:

a) Requirements allocated to hardware.
b) Requirements allocated to software.
c) Development assurance level for each requirement and a description of associated failure condition(s), if applicable.
d) Allocated failure rates and exposure interval(s) for hardware failures of significance.
e) Hardware/software interface description (system design).
f) Design constraints, including functional isolation, separation, and partitioning requirements.
g) System validation activities to be performed at the software or hardware development level, if any.
h) System verification activities to be performed at the software or hardware development level.”

ARP 4754 and DO-178B define in common five “Development Assurance Levels” as summarized in the following table:

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

2.1.6 Development assurance levels

ARP 4754 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.

2.1.7 Objective-oriented approach

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

The general approach is the following:

- Ensure appropriate goals are defined. For instance:
  a) Development assurance level of the software.
b Design standards.

- Define procedures for the verification of these goals. For instance:
  a Verify that independence of activities matches the development assurance 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 Remarks of document reviews are answered.
  b Coverage of requirements by testing is achieved.

2.1.8 DO-178B processes overview

DO-178B structures activities as a hierarchy of “processes”, as illustrated in Figure 2.2. The term “process” appears several times in the document. DO-178B 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. These processes are beyond the scope of this handbook.
- 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 throughout the software life cycle.

Figure 2.2: DO-178B life-cycle processes structure

In the remainder of this document, we focus on the development and verification processes.
2.2 DO-178B Development Processes

The software development processes, as illustrated below in Figure 2.3, are composed of:

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

The high-level software requirements (HLR) are produced directly through analysis of system requirements and system architecture and their allocation to software.

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

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

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

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

At all stages traceability is required: between system requirements and HLR; between HLR and LLR; between LLR and source code; and also between tests and requirements.

| 2.3 | DO-178B 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-178B 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 generally performed once, whereas analyses are more detailed and should be reproducible (e.g., compliance with coding standards).

Verification activities cover all the processes, from the planning process to the development processes; there are even verifications of the verification activities.

2.3.2 Reviews and analyses of the high-level requirements

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

a Compliance with the system requirements.

b Accuracy and consistency: each HLR is accurate, unambiguous and sufficiently detailed; requirements do not conflict with each other.

c Compatibility with target computer.

d Verifiability: each HLR has to be verifiable.

e Compliance with standards as defined by the planning process.

f Traceability with the system requirements.

g Algorithm accuracy.
### 2.3.3 Reviews and analyses of the low-level requirements

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

- **Compliance with high-level requirements**: the software LLRs satisfy the software HLRs.
- **Accuracy and consistency**.
- **Compatibility with the 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 HLRs were taken into account in the development of the LLRs.
- **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 SW architecture is compatible with the HLR**, is consistent and compatible with the target computer, is verifiable, and conforms to standards.
- **Software partitioning integrity** is confirmed.

### 2.3.4 Reviews and analyses of the source code

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

- **Compliance with the low-level requirements**: the source code is accurate and complete with respect to the software LLRs; no source code implements an undocumented function.
- **Compliance with the 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.

e **Traceability**: the source code implements all software LLRs.

f **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 avionics 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.

As shown in Figure 2.4, DO-178B dictates that all test cases, including low-level test cases, be requirements-based; namely that all test cases be defined from the requirements and never from the code.

**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.

2. Structural coverage analysis determines which code structures were exercised by the requirements-based test procedures. The main purpose of this step is to verify that only the requirements have been implemented; for instance, there are no unintended functions in the implementation (DO-248B/ED-94B, FAQ #43). Note that requirements coverage is an absolute prerequisite to this step.

---

Figure 2.4: DO-178B testing processes
STRUCTURAL COVERAGE RESOLUTION
If structural coverage analysis reveals structures 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 dead code (it cannot be executed, and its presence is an error), then this code should be removed and an analysis performed to assess the effect and the needs for reverification.
- 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 executed are prevented, isolated, or eliminated.
  - If it is only executed in certain configurations, the operational configuration for execution of this code should be established and additional test cases should be developed to satisfy coverage objectives.

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

- Level C: Statement Coverage is required; this means that every statement in the program was exercised.
- Level B: Decision Coverage is required; this means that 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”) and that every entry and exit point in the program was invoked at least once.
- Level A: MC/DC (Modified Condition/Decision Coverage) is required; this means that:
  - every entry and exit point in the program was invoked at least once;
  - every decision has taken all possible outcomes;
  - each condition in a decision was shown to independently affect that decision’s outcome (this may be shown by varying just that condition, while holding fixed all other possible conditions).

For instance, the following fragment requires four test cases, as shown below in Table 2.1.

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

Table 2.1: Example of test cases

<table>
<thead>
<tr>
<th>Case</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>2</td>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>3</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>4</td>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
2.4 What Are the Main Challenges in the Development of Airborne Display Software?

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

2.4.1 Avoiding multiple descriptions of the software

In such a process, software development is divided into several phases with their outputs. Each phase produces respectively the following:

- Software high-level requirements (HLR)
- Software architecture design and low-level requirements (LLR)
- Software source code

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-178B], is to detect the errors introduced during transformations from one written form to another.

2.4.2 Preventing ambiguity and lack of accuracy in specifications

Requirements and design specifications are traditionally written in some natural language, possibly complemented by non formal figures, diagrams, and pictures for displays. 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”?

2.4.3 Avoiding low-level requirements and coding errors

Coding is the last transformation in a traditional development life cycle. It takes as input the last formulation in natural language (or pseudo-code).

Since programmers have generally a limited understanding of the system, they are sensitive to ambiguities in the specification. Moreover, the code they produce is generally not understandable by the author of the system or software high-level requirements.
In the traditional approach, the combined risk of interpretation errors and coding errors is so high that a major part of the software life-cycle’s verification effort is consumed by code testing.

2.4.4 Allowing for an 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, both for the CPU (Central Processing Unit) and GPU (Graphics Processing Unit). It should easily and efficiently be retargetable to a given processor.

2.4.5 Finding 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.

2.4.6 Lowering the 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.

2.4.7 Improving verification efficiency

The level of verification for safety-critical airborne display software is much higher than for other non safety-critical commercial software. For Level A 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.
2.4.8 Providing an 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
3. Model-Based Development of Displays with Esterel SCADE

3.1 What is Esterel SCADE?

SCADE ORIGIN AND APPLICATION DOMAIN

Esterel SCADE is a product line that includes the following products:

- SCADE Display for the design of embedded displays graphics;
- SCADE Suite for the design of displays logics;
- SCADE System for system design;
- SCADE LifeCycle for the application lifecycle 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. SCADE Display has its own notation which is different from the Scade notation.

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 the full set of expressiveness needed to model any safety-critical application.

From its early industrial stages, SCADE Suite and SCADE Display were developed in conjunction with actual companies developing safety-critical software. They were used on an industrial basis for the development of safety-critical aeronautics display software, including Cockpit Display Systems (CDS), Head Up Displays (HUD), Flight Management Displays (FMD), Airport Navigation Systems (ANS), Cartographic Systems (Digital Maps), Traffic & Collision Avoidance Displays (TCAS), or Helmet Mounted Displays (HMD), by companies like Airbus, Thales, Sukhoi or Sikorsky.

SCADE addresses the applicative part of display software, including graphics and associated logic, as illustrated in Figure 3.1. This is usually the most complex and changeable aspect of software. It typically represents 60 percent to 80 percent of the software embedded in an airborne display computer.
Figure 3.1: SCADE addresses the applicative part of software dealing with the joint development of graphics with SCADE Display and the associated logic with SCADE Suite.

A BRIDGE BETWEEN SYSTEM ENGINEERING AND SOFTWARE ENGINEERING

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

- Systems engineers typically describe display systems using natural language descriptions, pictures and some informal notation for describing architecture blocks input/output. They may use semi-formal data-flows and state machines to express the logics associated with displays. They may use some informal pictures to describe the graphics of these displays.
- Software engineers describe their programs in terms of tasks, flow charts, and algorithms, as shown below in Figure 3.2.

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

To address this problem, Esterel SCADE offers rigorous software constructs that reflect system engineering constructs:

- SCADE Display allows a full description of the graphical part of the display system. SCADE Display Editor relies on a tree-structured graphical description of a display, as illustrated on Figure 3.3.
SCADE Suite provides notations for describing the logic associated with the display graphics. For example, a SCADE Suite model computes the rotation angle of an object that is described as part of a SCADE Display model. SCADE Suite Editor enables the combination of data-flow and state machine notations, as illustrated on Figure 3.6.

**Figure 3.3:** Graphics in SCADE Display Editor
3.2 Esterel SCADE Modeling Techniques

3.2.1 Modeling the graphics with SCADE Display

A SCADE Display symbology model is made of several layers that may be superposed to display complex images.

Each SCADE Display layer is made of:

- primitive graphical objects such as lines, arcs, circles, rectangles, bitmaps, etc. Objects have attributes such as color, line style, etc.
- groups of such objects that may be used to structure the graphical model in a tree and also to apply geometrical transformations such as translation, rotation, etc. Groups may contain subgroups, thus creating the tree-structured graphical model.

Some of the object attributes may be connected to variables. These variables are said to be “plugged”. For example, the angle of the “Power” object in Figure 3.4 is directly connected to the “Power” variable. Plugged variables have to be computed by the behavior model of the display specification (see Section 3.2.2).

![Figure 3.4: Plugged objects in a symbology layer](image)

3.2.2 Modeling the behavior with SCADE Suite

3.2.2.1 Familiarity and accuracy reconciled

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

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

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

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

Table 3.1: Components of Scade functional modules: operators

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

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

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

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

An operator is fully modular:
- There is a clear distinction between its interface and its body.
- There can be no side-effects from one operator to another one.
- The behavior of an operator does not depend on its context.
- An operator can be used safely in several places in the same model or in another one.
3.2.2.3 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. Data is continuously subject to the same transformation.

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

![Figure 3.6: Sample of SCADE Suite model data flows from a Flight Control system](image)

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

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

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

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

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

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

Figure 3.8: Functional expression of concurrency in SCADE Suite

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

- **Operands** SetRegulationMode and ComputeTargetSpeed are functionally parallel; since they are independent, the relative computation order of these operators does not matter (because, in Scade, there are no side-effects).
- ComputeThrottle functionally depends on an output of ComputeTargetSpeed. SCADE Suite KCG Code Generator takes this into account: it generates code that executes ComputeTargetSpeed before ComputeThrottle. The computation order is always up-to-date and correct, even when dependencies are very indirect and when the model is updated. The users do not need to spend time performing tedious and error-prone dependency analyses to determine sequencing manually. They can focus on functions rather than on coding.

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

Figure 3.9: Detection of a flow initialization problem

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

Figure 3.10: Initialization of flows
3.2.2.4 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.

Figure 3.11: SCADE State Machine

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. Scade hierarchical state machines are called SCADE State Machines (SSMs).

SSMs are hierarchical. States can be either simple states or macro states, themselves recursively containing a full SSM. When a macro state is active, so are the SSMs 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 SSMs 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 SSMs specifically forbids dubious constructs found in other hierarchical state machine formalisms: transitions crossing macro state boundaries, transitions that can be taken halfway and then backtracked, and so on. These are non modular, semantically ill-defined, and very hard to figure out, hence inappropriate for safety-critical designs. They are usually not recommended by methodology guidelines.

3.2.2.5 Combining data and control flows

Large applications contain cooperating continuous and discrete control parts. SCAD E Suite gives developers the ability to rigorously combine and nest data flows and control flows, as shown in Figure 3.12.
3.2.2.6 Scade data typing

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

- **Predefined types**: Boolean, Integer, Real, Enumeration, Character.
- **Structured types**:
  - Structures make it possible to group data of different types. Example:
    
    ```
    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 the SCADE Suite Semantic Checker.
3.2.3 Esterel SCADE cycle-based intuitive computation model

The cycle-based execution model of Esterel SCADE 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 3.13 below.

In this loop, there is a strict alternation between environment actions and program actions. Once the input sensors are read, the cyclic function generated from the SCADE Suite model starts computing the cycle outputs and it draws the graphics generated from the SCADE Display model while setting the relevant attributes with the values of the plugged variables that are some of the outputs of SCADE Suite. During that time, the cyclic function is blind to environment changes. 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 3.8). 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 Scade.

SCADE State Machines (SSM) have the very same notion of a cycle. For a simple state machine, a cycle consists in performing the adequate transition from the current state to this cycle’s active state and compute actions in the active state. Concurrent state machines communicate with each other, receiving the signals sent by other machines and possibly

3. It is still possible for interrupt service routines or other task to run, as long as they do not interfere with the cyclic function.
Esterel SCADE for Displays with DO-178B Objectives

sending signals back. Finally, data flow diagrams and SSMs in the same design also communicate at each cycle.

SCADE Display also has this notion of cycle in the sense that, once the values of the plugged variables are computed, then the graphical system is displaying all the layers, following their priority order. Once this is all done, a new computation cycle can start.

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, SSMs, or display of graphical layers. Concurrency is resolved at code generation time, and the generated code remains standard, sequential, and deterministic C code, all contained within a very simple subset of this language. What matters is that the final sequential code behaves exactly as the original concurrent specification, which can be formally guaranteed. Notice that there is no overhead for communication, which is internally implemented using well-controlled shared variables without any context switching.

3.2.4 Esterel SCADE as a model-based development environment

Esterel SCADE is an environment for the development of safety-critical avionics display software. It supports a model-based development paradigm illustrated in Figure 3.14.

• The SCADE Suite and SCADE Display models represent the software requirements. They are mainly used to represent high-level requirements, low-level requirements, and architecture. Both models rely on a formally defined notation.
• The models can be managed under configuration control.
• Documentation is automatically and directly generated from the models; it is correct and up-to-date by construction.
• Semantics checking can be performed in order to check that the SCADE Suite model follows the rules of the Scade notation semantics.
• Design checking can be performed in order to verify that the SCADE Display model follows a set of built-in and user-specified design rules.
• The SCADE Display model can be exercised through animation.
• Both models can then be exercised by simulation using the same integrated code as the embedded code.
• Model coverage analysis can be performed to assess how thoroughly the SCADE Suite model was tested and to detect unintended functions in the model.
• Formal verification techniques can be directly applied to the SCADE Suite model to detect corner cases defects or to prove safety properties.
• Time and stack analysis can be performed in order to verify that SCADE Suite model execution on the target can meet the execution time and stack space budgets.
• Code is automatically and directly generated from the models, in the case it represents low-level requirements, with the KCG qualified Code Generators: the source code is therefore correct and up-to-date by construction.
Object code verification is based on a sample of source C code constructs that can be generated from SCADE Suite and SCADE Display models and that has to be tested on the target.

SCADE Suite and SCADE Display-generated code can be wrapped in an RTOS task, thus implementing the needed cyclic function.

The DO-178B Certification Kits provide all of the evidence that is needed to qualify both SCADE Display KCG and SCADE Suite KCG as development tools at Level A.

**Figure 3.14:** Model-based development with SCADE Display and SCADE Suite

SCADE Display and SCADE Suite apply the following “golden rules”:

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

**BENEFITS OF THE DESIGN-VERIFY-GENERATE PRINCIPLE**

SCADE Display and SCADE Suite allow saving the time spent on significant verification efforts because models can be verified as soon as they are available (even in parts) thus avoiding situations where code has to be developed before any verification can start and every error that is detected requires a lengthy change cycle.


**BENEFITS OF THE "DO THINGS ONCE" PRINCIPLE**

SCADE models formalize a significant part of the software architecture and design. The model is written and maintained once in the project and shared among team members. Expensive and error-prone rewriting is thus avoided; interpretation errors are minimized. All members of the project team, from the specification team to the review and testing teams, can share SCADE models as a reference.

This formal definition can even be used as a contractual requirement document with subcontractors, for example between the aircraft manufacturer and the cockpit display system supplier. 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 Display and SCADE Suite in a DO-178B project.

3.2.5 **SCADE Display and SCADE Suite modeling and safety benefits**

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

- The SCADE modeling notations (of both SCADE Display and SCADE Suite) were rigorously defined. Their interpretation does not depend on readers or any tool. They rely on more than 20 years of academic research ([Esterel], [Lustre]). For example, the semantics kernel of the SCADE Suite modeling notation has not changed over the last 25 years.
- SCADE models are simple. They rely on very few basic concepts and simple combination rules of these concepts.
- Control structures remain at a high-level of abstraction. For example, array operations in Scade are expressed as such and do not require low-level loops and indexes. There is no need for goto’s, no need for the creation of memory at runtime, no way to incorrectly access memory through pointers or an index out of bounds in an array. Moreover, these principles are reflected in the generated code out of SCADE Suite KCG.
- The SCADE modeling notations contain specific features oriented towards safety: strong typing, mandatory initialization of flows, and so on.
- SCADE 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 modeling notations provide a simple and clean expression of concurrency at functional level (data flows express dependencies between operators). Within a Scade model, this avoids the traditional problems of deadlocks and race conditions.
- The Esterel SCADE Editors and Code Generators perform the complete verification of language rules, such as type and clock consistency, initialization of data flows, or causality in Scade models.
4. Software Development Activities with Esterel SCADE

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

4.1 Overview of Software Development Activities

The development process uses a combination of SCADE Display and SCADE Suite development flows and more traditional textual/manual flows. It was observed on industrial projects that the fraction developed with SCADE Display and SCADE Suite typically ranges from 60 percent to 90 percent of the applicative part of the software. Figure 4.1 shows the DO-178B development processes and highlights with shadows those where SCADE Display and SCADE Suite are used.

Figure 4.1: Software development processes with SCADE Display and SCADE Suite
Some companies start using Esterel SCADE to prototype displays during the system definition phase.

In the software requirements process, partial Scade modeling is a good support for the identification of high-level functions, their interfaces, and their data flows.

SCADE Display and SCADE Suite modeling is used extensively in the software design process to develop major parts of the low-level requirements and the architecture.

From such models, the qualified SCADE Display KCG 6.2 and SCADE Suite KCG 6.1.3 Code Generators can automatically generate C source code.

As shown in Figure 4.2, the HLRs that are described in SCADE Display or SCADE Suite models are also LLRs. Many other HLRs, which are textual, are refined in Scaed form in the design phase.

DO-178B requires traceability to be provided both ways between:

- System requirements and HLRs
- HLRs and LLRs
- LLRs and source code

Traceability between system requirements and software high- and low-level requirements described in SCADE can be managed by the SCADE LifeCycle Requirements Management Gateway, as indicated in Figure 4.2 above.
4.2 Software Requirements Process with Esterel SCADE

In DO-178B terminology, the inputs to the Software Requirements Process are the System Requirements. Regarding displays, this will typically be a high-level textual description of the elements to be displayed, together with the ergonomics standards to be applied (see for example [ARP 5056] and [AC 25-11A] in Section 2.1.3). The Software Requirements Process then produces the high-level requirements (HLR).

The requirements of the application will be sorted to take into account both the graphics requirements (to be allocated to SCADE Display) and the SW logics requirements (to be allocated to SCADE Suite).

4.2.1 Graphics high-level requirements

At this stage, it is expected that the graphics HLR are expressed as a SCADE Display model with a structure in layers as shown in Figure 4.3.

For each layer, a static graphical model is expressed as a SCADE Display model with a structure in layers as shown in Figure 4.3. The software design process with SCADE Display is structured along three parts:

- The first one is the layer: the top-level functional structure of the design. Each layer is independent and leads to a pair of C functions in the generated code (init and draw). At integration time, one can use these functions to obtain different behaviors (e.g., refresh rates, screen dispatching). Figure 4.4 shows the main design functionalities as separated (ADI - Attitude Direction Indicator - and Scale Tapes - indicating aircraft speed and altitude).
The second one is the tree structure. Each element of the tree is an object that can be composed of objects. The functional behavior of objects (driven by plugs) has to be defined at the highest logical level of the tree. The order of the object in the tree sets the drawing priorities. As shown in Figure 4.5, the object at the top of the tree is the first drawn. As a result the object at the bottom is the last one that is drawn (and shall graphically cover the previous objects of the tree).

The third one is the reference object. A reference object is an instance of a reusable part of a third party model (e.g., library of graphical components). It is a good idea to use reference objects to factorize common symbols and share them along several display specifications.

The common aspects shared with SCADE Suite methodology are the following ones:

- Use of explicit names for design elements.
- Use of the SCADE LifeCycle Requirements Management Gateway for traceability.

### 4.2.2 Logics high-level requirements

A partial Scade model (see example in Figure 4.6) can be developed at this stage to:

- Identify the high-level functions. One would typically develop a functional breakdown down to a depth of two or three.
- Formalize the interfaces of these functions: names, data types.
- Describe the data flows between these functions.
- Verify consistency of the data flows between these functions using the SCADE Suite Semantic Checker.
- Prepare the framework for the design process: define the top-level functions while ensuring consistency of their interfaces.
SCADE Suite’s flexible annotation feature allows attaching comments or more specific information to the Scade operators, interfaces, and data types.

The document items generated from the SCADE model can be inserted or handled as an annex to the HLR document. This facilitates reviews and analyses that prove the objectives have been met.

4.3 Software Design Process with Esterel SCADE

In DO-178B terminology, the software design process produces the architecture and the low-level requirements. Figure 4.7 illustrates the design flow with SCADE Display and SCADE Suite that is detailed in the next sections.
4.3.1 Architecture design

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

The application is decomposed functionally into its main design units. The characteristics of these units serve as a basis for allocating their refinement in terms of techniques (SCADE Display, SCADE Suite, C, assembler, ...) and team. Among those characteristics, one has to consider:

- The type of processing (e.g., graphics, 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 Display is well-adapted for all the graphical display part of the software. SCADE Suite is well-adapted to the functional parts of the software, such as implementation of the geometrical transformations described in Section 4.2.1, logic, filtering, regulation. It is well-suited to completely specify the dynamic behavior of the Display application. It may be less appropriate for low-level software such as hardware drivers, interrupt handlers, and encoding/decoding routines.

SCADE ARCHITECTURE DESIGN
The objective of the SCADE architecture design activity is to lay the foundations for the development of the SCADE LLRs.

A good SCADE Display architecture is composed of several superposed layers corresponding to several pages; on each layer, the top-level group of objects display information in identified zones of the screen, successive groups of objects enable further
structuring as well as geometric transformations or show/hide parts of information; eventually, leaf groups gather graphics primitive drawings.

A good SCADE Suite architecture is composed of data type definitions, top-level operators, and their connections.

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 analyzability**: Readability comes naturally through the clear and unambiguous Scade language semantics and simple and intuitive graphical symbology. Analyzability comes naturally with a formal notation such as Scade.
- **Efficiency**: There is no magic recipe to achieving a good Scade model architecture, rather it requires a mix of experience, creativity, and rigor. Here are a few suggestions:
  - Be reasonable and realistic: nobody can build a good architecture in one shot. Do not develop the full model from your first draft, but build two or three architecture variants, then analyze and compare them; otherwise, you may have to live with a bad architecture for a long time.
  - Review and discuss the architecture with peers.
  - Simulate the impact of some changes that are likely to occur (e.g., adding a sensor or an error case in SCADE Suite, or re-arrange the contents of layers and groups in SCADE Display) and evaluate the robustness of the architecture to such changes.
  - Select the architecture that minimizes complexity of connections and is robust to changes.

For example, the architecture shown in Figure 4.6, the architecture groups several logical controls in one structured top-level operator in SCADE Suite. Such design is more maintainable than if each individual control would have its own function with duplicated interfaces in the model.

**Note**: If SCADE was already used for HLRs, it has to be considered as the first candidate for SCADE architecture, since it has the best direct traceability to HLRs. Nonetheless, it is recommended to verify also this architecture and ensure it has the right properties for maintainability.

### 4.3.2 SCADE low-level requirements development

Once the SCADE architecture is defined, the modules are refined to formalize the low-level requirements (LLR). The objective of this activity is to produce a complete and consistent software model.

The following sections provide some examples of modeling patterns with SCADE Suite.

**INPUT/OUTPUT HANDLING WITH SCADE SUITE**

We assume that raw acquisition from physical devices and/or from data buses is done with drivers to feed SCADE Suite inputs.

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 4.8

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 4.9 below.

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

**FILTERING AND REGULATION WITH SCADE SUITE**

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 in exactly the same way as control engineers do, using the same semantics.

The Scade time operators fit the z operator of control engineering. For instance, the z⁻¹ operator of control engineering (meaning a unit delay) has equivalent operators called “pre” and “fby” in Scade. For example, if a control engineer has written an equation such as \[ s = K_1u - K_2z^{-1}s \], which means \[ s(k) = K_1u(k) - K_2s(k-1) \], this can be expressed in textual Scade as \[ s = K_1u - K_2*pre(s) \] or graphically, as shown in Figure 4.10 below.

Scade can implement both Infinite Impulse Response (IIR) filters 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.

There are two possibilities for building filtering or regulation algorithms with SCADE Suite:

1. Develop these algorithms directly in graphical or textual Scade.
Develop them by reusing library operators such as “first order filter”, “integrator”, etc. These library operators are themselves developed with SCADE Suite.

Using library operators has many advantages:
- It saves time.
- It relies on validated components.
- It makes the model more readable and more 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.

**DECISION LOGIC WITH SCADE SUITE**

In modern controllers, logic is often more complex than filtering and regulation. The controller has to handle:
- Identification of the situation
- Detection of abnormal conditions
- Decision making
- Management of redundant computation chains

SCADE Suite offers a variety of techniques for handling logic:
- Logical operators (such as and/ or/ xor) and comparators.
- Selecting flows, based on conditions, with the “if” and “case” constructs.
- Building complex functions from simpler ones. 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 4.11**.

- Conditional activation of operators depending on Boolean conditions.
- SCADE State Machines (SSM), as in **Figure 4.12**.
WHICH TECHNIQUE TO USE 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:

To select between state machines and logical expressions:

- Does the output depend on the past? If it only depends on the current inputs, then this is just combinational logic: simply use a logical expression in the data flow. A state machine that just jumps to state $X_i$ when condition $C_i$ is true, independently of the current state, is a degenerated one and does not deserve to be a state machine.

- Does the state have a strong qualitative influence on the behavior? This is in favor of a state machine.

To express concurrency:

- Simply design parallel data flows and SCADE State Machines (SSM): this is natural and readable, and the code generator is in charge of implementing this parallel specification into sequential code.

ROBUSTNESS

Robustness issues must be addressed at each level. Esterel Technologies recommends that robustness be addressed differently at the design and coding levels.

- **Design Level with SCADE Suite**
  At the design level, the specification should explicitly identify and manage the safety and the robustness of the software with respect to invalid input data (see Input/Output Handling with SCADE Suite). There should be no exception mechanism to respond to incorrect sensor or pilot data, but planned mastered reaction. This involves techniques such as voting, confirmation, and range checking. At this level, one should explicitly manage the ranges of variables. For instance, it is highly recommended that an integrator contain a limiter. Or, if there is a division, the case when the divider is zero has to be managed explicitly. In the context of the division, the division should only be called when the divider is not zero (or, more precisely, far enough from zero). The action to be taken when the divider is near zero has to be defined by the writer of the software requirements, not by the programmer.

  It is easy to define libraries of robust operators, such as guarded division, voters, confirmators, and limiters. Their presence in the diagrams is very explicit for the reader. Esterel Technologies recommends that one uses the same numeric data types on the host and on the target with libraries that have the same behavior.

- **Coding Level**
  On the contrary, if an attempt to divide by zero happens at runtime in spite of the above-mentioned design principles, this is an abnormal situation caused by a defect in the software design. Such a failure has to be handled as a real exception. The detection of the event can be typically part of the arithmetic library (the optimal implementation of that library is generally target-dependant). The action to be taken (e.g., raise an exception and call a specific exception handler) has to be defined in the global architecture design of the computer.
4.4 Software Coding Process

SCADE Display and SCADE Suite KCG Code Generators automatically generate the complete code that implements the software system design and module design defined in SCADE for graphical display and logic (see Figure 4.13). In this way, the complete software is automatically implemented.

Let us now distinguish both cases when using automatic code generation: display with SCADE Display KCG and logic with SCADE Suite KCG.

4.4.1 Code generation from SCADE Display model

The SCADE Display model completely defines the expected graphical rendering of the generated code including calls to the embedded graphical library.

PROPERTIES OF THE GENERATED CODE

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

- The code is portable: it is ISO-C compliant. It performs calls to the embedded graphical library that is implemented on the target platform as well as calls to mathematical and string libraries.
- The code does not perform any operating system call.
- The code structure reflects the SCADE Display model architecture.
- The code is readable and traceable to the SCADE Display model through the use of names and comments.
- Memory allocation is fully static (no dynamic memory allocation).
- There is no recursion and no unbounded loops.
- Execution time is bounded.
- No dynamic address calculation is performed (no pointer arithmetic)
Traceability of the code to a SCADE Display model is illustrated in Figure 4.15 below.

4.4.2 Code generation from SCADE Suite models

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

PROPERTIES OF THE GENERATED CODE

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

• The code is portable: it is [ISO-C] compliant.
• The code structure reflects the model architecture for data-flow parts. For control-flow parts, traceability between state names and C code is ensured.

• The code is readable and traceable to the input Scade model through the use of corresponding names, specific comments, and traceability file.
• Memory allocation is fully static (no dynamic memory allocation).
• There is no recursive call.
• Only bounded loops are allowed, since they use static values known at SCADE Suite KCG code generation time.
• Execution time is bounded.
• The code is decomposed into elementary assignments to local variables (this restricts the use of the optimization options of the C compiler).
• Expressions are explicitly parenthesized.
• No dynamic address calculation is performed (no pointer arithmetic).
• There are no implicit conversions.
• There is no expression with side-effects (no i++, no a += b, no side-effect in function calls).
• No functions are passed as arguments.
Traceability of the code to a Scade data flow model is illustrated in Figure 4.15.

Figure 4.15: Scade data flow to generated C source code traceability

Traceability of the code to the SCADE State Machine model is illustrated in Figure 4.16.

Figure 4.16: SCADE State Machine to generated C source code traceability
To further support reviews and automated analysis, a traceability file is generated by SCADE Suite KCG as shown in Figure 4.17.

```
<NoExpNode scadeName="AltContrib" headerFile="AltContrib_FlightControl.h" targetCycleFr="AltContrib_FlightControl">
  <Input scadeName="AltTarget" scadeType="real" targetName="AltTarget" targetType="kcg.real"/>
  <Input scadeName="AltSensor" scadeType="real" targetName="AltSensor" targetType="kcg.real"/>
  <Output scadeName="AltContrib" scadeType="real" targetName="AltContrib" targetType="kcg.real"/>
  <Local scadeName="L14" scadeType="real" targetName="AltContrib" targetType="kcg.real"/>
  <Local scadeName="L16" scadeType="real" targetName="AltTarget" targetType="kcg.real"/>
  <Local scadeName="L17" scadeType="real" targetName="AltSensor" targetType="kcg.real"/>
  <NodeInclusion scadeName="linear_gen" instName="1"/>
</NoExpNode>
```

Figure 4.17: Sample of kcg_trace.xml file for traceability between Scade model and source code

**TUNING CODE TO TARGET AND PROJECT CONSTRAINTS**

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

- **Call mode**: the operator is generated as a C function.

- **Inline mode**: the whole code for the operator is expanded where it is called.

This is illustrated in Figure 4.18.

```
Call mode
A{
  ... 
  B();
  ...
  C();
  ...
}

// code of A, B, C */
```

```
Inline mode
A{
  ...
  /* code of A, B, C */
  ...
}
```

Figure 4.18: Comparing Call and Inline modes

Both of these code generation modes (Call or Inline) can be composed at will, performing a call for some operators and inlining for other operators.
4.5 Software Integration Process

4.5.1 Integration aspects

Integration of SCADE-generated code is about:
- Input/output
- Integration of SCADE Suite and SCADE Display-generated code
- Integration of external data types and constants
- Scheduling and tasking
- Integration of SCADE Display-generated code with the embedded graphical library

4.5.2 Input/output

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

4.5.3 Integration of SCADE Suite and SCADE Display generated code

As for interfaces to physical sensors or connections to data buses, the interfaces between SCADE Suite and SCADE Display designs are among the most changing parts in the display application.

Both toolsets are tightly integrated at the co-editing, co-simulation and co-reporting stages of the software development. This level of integration ensures that formalism checks between the logics part and the graphics part of the display specification are robust by nature.

The integrated connection mechanism between SCADE Suite variables and SCADE Display plugs stores mapping data as XML files. This mechanism facilitates the generation of the wrapping code between both logics and graphical parts of the application and allows the necessary level of flexibility to adapt to your display application architecture.

4.5.4 Integration of external data and code

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

4.5.5 SCADE scheduling and tasking

Scheduling has to be addressed in the preliminary design phase, but for the sake of simplicity this section describes it below.

First, the section recalls the execution semantics of Scade (for both SCADE Display and SCADE Suite models), and then examines how to model and implement scheduling of a SCADE Suite and SCADE Display model in single or multirate mode, while in single tasking or multitasking mode.

SCADE EXECUTION SEMANTICS

SCADE execution semantics is based on the cycle-based execution model as described in Section 3.2.3. This model can be represented with Figure 4.19.

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 SCADE Suite code execution is ended, the value of some its output variables may be sent as inputs to SCADE Display, through plugged variables. The main SCADE Display function of the generated code (the “draw” function) is called, and 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:

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

SINGLE-TASK INTEGRATION OF SCADE FUNCTION WITH AN RTOS

A SCADE 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 4.20. The infinite loop construct is replaced by a task. This task is activated by the start event of the SCADE design, which can be a periodic alarm or a user activation.

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

Note that within a SCADE Suite model concurrency is expressed functionally in the model and that the Code Generator takes into account the model structure to generate sequential code, taking into account this functional concurrency and the data flow dependencies. There is no need for the user to spend time sequencing parallel flows, neither during modeling nor during implementation. There is no need to develop multiple tasks with complex and error-prone synchronization mechanisms.

Note that other code, such as hardware drivers, may run in separate tasks, provided they do not interfere with the SCADE-generated code.

MULTIRATE, SINGLE-TASK APPLICATIONS

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

The following application has two rates: Sys1 (as fast as the top-level) and Sys2 (four times slower), as shown in Figure 4.21.
The schedule of this application is as shown in Figure 4.22 below:

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 4.23:

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 4.24 below:

4. 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 Scade model. The operator execution only occurs when a given activation condition is true.
Note that Sys1 execution time can be longer than with the previous design. This means that a slower, but less expensive, processor can be used.

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

Such design has advantages, but it also has constraints:

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

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

**MULTITASKING IMPLEMENTATION**

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

It is possible to build a global Scade model, which formalizes the global behavior of the application, while implementing the code on different tasks. While it is also possible to build and implement separate independent models, the 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).

4.5.6 Integration of the SCADE Display generated code with the embedded graphics library

SCADE Display KCG Code Generator generates C source code that contains calls to high-level SGL graphics functions (e.g., `sglDrawFastBitmap` to draw a block of pixels).

In most cases, the SGL graphics functions are implemented by the SCADE Display O GLX embedded graphics library that Esterel Technologies provides together with SCADE Display KCG. The O GLX library, which in this case has to be certified for DO-178B at Level A, provides an implementation of the SGL calls by using function calls obeying the OpenGL SC (Safety Critical) or the OpenGL ES (Embedded Systems) graphics standards.
This means that an OpenGL library must be available on the target platform and this library must be certified for DO-178B at Level A. This typical architecture is described below.

1. The graphical functions contained in the code generated by SCADE Display KCG correspond to SGL commands.
2. SCADE Display OGLX generates calls to OpenGL functions that are implemented by the OpenGL driver.
3. Finally, the OpenGL driver generates graphics rendering.

### 4.6 Teamwork

Working efficiently on a large project requires both distribution of the work and consistent integration of the pieces developed by each team.

The Scade language is modular: there is a clear distinction between the interfaces and the contents of operators and there are no side effects from one operator to another.

A typical project organization is shown in Figure 4.26:

- A software architect defines the top-level operators, their interfaces, and connections.
- Utility libraries are developed.
- Each major subfunction, corresponding to a top-level operator is developed by a specific team; the interfaces of these top-level operators define a framework for these teams, which maintain consistency of the design.
At each step, the team can verify in a mouse click that the subsystem remains consistent with its interface. Later, the integration of those parts into a larger model can be achieved by linking these “projects” to the larger one. At any stage, the SCADE Suite Semantic Checker verifies the consistency of this integration in a mouse click.

Teamwork also addresses separation between the logics and the graphics (teams working on logics versus teams working on graphics). Both can share the same interfaces, while concentrating on their implementation.

All these data have to be kept under strict version and configuration management control. SCADE Display and SCADE Suite can be integrated with the user configuration management system via Microsoft SCCI™ (Source Code Control Interface), supported by most commercial Configuration Management Systems.

Reuse is also an important means of improving productivity and consistency in a project or a series of projects. SCADE Suite libraries can store definitions of operators and/or data types for reuse in several places. These range from basic operators such as latches or integrators to complex, customer-specific systems. SCADE Display objects can be stored in reference object files and reused amongst several projects.
5. Software Verification Activities

5.1 Overview

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.

According to DO-178B, 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.

5.2 Verification of the SCADE High-Level Requirements

5.2.1 Verification objectives for HLR

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

Table 5.1: DO-178B Table A-3

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

For those elements of the SCADE Display and/or SCADE Suite models that are developed during the requirements phase, they must be verified against the objectives of DO-178B Table A-3.
In case they are also used to generate code, they also must be verified against the objectives that DO-178B defines for low-level requirements (see Section 5.3), since “when code is generated from HLR, these are also considered LLR, and the guidelines for LLR also apply to them” (DO-178B; Section 5.0).

5.2.2 Verification methods for HLR

COMPLIANCE WITH SYSTEM REQUIREMENTS
At this stage, and as described in 4.2.1, the graphics HLR have been partially described as SCADE Display/SCADE Suite models. In particular, the graphics have been described in the form of SCADE Display models. In order to verify that these graphics models comply with the corresponding systems requirements allocated to software, a review activity is organized. This is completed by exercising the graphics models with the animation feature of SCADE Display.

It may also be the case that some top-level SCADE Suite models are designed as HLR for the logic part. As they are incomplete models at this stage, verification is typically performed by review.

ACCURACY AND CONSISTENCY
Again, regarding the SCADE Display models which are complete, SCADE Display Design Checker is used to verify model consistency.

Regarding the SCADE Suite models, which at this stage are incomplete, some consistency checks of both the interface and the connections between model elements may be automated by SCADE Suite Semantic Checker.

COMPATIBILITY WITH TARGET COMPUTER
There is nothing specific to SCADE Display or SCADE Suite at this stage.

VERIFIABILITY
The SCADE Display and/or SCADE Suite models identify the graphical structure and objects, the top-level functions and describe the functional breakdown and data flows between top-level functions. This description is verifiable.

COMPLIANCE WITH STANDARDS
The SCADE notations have precise syntactic and semantic rules (e.g., graphical behavior, data type consistency) defined in the SCADE Display User Manual [DISPLAY_UM] and the Scade 6 Language Reference Manual [SCADE-LRM]. Compliance with the SCADE Display standards is automatically enforced because there is no way to enter an incorrect display model within SCADE Display Editor and SCADE Display Design Checker can be used to enforce specific design rules. Regarding SCADE Suite, the syntactic correctness of a model is guaranteed as well by SCADE Suite Editor, and its semantic correctness can be checked by SCADE Suite Semantic checker.
TRACEABILITY TO SYSTEM REQUIREMENTS
Traceability can be managed with the SCADe LifeCycle Requirements Management Gateway to reference software safety requirements (see Section 6.3).

ALGORITHMS ACCURACY
There are usually no Scade algorithms at this stage.

5.2.3 Verification summary for HLR

Table 5.2 summarizes verification objectives and methods for the software high-level requirements described in SCADe.

<table>
<thead>
<tr>
<th>Objective</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td>Software high-level requirements comply with system requirements. Review of SCADe Display and SCADe Suite models, plus animation of SCADe Display models</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td>High-level requirements are accurate and consistent. Review of SCADe models. Use of SCADe Display Design Checker and SCADe Suite Semantic Checker.</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td>High-level requirements are compatible with target computer. Review</td>
</tr>
<tr>
<td><strong>4</strong></td>
<td>High-level requirements are verifiable. Review</td>
</tr>
<tr>
<td><strong>5</strong></td>
<td>High-level requirements conform to standards. SCADe Display Editor and Design Checker, SCADe Suite Editor and Semantic Checker.</td>
</tr>
<tr>
<td><strong>6</strong></td>
<td>High-level requirements are traceable to system requirements. SCADe LifeCycle Requirements Management Gateway results analysis</td>
</tr>
<tr>
<td><strong>7</strong></td>
<td>Algorithms are accurate. N/A (No Scade algorithm at this stage)</td>
</tr>
</tbody>
</table>
5.3 Verification of the SCADE
Low-Level Requirements and
Architecture

5.3.1 Verification objectives for the
LLR and architecture

The complete SCADE Display and SCADE
Suite models have to be verified against the
objectives that DO-178B defines for low-level
requirements (see Table 5.3). This is the case
even if all or part of these SCADE models are
developed as HLR. Indeed, “when code is generated
from HLR, these are also considered LLR, and the
guidelines for LLR also apply to them” (DO-178B;
Section 5.0).

For LLR that are not developed in SCADE,
verification activities have to be performed in
the traditional way.

Table 5.3: DO-178B Table A-4

<table>
<thead>
<tr>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Low-level requirements comply with high-level requirements.</td>
</tr>
<tr>
<td>2 Low-level requirements are accurate and consistent.</td>
</tr>
<tr>
<td>3 Low-level requirements are compatible with target computer.</td>
</tr>
<tr>
<td>4 Low-level requirements are verifiable.</td>
</tr>
<tr>
<td>5 Low-level requirements conform to standards.</td>
</tr>
<tr>
<td>6 Low-level requirements are traceable to high-level requirements.</td>
</tr>
<tr>
<td>7 Algorithms are accurate.</td>
</tr>
<tr>
<td>8 Software architecture is compatible with high-level requirements.</td>
</tr>
<tr>
<td>9 Software architecture is consistent.</td>
</tr>
<tr>
<td>10 Software architecture is compatible with target computer.</td>
</tr>
<tr>
<td>11 Software architecture is verifiable.</td>
</tr>
<tr>
<td>12 Software architecture conforms to standards.</td>
</tr>
<tr>
<td>13 Software partitioning integrity is confirmed.</td>
</tr>
</tbody>
</table>

5.3.2 Scade model accuracy and
consistency

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

- Detection of missing definitions
- Warnings on unused definitions
- Detection of non initialized variables
- Coherence of data types and interfaces
• Coherence of “clocks,” namely of production/consumption rates of data
It is also possible to add custom verification rules using the programmable interface (API) of SCADE Suite Editor.

5.3.3 Compliance with design standard

The SCADE Display User Manual [DISPLAY_UM] defines the design standard for the SCADE Display architecture and LLRs: it defines precisely the rules applying to the symbology layers, the graphical objects, and the ways to group them. SCADE Display Editor guarantees that a SCADE Display model complies with those rules. Furthermore, SCADE Display Design Checker can be used to enforce specific design rules.

The Scade 6 Language Reference Manual [SCADE-LRM] defines the design standard for the SCADE Suite architecture and LLRs; it defines precisely the syntactic and semantic rules that a Scade model has to follow. The SCADE Suite Syntactic and Semantic Checkers (included in KCG) verify compliance with this standard. Furthermore, the SCADE Suite API and the TLC scripting language can be used to enforce specific Scade design rules.

5.3.4 Traceability from SCADE LLR to HLR

Traceability from SCADE LLRs to the HLRs can efficiently be supported by the SCADE LifeCycle Requirements Management Gateway (RM Gateway).

Using this tool, HLRs can be captured in documents of various formats such as Adobe® PDF, Microsoft® Word and Excel, Visio®, Borland® CaliberRM™, IBM® Rational® RequisitePro® and DOORS®, etc. Traceability between SCADE LLRs and HLRs can then be entered within the RM Gateway, as shown in Figure 5.1.
With a mouse click, it is possible to navigate between HLRs and LLRs to understand the impact of a change in the LLRs implementation, etc.

Traceability analysis finally can be performed to check for situations such as uncovered requirements, undefined requirements, etc.

Traceability between HLRs and test cases can later be exercised with the RM Gateway as well.

5.3.5 Verifiability

The SCADE Display and SCADE Suite models describe the low-level requirements and the architecture of the corresponding software part.
Since the SCADE Display and SCADE Suite notations have a formal definition, these LLRs are formally verifiable.

### 5.3.6 Compliance with high-level requirements

Verifying compliance of the SCADE Display and SCADE Suite models with the HLR uses several complementary techniques:
- Peer review
- Animation and Simulation
- Formal verification

**REVIEW OF THE SCADE MODEL**

Peer review is an essential technique for verifying compliance of LLRs with HLRs.

For review, a report containing all SCADE Display and SCADE Suite models data can be automatically generated by SCADE LifeCycle Reporter. The SCADE Display and SCADE Suite design notations have several advantages compared to a textual notation:
- The description is not subject to interpretation. This is because the SCADE Display and SCADE Suite notations have a formal definition.
- The description is complete. Incompleteness is not possible in SCADE Display editing and it is detected by SCADE Suite Semantic Checker.
- Its graphical form is simple and intuitive.

Peer review can verify adherence to the design rules ensuring robustness as explained in Section 4.3.2.

**SCADE DISPLAY ANIMATION**

The animation capability of SCADE Display allows animating the display with some predefined scenarios.

![Animation of a SCADE Display Specification](image)
Animation laws are applied to the plugged variables of the display. Variation of these variables may be continuous (e.g., a translation) or discrete (e.g., a color). In the case of continuous variables, various laws can be applied, such as “ramp”, “triangular”, and “linear”, as shown on Figure 5.2 above.

Animation is performed until the specification reaches, at this stage, the required level of quality. In the next verification phase, with SCADE Suite and SCADE Display co-simulation, it is possible to exercise the display on the basis of much more realistic scenarios.

**SCADE DISPLAY/SUITE CO-SIMULATION**

It is now helpful to dynamically exercise the behavior of a SCADE Display/SCADE Suite model to better verify how it functions. As soon as a SCADE Suite model (or pieces of it) is available, it can be simulated with SCADE Suite Simulator, as shown in Figure 5.3. Simulation can be run interactively or in batch. Scenarios (input/output sequences) can be recorded, saved, and replayed later on the Simulator or on the target. Esterel Technologies recommends that SCADE Suite users run simulation in batch mode and use several optimization levels while checking that the simulation results obtained are all identical. Note that all simulation scenarios, like all testing artifacts, have to be based on the software high-level requirements and must be verified.

Moreover, the SCADE Suite and SCADE Display models can be co-simulated as shown on Figure 5.3 to provide a fully realistic view of both graphics and logics.
Simulation supports the detection of assertion violation, which is extremely helpful during the verification of robustness.

**SCADE SUITE FORMAL VERIFICATION WITH DESIGN VERIFIER**

SCADE Suite Design Verifier\(^5\) provides, for the logic associated with the display, an original and powerful verification technique based on formal verification technologies.

---

5. Powered by Prover® Plug-In. Prover, Prover Technology, Prover Plug-In and the Prover logo are trademarks or registered trademarks of Prover Technology AB in Sweden, the United States and in other countries.
Testing activities, including SCADE Suite simulation, let the user test and verify the correctness of the design. However, with testing, one is never 100 percent sure that the design is correct because one usually never tests all possible scenarios.

Formal verification of computer systems consists of a set of activities using a mathematical framework to reason about system behaviors and properties in a rigorous way. The recipe for formal verification of safety properties is:

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

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

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

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

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

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

![Figure 5.4: Observer operator containing landing gear safety property](image)

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

![Figure 5.5: Connecting the observer operator to the landing gear controller](image)

Traditionally, expressing a property and finding a proof for a real system containing complex algorithms and control logic required a large amount of time and expertise in mathematics. Thus the use of formal verification techniques was marginal. Hence, the major challenge of formal verification is to provide system engineers and software developers with an efficient easy-to-use and friendly framework that does not require a lot of time to use and also enables increased confidence in the system. To meet this challenge, SCADE Suite Design
Verifier offers users a solution for easy access to formal verification that relies on the following characteristics:

- **Property Expression**: The Scade language itself expresses properties. There is no need to learn a mathematical dialect to express the property requirements that a user wants the design to fulfill.

- **Property Verification**: This is a push-button feature of the SCADE Suite toolset, which provides a yes/no answer. Moreover, in the case of a no answer, SCADE Suite lets the user discover, in an automatic and user-friendly way, why a no answer was reached: generation of a counter example scenario that can be played back in SCADE Suite Simulator.

Design Verifier helps detect specification errors at the early phase of the software flow, minimizing the risk of discovering these errors during the final integration and validation phases. The input to Design Verifier is a set of properties that must be checked for correctness in the design. This set of safety properties is extracted and specified from the high-level requirements and/or from the safety analysis.

**Figure 5.6**: Design Verifier workflow. It consists of successive tasks that may be iterated. There are three kinds of tasks:

- **Property Definition**: This task consists in extracting properties from the high-level requirements to be checked with Design Verifier.

- **Property and Environment Specification**: This task consists in formally describing, as Scade observer properties, the requirement extracted as properties in SCADE Suite. Necessary information from the environment of the design must be specified formally in Scade as well. For example, if the altitude is always more than 100 feet above sea level, this assertion has to be attached to the model, in order to eliminate non-relevant cases.

- **Design Verifier Execution**: This task corresponds to the usage of Design Verifier.

Formal verification can add efficiency to the communication between the Safety Assessment Process and the System Development Process.

5.3.7 **Compatibility with target computer**

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

- Formal worst-case execution time analysis
WORST-CASE EXECUTION TIME ANALYSIS

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

Testing of WCET properties: 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?).

Static formal analysis: SCADE Suite Timing Verifier relies on formal analysis methods for safe prediction of upper bounds for execution time of the SCADE Suite-generated code.

The method of abstract interpretation of models for worst-case execution time allows a safe prediction of WCET as close as possible to the real WCET. Safe prediction means that the real WCET never exceeds the predicted WCET.

SCADE Suite Timing Verifier can be used as a verification tool on the customer project. SCADE Suite Timing Verifier provides an easy-to-use means of predicting the execution time properties of a given application on the desired target. This can be performed very early in the development cycle, as soon as a Scade model of the application is available.

Timing Verifier principles: SCADE Suite Timing Verifier relies on object code files plus additional information directly derived from the formal Scade model.

Figure 5.7: Timing Verifier integration in SCADE Suite
A suitable entry point into the executable file is calculated before calculating a prediction of the WCET. The resulting detailed information is mapped back to the Scade model and can be visualized directly in SCADE Suite IDE.

While object code is fully descriptive on the elementary level (each machine instruction is fully self-descriptive), it contains no information on branches and loops because they are a function of input data or of the high-level structure of the application.

The analysis engine takes as input object code as well as an annotation file that contains specific configuration data (such as loop count upper bounds). As a Scade model and its resulting code are fully deterministic (i.e., the same sequence of input data always produces the same behavior), all information that depends on the structure of the model can be derived and any necessary annotations for the analysis engine (e.g., loop execution counts as statically defined in the Scade model) can be statically determined and given as input to the engine.

Before solving the final execution time, several stages of analysis (loop transformation, cache analysis, pipeline analysis) are carried out as in Figure 5.8. A precise model of the targeted CPU is the basis for the formal abstract interpretation leading to very realistic WCET estimates.

Figure 5.8: Timing and Stack Verifiers analysis stages

Once the execution time is computed, the Timing and Stack Verifiers provide visualization feedback and reporting.

Usage of Timing Verifier: From SCADE Suite IDE, users can generate code directly for each application. For WCET analysis, the user chooses to combine a specific CPU and the matching cross-compiler. The list of supported CPUs is available from Esterel Technologies and is continually expanded.

SCADE Suite initiates the generation of C code with the corresponding Timing Verifier annotation file. The cross-compiler linker chain is called with a custom make file and the Timing Verifier engine starts running the analysis.

The results are directly shown in SCADE Suite IDE. Hyperlinks are available for direct reference to the model constructs matching each WCET result. If needed, more fine-grained visualization is available down to each machine instruction in order to enable the user to carry out deeper analysis of the timing properties of the application.
In complex Scade models with iteration on data arrays, a reorganization of the data structures can sometimes have a significant impact on timing properties. Such issues are immediately visible on the call graph as shown in Figure 5.9.

The following Figure 5.10 finally shows a WCET Report produced by SCADE Suite Timing Verifier. It allows to understand the contribution of each node to the execution time and, on the basis of a Diff Report, it allows to understand what is the effect of a design change (or code generation option change) on code performance.

Figure 5.9: Timing and Stack Verifiers analysis visualization results
STACK USAGE ANALYSIS

Stack usage problem: Stack overflow is 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?

Static stack usage analysis: SCADE Suite Stack Verifier relies on the same method of abstract interpretation to infer safe predictions about stack usage. While object code is fully descriptive on the elementary level (each machine instruction is fully self-descriptive), it contains no information about branches and
loops because they are either a function of input data or a function of the high-level structure of the application.

The analysis engine takes as input: object code and an annotation file containing specific configuration data. Since a Scaide model and its resulting code are fully deterministic (i.e., the same sequence of input data always produces the same behavior), all information that depends on the structure of the model can be derived. Thus, any annotations necessary for the stack analysis engine can be statically determined and given as input to the engine.

SCADE Suite Stack Verifier therefore shares the first steps of the analysis process with Timing Verifier, but instead of using the call tree to determine time consumption, the information is used to calculate the data that will be on the stack. SCAFE Suite Stack Verifier supports a large number of processors: the current list is available from Esterel Technologies.

### 5.3.8 Partitioning

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

### 5.3.9 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-178B Table A-4 Objectives Achievement

<table>
<thead>
<tr>
<th>Objective</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Low-level requirements comply with high-level requirements.</td>
<td>Review of SCAFE LLRs from SCAFE Design Documents, animation, co-simulation, formal verification</td>
</tr>
<tr>
<td>2 Low-level requirements are accurate and consistent.</td>
<td>SCAFE Display Design Checker and SCAFE Suite KCG Semantic Checker result analysis</td>
</tr>
<tr>
<td>3 Low-level requirements are compatible with target computer.</td>
<td>SCAFE computational model uses no target-specific resource. Remains to be verified: memory and CPU consumption. This may be done with the help of SCAFE Suite Timing and Stack Verifiers.</td>
</tr>
<tr>
<td>4 Low-level requirements are verifiable.</td>
<td>SCAFE Suite KCG Semantic Checker results analysis</td>
</tr>
<tr>
<td>5 Low-level requirements conform to standards.</td>
<td>Automated by SCAFE Display Design Checker and SCAFE Suite KCG Syntactic and Semantic Checker Other rules: by review or other specific means</td>
</tr>
<tr>
<td>6 Low-level requirements are traceable to high-level requirements.</td>
<td>SCAFE LifeCycle Requirements Management Gateway traceability analysis</td>
</tr>
</tbody>
</table>
Table 5.4: DO-178B Table A-4 Objectives Achievement (Continued)

<table>
<thead>
<tr>
<th>Objective</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>7  Algorithms are accurate.</td>
<td>Review and simulation</td>
</tr>
<tr>
<td>8  Software architecture is compatible with high-level requirements.</td>
<td>Review</td>
</tr>
<tr>
<td>9  Software architecture is consistent.</td>
<td>SCADE Display Design Checker and SCADE Suite KCG Semantic Checker results analysis</td>
</tr>
<tr>
<td>10 Software architecture is compatible with target computer.</td>
<td>For the SCADE Suite part, estimation by SCADE Suite Timing and Stack Verifiers</td>
</tr>
<tr>
<td>11 Software architecture is verifiable.</td>
<td>SCADE Suite KCG Semantic Checker results analysis</td>
</tr>
<tr>
<td>12 Software architecture conforms to standards.</td>
<td>SCADE Display Design Checker and SCADE Suite KCG Semantic Checker results analysis</td>
</tr>
<tr>
<td>13 Software partitioning integrity is confirmed.</td>
<td>SCADE Display and SCADE Suite introduce no specific risk, but provide no partitioning mechanism; traditional method has to be used</td>
</tr>
</tbody>
</table>
5.4 Verification of Coding Outputs and Integration Process

5.4.1 Verification objectives for coding output and integration process

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

<table>
<thead>
<tr>
<th>Objective</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Source code complies with low-level requirements</td>
</tr>
<tr>
<td>2</td>
<td>Source code complies with software architecture</td>
</tr>
<tr>
<td>3</td>
<td>Source code is verifiable</td>
</tr>
<tr>
<td>4</td>
<td>Source code conforms to standards</td>
</tr>
<tr>
<td>5</td>
<td>Source code is traceable to low-level requirements</td>
</tr>
<tr>
<td>6</td>
<td>Source code is accurate and consistent</td>
</tr>
<tr>
<td>7</td>
<td>Output of software integration process is complete and correct</td>
</tr>
</tbody>
</table>

5.4.2 Impact of code generators qualification

The SCADE Display KCG and SCADE Suite KCG Code Generators can be qualified as development tools because they were developed by Esterel Technologies to fulfill the DO-178B objectives development tools at Level A (see Appendix C for details about qualification).

This has the following consequences:

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

This is ensured by the qualification of the Code Generators.

**SOURCE CODE COMPLIES WITH SOFTWARE ARCHITECTURE**

This is ensured by the qualification of the Code Generators.

**SOURCE CODE IS VERIFIABLE**

By specification of the Code Generators, the generated code reflects the model and is verifiable. The qualification of the Code Generators ensures that this is respected.

**SOURCE CODE CONFORMS TO STANDARDS**

The specification of the code generation defines precise coding standards: it defines precisely how SCADE Display objects and SCADE Suite constructs have to be implemented in C. The qualification of the Code Generators ensures that this standard is respected.

**SOURCE CODE IS TRACEABLE TO LOW-LEVEL REQUIREMENTS**

By specification, the generated code has a simple, readable structure that is traceable to the models by names and by comments. The qualification of the Code Generators ensures that this is respected.
SOURCE CODE IS ACCURATE AND CONSISTENT
The specification of the Code Generators defines accurate and consistent code, reflecting accurate and consistent input models. The qualification of the Code Generators ensures that this is respected.

OUTPUT OF THE SOFTWARE INTEGRATION PROCESS IS COMPLETE AND CORRECT
This verification is achieved by analyzing the linker listing for completeness and consistency (see Combined Testing Process in Section 5.5).

5.4.3 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-178B Table A-5 Objectives Achievement

<table>
<thead>
<tr>
<th>Objective</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Source code complies with low-level requirements. Ensured by SCADE Display KCG and SCADE Suite KCG qualification</td>
</tr>
<tr>
<td>2</td>
<td>Source code complies with software architecture. Review of interfaces between SCADE Suite KCG generated code and imported manual C code</td>
</tr>
<tr>
<td>3</td>
<td>Source code is verifiable. Ensured by SCADE Display KCG and SCADE Suite KCG qualification</td>
</tr>
<tr>
<td>4</td>
<td>Source code conforms to standards. Ensured by SCADE Display KCG and SCADE Suite KCG qualification</td>
</tr>
<tr>
<td>5</td>
<td>Source code is traceable to low-level requirements. Ensured by SCADE Display KCG and SCADE Suite KCG qualification</td>
</tr>
</tbody>
</table>

6. Source code is accurate and consistent. Ensured by SCADE Display KCG and SCADE Suite KCG qualification
7. Output of software integration process is complete and correct. Integration results analysis

5.5 Verification of Outputs of the Integration Process

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

Table 5.7: DO-178B Table A-6

<table>
<thead>
<tr>
<th>Objective</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Executable object code complies with high-level requirements</td>
</tr>
<tr>
<td>2</td>
<td>Executable object code is robust with high-level requirements</td>
</tr>
<tr>
<td>3</td>
<td>Executable object code complies with low-level requirements</td>
</tr>
<tr>
<td>4</td>
<td>Executable object code is robust with low-level requirements</td>
</tr>
<tr>
<td>5</td>
<td>Executable object code is compatible with target computer</td>
</tr>
</tbody>
</table>

a. Users must verify the absence of any errors in the log file generated by KCG.
5.5.2 Combined Testing Process

In a traditional development process, testing combines the search for design, coding, and compilation errors. The Combined Testing Process optimizes the testing effort by using a "divide-and-conquer" approach. It benefits from the qualification of SCADE Display KCG and SCADE Suite KCG and from the characteristics of the generated code:

1. Compliance of the object code to the HLRs and robustness with HLRs is verified by testing HLRs on host and target.

2. Compliance of the source code with the SCADE Display and SCADE Suite LLRs is ensured by the qualification of SCADE Display KCG and SCADE Suite KCG code generators.

3. Compliance of the object code to the source code relies for both SCADE Display and SCADE Suite on a sample-based approach as explained in Appendix D:
   - Regarding SCADE Display, one has to consider several layers, as discussed in 4.5.6, and demonstrate that each of the layers performs correctly:
     - The OGL Tracer tool is provided as an encapsulation of OGLX running on the target. OGL Tracer verifies that, for SGL inputs, the OpenGL functions that are called are the expected ones. Discrepancies may be due to arithmetic or memory considerations in the OGLX library running on the target. This tool is part of the OGLX Certification Kit.
     - The User Context Verification Kit (UCVK) is provided and contains all the graphical patterns (C Source Code containing SGL calls) that can be generated by SCADE Display KCG. Running it on the target platform allows verification of the correct graphical behavior of the compiled and linked source code on the target platform, including CPU and GPU. This is an "end-to-end" verification and it is further discussed in Appendix D-2.
   - Regarding SCADE Suite, code is verified on a representative sample (CVK) that allows to detect C compiler and linker errors on the target platform for the C source code generated by SCADE Suite KCG.

The part of the Combined Testing Process that verifies the SCADE Display and SCADE Suite models perform correctly on the host was described in the previous sections. This section details the part of the combined testing process that is performed on the target. Appendix D provides further details about the justification of the sample-based approach and about the process to build the samples both for CVK (SCADE Suite) and UCVK (SCADE Display).

Figure 5.11 summarizes the Combined Testing Process.
The Combined Testing Process is organized in the following way:

1. The coding and graphics environment is prepared as follows:
   - The source to object code Compiler/Linker is installed in a stable version. Appropriate compiler options are selected (for example, no or little optimization). The same environment is used for hand code and SCADE Display KCG and SCADE Suite KCG code in the project.
   - Source to object code traceability analysis is performed according to CAST Paper 12 [CAST-12]. In particular, it should be verified that the selected compiler options do not compromise traceability.
   - The graphical environment is installed on the target platform. This may include hardware for graphics and graphic software libraries. This may involve certification activities, at the proper level, for both the hardware (DO-254) and the software (DO-178B).

2. For the functions that are hand-coded in the source code language, all traditional verification activities are performed on the complete code (source code review, all level testing, and structural code coverage analysis).

3. For the source code generated by SCADE Display KCG:
   - Low-level testing is performed on a representative sample of the generated C code including SGL function calls and verification of the graphical behavior based on the target graphical environment. Appendix D describes how the UCVK Test Suite containing this sample is developed and used.
   - If there is imported code (e.g., 3D graphics code developed by other means and integrated in the generated code of displays), integration testing between SCADE Display-generated code and this imported code is performed.

4. For the C source code automatically generated by SCADE Suite KCG:
   - A representative sample of the generated code is verified in the same way as manual code, including code review and testing with structural code coverage. Appendix D describes how the CVK Test Suite containing this sample is developed and used.
   - If there is imported code (manual code called by SCADE-generated code), integration testing between Scade code and imported code is performed.

5. For the whole application:
   - The user performs extensive high-level requirements-based software and hardware/software integration testing.
### 5.5.3 Verification summary for the outputs of the integration process

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

**Table 5.8: DO-178B Table A-6 Objectives Achievement**

<table>
<thead>
<tr>
<th>Objective</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Executable object code complies with high-level requirements.</td>
</tr>
<tr>
<td></td>
<td>HLR normal test cases performed on host and target with the help of SCADE LifeCycle QTE + qualification of SCADE Display KCG + qualification of SCADE Suite KCG</td>
</tr>
<tr>
<td>2</td>
<td>Executable object code is robust with high-level requirements.</td>
</tr>
<tr>
<td></td>
<td>HLR robustness test cases performed on host and target + qualification of SCADE Display KCG + qualification of SCADE Suite KCG</td>
</tr>
<tr>
<td>3</td>
<td>Executable object code complies with low-level requirements.</td>
</tr>
<tr>
<td></td>
<td>Ensured by qualification of SCADE Display KCG + qualification of SCADE Suite KCG + compiler verification (with CVK and UCVK) + graphical behavior verification (with UCVK)</td>
</tr>
<tr>
<td>4</td>
<td>Executable object code is robust with low-level requirements.</td>
</tr>
<tr>
<td></td>
<td>Use robust building blocks and perform robustness testing on these blocks</td>
</tr>
<tr>
<td>5</td>
<td>Executable object code is compatible with target computer.</td>
</tr>
<tr>
<td></td>
<td>All tests run on target</td>
</tr>
</tbody>
</table>

a. For integration levels to go beyond the SCADE part, running the corresponding test cases on the host may not be applicable.
6. Verification of the Verification Activities

6.1 Verification Objectives

One now has to assess how well the above-mentioned verification activities have been performed. Table 6.1 summarizes the verification objectives for the verification of verification process results.

Table 6.1: DO-178B Table A-7

<table>
<thead>
<tr>
<th>Objective</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Test procedures are correct.</td>
</tr>
<tr>
<td>2</td>
<td>Test results are correct and discrepancies are explained.</td>
</tr>
<tr>
<td>3</td>
<td>Test coverage of high-level requirements is achieved.</td>
</tr>
<tr>
<td>4</td>
<td>Test coverage of low-level requirements is achieved.</td>
</tr>
<tr>
<td>5</td>
<td>Test coverage of software structure (modified condition/decision) is achieved.</td>
</tr>
<tr>
<td>6</td>
<td>Test coverage of software structure (decision coverage) is achieved.</td>
</tr>
<tr>
<td>7</td>
<td>Test coverage of software structure (statement coverage) is achieved.</td>
</tr>
<tr>
<td>8</td>
<td>Test coverage of software structure (data coupling and control coupling) is achieved.</td>
</tr>
</tbody>
</table>

6.2 Verification of Test Procedures and Test Results

TEST PROCEDURE CORRECTNESS

The following categories of test procedures have to be reviewed for correctness:

- High-level requirements-based test procedures.
- Low-level test procedures of the C samples for compiler and graphical behavior verification (test cases have already been verified by Esterel Technologies).

TEST RESULT CORRECTNESS AND DISCREPANCY EXPLANATION

The results of the above-mentioned test procedures have to be analyzed and discrepancies explained.

6.3 HLR Coverage Analysis

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

All test cases for HLRs described in the form of SCADE Display and SCADE Suite models are based on system requirements allocated to software (SRATS) and they contain links to the SRATS they verify.
All test cases for HLRs described in the form of SCADE Display and SCADE Suite models are based on system requirements allocated to software (SRATS) and they contain links to the SRATS they verify. The analysis of these links confirms full coverage of the HLR by the test cases; otherwise, test cases have to be complemented.

6.4 LLR Coverage Analysis with SCADE Suite MTC

6.4.1 Objectives of model coverage

This section addresses the coverage of the SCADE Suite LLRs.

Model coverage analysis is a means of assessing how far the behavior of a model was explored. It is complementary to traceability analysis and high-level requirements coverage analysis.

Model coverage verifies that every element of the SCADE Suite model (representing a LLR) was dynamically activated when the requirements-based tests are exercised. A primary objective is to detect unintended functions in the software design.

The term “unintended function” requires clarification. At the implementation level, an unintended function is one that exists in the actual software implementation but not by design. Similarly, the existence of an unintended function in software design is unplanned in the software high-level requirements.

One might think that unintended functions are just things that should not be present, that they could be easily detected by scanning the traceability matrix and then removed systematically. Reality is somewhat more complex:

1. “Unintended” does not necessarily mean “wrong” or “extraneous”. It just means “not explicitly intended.” An unintended function may well be necessary and correct but missing in the high-level requirements.

2. Regarding the reason and number of unintended functions, there is a definite difference between the software design and the software code:

   • The software design is very detailed, and the software code must reflect this. Any difference in functionality between the software implementation and its design is either an error in the implementation, a derived requirement that has not been made explicit, or an error/omission in the software requirements. Basically, it is the result of an error.

   • The high-level requirements are usually not as detailed as the definitive software design. Practically, it is often necessary to add details when developing the software design. These additional details must be verified and fed back into the high-level requirements. Many of the “unintended” functions that exist in the software design fill “holes” in the high-level requirements; their presence is often beneficial. However, their explicit identification and verification are required.
3 An unintended function is often not directly visible by the mere presence of a high-level requirement paragraph (text form) or of a Scade operator (graphical form). It may involve some dynamic behavior.

Thus, there is a need to analyze the dynamic behavior of the software design.

These existing criteria are a valuable source of inspiration for SCADE Suite Model Test Coverage (MTC). However, in order to define relevant criteria for SCADE Suite models, one needs to take into account the characteristics of SCADE Suite models:

• A Scade model describes the functionality of software, while a C program describes its implementation. As shown in the next section, this creates a major difference both in terms of abstraction level (feature coverage versus code coverage) and in terms of coverage of multiple occurrences.

• Scade models are based on functional data flows and state machines, while most programming languages and their criteria are sequential.

• Every Scade operator that is not under activation condition is computed at each cycle. This makes the control flow somehow “degenerated” (compared to a traditional program) for the vast majority of Scade models, which contain few activation conditions.

• Regarding the variables definition/use flow, a Scade model explicitly describes this flow, and the language contains semantic integrity rules for this flow: every variable in the model is defined once and only once. The SCADE Suite Semantic Checker verifies such rules.

In order to define proper coverage criteria, this handbook takes into account the following requirements:

• They should capture the activation of elementary functions in a Scade model.

• They should be simple and easy to understand by the persons developing or verifying the high-level or low-level requirements.

• They should be reasonably easy to capture and analyze.

6.4.2 LLR coverage analysis with SCADE Suite MTC

MTC (Model Test Coverage) is a module of SCADE Suite, which allows for measuring the coverage of a Scade model by a high-level requirements-based test suite. The purpose of this measure is to assess how thoroughly the Scade model was exercised. As described in the previous section, the coverage criterion is based on the observation of operator features activation, each operator being associated with a set of characteristic features. Figure 6.1 shows the position of Model Test Coverage within the software verification flow.
The use of MTC is decomposed in the following phases:

1. **Model Coverage Acquisition**: Running test cases in SCADE Suite Simulator, while measuring the coverage of each operator.

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

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

Figure 6.2 illustrates the use of MTC. Coverage of each operator is indicated via colors and percentages. The tool gives a detailed explanation of the operator features that have not been fully covered.
One must further detail Model Coverage Analysis, which allows uncovering model operator features that have not been activated. This may reveal the following deficiencies:

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

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

3 **Dead parts in the SCADE model**: In that case, resolution may consist in removing the dead feature, assessing the effect and the needs for reverification.

---

**Figure 6.2: Using SCADE Suite Model Test Coverage (MTC)**
4 Deactivated parts in the Scade model: In this case, resolution may consist in explaining the reason why a deactivated feature remains in the design.

EXAMPLE 1: INSUFFICIENT TESTING

![Diagram of Non activated Confirmator](image)

- 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?

- Analysis: Resetting the filter here is a correct SW 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.

IMPACT ON UNINTENDED FUNCTIONS

In a traditional process (Figure 6.5), unintended functions are introduced both during the design process and during the coding process. Structural code coverage analysis is needed to detect both categories.
When using SCADE Suite MTC and KCG as illustrated on Figure 6.6, unintended functions are detected and eliminated at model level:

- MTC makes it possible to detect and resolve unintended functions in the LLR (Scade model).
- KCG does not introduce unintended functions into the source code, which exactly reflects the LLR.

**Figure 6.6: Elimination of unintended functions with SCADE Suite MTC and KCG**

### 6.4.3 Model coverage criteria

The goal is to measure the coverage of a Scade model with respect to well-defined coverage criteria on the basis of the execution of requirement-based test cases.

A Scade model can be seen as a network of operators where data flow from one of these operators to another through edges.

It is considered that a Scade model is covered if each operator within the network of operators is covered. The question is: how can one determine that one of these operators is covered?

One can look at this coverage question from two different viewpoints:

1. The use of each instance of each of the operators in the Scade model
2. The contents of each of the operators that are used in the Scade model

**COVERING THE USE OF AN OPERATOR IN A SCADE MODEL**

For covering the use of operators in a Scade model, which measures integration testing, SCADE Suite considers three levels:

1. Operator activation coverage
2. Operator interface coverage
3. User-defined operator coverage
OPERATOR ACTIVATION COVERAGE

The most basic coverage criterion for an operator corresponds to the fact that it has been activated. This criterion is called operator activation coverage, and it is similar to the procedure call criterion for code coverage.

Every flow in a Scade model has a clock. This clock determines when (i.e., in which instant) such a flow is defined and when it can be used.

On Figure 6.7, the effect of the top “when” operator is to make its output flow run on clock clk1. Flow “n”, operator “Op10”, flow “q” and Output3 also run on clock clk1. For instance if Op10 is a counter, it only counts while clock clk1 is true. Similarly the bottom “when” clocks its output on instants where clk1 is false, and “o”, “Op11” and “r” run on instants where clock clkkl is false. The “merge” operator takes as input two complementary flows q and r and produces s, which runs at the clock of the enclosing operator, like “l” and “m”.

In many cases, this criterion may be considered too weak, and thus one needs to propose more relevant criteria.

OPERATOR INTERFACE COVERAGE

When operator interface coverage is selected, the coverage of the operator is achieved only if each input data to the operator has changed while the operator was activated.

In some of the cases, even this criterion is considered too general, and one may need to consider more specifically the coverage of a characteristic functionality that can be expected from the operator. This is the role of the user-defined activation coverage criterion.

USER-DEFINED OPERATOR COVERAGE

Each operator is characterized by a set of features, which are representative of the operator’s elementary functions. It is called a user-defined operator coverage criterion. The following example illustrates this concept.

Example: Confirmator

A Confirmator is a library operator whose purpose is to validate a Boolean input. Its output O is true when its input I remained true during at least N cycles, and is false otherwise.

Assume one measures the coverage of this operator. A reasonable coverage criterion corresponds to covering the following features:

• Any false input triggers a negative output (I is false -> O is false)
• Confirmation (I was true for N cycles -> O true)

COVERING THE DEFINITION OF AN OPERATOR IN A SCADE MODEL

Examining now the question of covering the definition of an operator that is used in a Scade model, it is possible to use the same method as...
described previously in a recursive way until we reach the use of the primitive operators of the Scade language. For those primitive operators, SCADE Suite proposes more elaborate criteria for measuring coverage (meaning the necessity to exercise more test cases for achieving 100% coverage of these operators).

Before doing so, it is necessary to introduce the notions of paths and activation condition.

**PATHS AND ACTIVATION CONDITIONS IN A SCADE DATA FLOW DIAGRAM**

A path in a Scade data flow diagram is either a unit path that connects two edges of the data flow diagram or a sequence of such unit paths. In Figure 6.8 below, \( \{A, E\} \) is a unit path and \( \{A, E, G\} \) is a non-unit path.

The activation condition (\(AC\)) of a path is a Boolean expression that defines when data can flow along a path. The \(AC\) of a path \(P\) is written as \(AC(P)\).

Again considering the example in Figure 6.7, the activation condition of the path between Input6 and Output3 is “\( \text{clk1} \)”. At this point, it is necessary to consider the special cases of state machines, selectors, and Boolean expressions.

**STATE MACHINES COVERAGE**

In case a Scade operator is a state machine diagram, with states and transitions between states, SCADE Suite considers that the state machine is covered if each state and each transition between states is covered, meaning each state was entered and each transition was fired.

**SELECTORS COVERAGE**

Data Flow selection selects a flow among a set of input flows according to a Boolean (\( \text{if} \) operator) or of enumerated selection flows (\( \text{case} \) operator). Contrarily to the effect of a clock, a selection does not impact computation of the incoming flows from a semantic perspective. For instance, if a flow is produced by a counter, this counter must count whatever the value of the selector.
Nevertheless, the coverage criterion proposed for selectors requires that the selection variable of each individual selector takes all its possible values, i.e., true/false for if operator or the enumeration domain for case operator.

For instance, the activation condition of the path between input 5 and output o2 is, as shown in Figure 6.10: “(not c1) or (c3 and (In1=e3) and c5 and (not c2))”. One has to make sure that all activation conditions corresponding to all paths through the above diagram are covered.

**Figure 6.10: Activation Conditions with selectors**

**BOOLEAN EXPRESSIONS COVERAGE**

In addition to the operator interface coverage previously described, Boolean expressions can be assessed with respect to two additional criteria:

- Decision Coverage (DC)
- Modified Decision Coverage (MC/DC) (see [NASA-MCDC])

DC of a Boolean expression is achieved when the output takes the true and false values.
The paths activation conditions for MC/DC can be defined by the following table:

**Table 6.2: MC/DC Activation Conditions for Boolean Operators**

<table>
<thead>
<tr>
<th>Construct</th>
<th>Activation Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diag1" /></td>
<td>AC(a,s) = b</td>
</tr>
<tr>
<td><img src="image2.png" alt="Diag2" /></td>
<td>AC(a,s) = not b</td>
</tr>
<tr>
<td><img src="image3.png" alt="Diag3" /></td>
<td>AC(a,s) = true</td>
</tr>
<tr>
<td><img src="image4.png" alt="Diag4" /></td>
<td>AC(a,s) = b</td>
</tr>
</tbody>
</table>

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

The following example illustrates path activation conditions.

First, considering path (A,S), there is the activation condition \( AC(A,S) = \text{not} \ (B \text{ and } C) \) which may be covered by these test cases:

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

Considering path (B,S), there is \( AC(B,S) = \text{not} \ A \text{ and } C \) which is already covered by test case 2 plus an additional test case:

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

Considering path (C,S), there is \( AC(C,S) = \text{not} \ A \text{ and } B \) which is already covered by test case 3 plus an additional test case:

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

Thus, the above four test cases fully cover the Boolean expression regarding the masking MC/DC coverage criterion.
6.5 Structural Coverage of the Source Code

6.5.1 Control structure coverage

The activity that is described in this section is performed in a context where:

- model coverage was already resolved through the use of MTC for the SCADE Suite models;
- the qualified SCADE Display KCG and SCADE Suite KCG Code Generators are used to generate the code;
- compiler source to object code traceability analysis was performed. In particular, one has verified that the selected compiler options do not compromise the required level of traceability.

The structural coverage objective depends on the safety level; for instance, Level A requires MC/DC.

Structural coverage has to be verified on:

- the complete manual code;
- the C sample for both CVK and UCVK (see Section 5.5);
- the C Source Code generated from SCADE Display models.

Test cases ensuring structural coverage of all the basic C blocks are developed. They are exercised both on host and on target processor. Then, one can assert that every required computational path through the generated code for the primitive computational block level was exercised correctly via the [CVK] and [UCVK] in the target platform environment.

For a given implementation of a low-level requirement as a SCADE Suite operator, model coverage does not necessarily guarantee that every piece of generated code for that low-level requirement was exercised in that context. This is similar to a situation that may occur with runtime libraries. Such libraries have commonly been approved through separate, structure-based testing and analysis in unrelated avionics applications. For any paths exercised by tests that meet the model-coverage criteria, though, the behavior of the source code is correct. If subsequent changes to the low-level requirements activate other paths in the source code, their behavior will also be correct, given that all such paths have been evaluated as part of the [CVK].

6.6 Data and Control Coupling

6.6.1 Definitions

DO-178B 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.”
6.6.2 Control Coupling

6.6.2.1 Control coupling between SCADE Modules

**DESIGN**

[CAST-19] recommends precise description of control coupling in the design as the first major step. At the model level, control coupling is accurately and completely described in terms of operator activation, either at every cycle of the basic clock or subject to derived clocks (activate operators). As an example, if an operator B consumes data from operator A, and if the clocks (i.e., the times where data is present) are not consistent, SCADE Suite KCG rejects such a model.

**CODING**

At the code level, SCADE Suite KCG ensures that this control coupling at the model level is exactly reflected in the generated code.

**TESTING**

SCADE Suite MTC allows for verifying that the activation of all operators has occurred, thus covering effective control coupling.

6.6.3 Data Coupling

6.6.3.1 Data coupling between SCADE Modules

**DESIGN**

[CAST-19] recommends precise description of data coupling in the design as the first major step.

With SCADE Suite, at the model level, data coupling is accurately and completely described in terms of operators’ interfaces and fully explicit operator connections. Consistency is verified automatically with the SCADE Suite Semantic Checker. As an example, if an operator B consumes data from operator A, and if the data types or the clocks (i.e., the times where data is present) are not consistent, SCADE Suite KCG rejects such a model.

With SCADE Display, at the model level, data coupling is accurately and completely described in terms of objects, attributes, plugs, and groups. Consistency is fully verified with SCADE Display Editor and later with SCADE Display KCG Code Generator. For example, when a variable is used as a plug for two properties of different types, SCADE Display Editor detects such inconsistency.
CODING
At the code level, SCADE Display KCG and SCADE Suite KCG qualified code generators ensure that this data coupling at the model level is exactly reflected in the generated code.

TESTING
SCADE Suite MTC allows verifying that all input flows of all operators have changed, which demonstrates that during testing these flows have been exercised, thus confirming effective data coupling.

6.6.3.2 Data coupling between SCADE Suite and SCADE Display-generated code
Since the generation of the wrapper code between SCADE Suite and SCADE Display-generated code is not qualified, data coupling is verified manually in the traditional way.

6.6.3.3 Data coupling between SCADE Suite generated code and manual code
Data coupling between SCADE Suite generated code and manual code is verified manually in the traditional way.

6.6.3.4 Control coupling between SCADE Suite and SCADE Display-generated code
Since the generation of the wrapper code between SCADE Suite and SCADE Display-generated code is not qualified, control coupling is verified manually in the traditional way.

6.6.3.5 Control coupling between SCADE Suite-generated code and manual code
Control coupling between SCADE Suite-generated code and manual code is verified manually in the traditional way.
6.7 Summary of Verification of Verification

Table 6.3 summarizes verification objectives and methods for the verification activities.

### Table 6.3: DO-178B Table A-7 Objectives Achievement

<table>
<thead>
<tr>
<th>Objective</th>
<th>Verification Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Test procedures are correct. Test cases and procedures review N.B. concerns both host and target</td>
</tr>
<tr>
<td>2</td>
<td>Test results are correct and discrepancies are explained. Test results review N.B. concerns both host and target</td>
</tr>
<tr>
<td>3</td>
<td>Test coverage of high-level requirements is achieved. SCADE LifeCycle Requirements Management Gateway traceability analysis</td>
</tr>
<tr>
<td>4</td>
<td>Test coverage of low-level requirements is achieved. Model Test Coverage analysis</td>
</tr>
<tr>
<td>5</td>
<td>Test coverage of software structure (modified condition/decision) is achieved. MC/DC model coverage analysis plus comparison of test results between host and target MC/DC code coverage analysis of the SCADE Display-generated code</td>
</tr>
<tr>
<td>6</td>
<td>Test coverage of software structure (decision coverage) is achieved. DC model coverage analysis plus comparison of test results between host and target DC code coverage analysis of the SCADE Display-generated code</td>
</tr>
<tr>
<td>7</td>
<td>Test coverage of software structure (statement coverage) is achieved. DC model coverage analysis plus comparison of test results between host and target Statement coverage analysis of the SCADE Display-generated code</td>
</tr>
<tr>
<td>8</td>
<td>Test coverage of software structure (data coupling and control coupling) is achieved. Semantic check of both SCADE Display and SCADE Suite models, MTC coverage and manual verification of integration between SCADE Suite and SCADE Display-generated code and integration with manual code</td>
</tr>
</tbody>
</table>
Appendixes and Index
A References


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>
## B Acronyms and Glossary

### ACRONYMS

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Advisory Circular</td>
</tr>
<tr>
<td>A/C</td>
<td>Aircraft</td>
</tr>
<tr>
<td>ADI</td>
<td>Attitude Direction Indicator</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ARP</td>
<td>Aerospace Recommended Practices</td>
</tr>
<tr>
<td>CAST</td>
<td>Certification Authorities Software Team</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The-Shelf</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CVK</td>
<td>Compiler Verification Kit</td>
</tr>
<tr>
<td>DAL</td>
<td>Development Assurance Level</td>
</tr>
<tr>
<td>DC</td>
<td>Decision Coverage</td>
</tr>
<tr>
<td>DV</td>
<td>Design Verifier</td>
</tr>
<tr>
<td>EASA</td>
<td>European Aviation Safety Agency</td>
</tr>
<tr>
<td>EUROCAE</td>
<td>European Organization for Civil Aviation Equipment</td>
</tr>
<tr>
<td>FAA</td>
<td>Federal Aviation Administration</td>
</tr>
<tr>
<td>FHA</td>
<td>Functional Hazard Analysis</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HLR</td>
<td>High-level Requirement</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>IIR</td>
<td>Infinite Impulse Response</td>
</tr>
<tr>
<td>KCG</td>
<td>Qualified Code Generator</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>LLR</td>
<td>Low-level Requirement</td>
</tr>
<tr>
<td>MC/DC</td>
<td>Modified Condition/Decision Coverage</td>
</tr>
<tr>
<td>MTC</td>
<td>Model Test Coverage</td>
</tr>
<tr>
<td>N/A</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>N.B.</td>
<td>Nota Bene</td>
</tr>
<tr>
<td>PSSA</td>
<td>Preliminary System Safety Assessment</td>
</tr>
<tr>
<td>QTE</td>
<td>Qualified Test Environment</td>
</tr>
<tr>
<td>RM</td>
<td>Requirements Management</td>
</tr>
<tr>
<td>ROI</td>
<td>Return On Investment</td>
</tr>
<tr>
<td>RTCA</td>
<td>Radio Technical Commission for Aeronautics, RTCA, Inc.</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating System</td>
</tr>
<tr>
<td>SAE</td>
<td>Society of Automotive Engineers</td>
</tr>
<tr>
<td>SCADE</td>
<td>Safety Critical Application Development Environment</td>
</tr>
<tr>
<td>SCCI</td>
<td>Source Code Control Interface</td>
</tr>
<tr>
<td>SGL</td>
<td>Standard Graphic Library</td>
</tr>
<tr>
<td>SQA</td>
<td>Software Quality Assurance</td>
</tr>
<tr>
<td>SRATS</td>
<td>System requirements allocated to software</td>
</tr>
<tr>
<td>SSA</td>
<td>System Safety Assessment</td>
</tr>
<tr>
<td>SSM</td>
<td>SCADE State Machine</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>TAS</td>
<td>Tool Accomplishment Summary</td>
</tr>
<tr>
<td>TCI</td>
<td>Tool Configuration Index</td>
</tr>
<tr>
<td>TORD</td>
<td>Tool Operational Requirements Data</td>
</tr>
<tr>
<td>UCVK</td>
<td>User Context Verification Kit</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst Case Execution Time</td>
</tr>
</tbody>
</table>
GLOSSARY

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.

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) that, as a result of a design error, cannot be executed (code) or used (data) in an operational configuration of the target computer environment and 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.

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.

Hardware/software integration
The process of combining the software into the target computer.
High-level requirements
Software requirements developed from analysis of system requirements.

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 that 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 varying just that condition, while holding fixed all other possible conditions.

Robustness
The extent to which software can continue to operate correctly despite invalid inputs.

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 oath or to verify compliance with a specific requirement.

Tool qualification
The process necessary to obtain certification credit for a software tool within the context of a specific airborne system.

Traceability
The evidence of 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-178B Qualification of SCADE Display KCG and SCADE Suite KCG

### C-1 What Does Qualification Mean and Imply?

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

Within DO-178B, development tools are those whose output is part of the embedded software; thus, they can introduce errors in that embedded software.

The way to achieve the qualification of a development tool is as follows:

- If a software development tool is to be qualified, the software development processes for the tool should satisfy the same objectives as the software development processes of embedded software.
- The software level assigned to the tool should be the same as that of the embedded software it produces, unless the applicant can justify a reduction in software level of the tool to the certification authority.

In summary, users have to make sure that if they intend to use a tool, that tool was developed in such a way that qualifies for its intended role (in our case, development) and at the level of the target software.

### C-2 Development of SCADE Display KCG and SCADE Suite KCG

The SCADE Display KCG Code Generator was developed to meet the objectives of DO-178B for Level A. These objectives were described in the following documents, which have been audited by the Certification Authorities on a number of past projects:

- The **Tool Operational Requirements Data** (TORD) contains the software requirements specification and the following documents:
  - Interface documents for SGL, BMP, SGDK, and RF (Reticule Files for advanced objects)
  - Version content
- The **Tool Qualification Plan** (TQP) presents the strategy and the organization for the Qualification of the SCADE Display KCG Code Generator and refers to other project plans.
- The **Tool Accomplishment Summary** (TAS) presents the compliance status with the Tool Qualification Plan, the usage conditions, and the possible limitations of the tool.
- The **Software Configuration Index** (SCI) presents the configuration status of the tool.

The SCADE Suite KCG Code Generator was developed to meet the objectives of DO-178B for Level A. These objectives were described in
the following documents, which have been audited by the Certification Authorities on a number of past projects:

- The **Tool Operational Requirements Data (TORD)** presents the software requirements specification and is split into the following documents:
  - Requirements Data of SCADE Suite KCG.
- The **Tool Qualification Plan (TQP)** presents the strategy and the organization for the Qualification of the SCADE Suite KCG Code Generator and refers to other project plans.
- The **Tool Accomplishment Summary (TAS)** presents the compliance status with the Tool Qualification Plan, the usage conditions, and the possible limitations of the tool.
- The **Tool Configuration Index (TCI)** presents the configuration status of the tool.

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

<table>
<thead>
<tr>
<th>Data</th>
<th>DO-178B &amp; FAA 8110.49 Reference</th>
<th>SCADE Suite KCG Package</th>
<th>DO-178B Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool Qualification Plan</td>
<td>Delivered in the Certification Kit</td>
<td>Tool Qualification Plans of SCADE Display KCG and SCADE Suite KCG</td>
<td>12.2.3.a(1), 12.2.3.1, &amp; 12.2.4</td>
</tr>
</tbody>
</table>
| Tool Operational Requirements | Delivered in the Certification Kit | For SCADE Suite KCG:  
  - Version Content  
  - Software requirements data of SCADE Suite KCG  
  - Scade 6 Language Reference Manual  
  For SCADE Display KCG:  
  - Version Content  
  - Four Interface Documents (SGL, BMP, SGDK, RF) | 12.2.3.c(2) & 12.2.3.2 |

**C-3 SCADE Display KCG and SCADE Suite KCG Life-Cycle Documentation**

Table C.1 describes the documents that must be considered by the Certification Authorities during the qualification audits of SCADE Display KCG and SCADE Suite KCG. Data is available for both tools.

This list conforms to FAA Order 8110.49 “Software Approval Guidelines,” Chapter 9 [Order 8110.49].

Documents that are “Submit” must be submitted to the Certification Authorities. They become part of the SCADE Display KCG and SCADE Suite KCG Certification Kits that are delivered to SCADE users.

Documents that are “Available” can be audited by the Certification Authorities at Esterel Technologies.
The OGLX library is developed by Esterel Technologies following the DO-178B guidelines at Level A. The documents that are necessary for certification of this library are provided in a certification kit that includes the following:

- Software Requirements
- Software Verification Records
- Software Configuration Index
- Software Accomplishment Summary
- OGL Tracer Tool Operational Requirements

Other certification data can be audited by the Certification Authorities.

### Table C.1: Documents required for KCG qualification audit by Certification Authorities (Continued)

<table>
<thead>
<tr>
<th>Data</th>
<th>DO-178B &amp; FAA 8110.49 Requirement</th>
<th>SCADE Suite KCG Package</th>
<th>DO-178B Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool Accomplishment Summary</td>
<td>Delivered in the Certification Kit</td>
<td>Tool Accomplishment Summaries of SCADE Display KCG and SCADE Suite KCG</td>
<td>12.2.3.c(3) &amp; 12.2.4</td>
</tr>
<tr>
<td>Tool Verification Records</td>
<td>Delivered in the Certification Kit</td>
<td>Accessible at Esterel Technologies premises</td>
<td>12.2.3</td>
</tr>
<tr>
<td>Tool Qualification Development data</td>
<td>Delivered in the Certification Kit</td>
<td>Accessible at Esterel Technologies premises</td>
<td>12.2.3</td>
</tr>
<tr>
<td>Software Configuration Index</td>
<td>Delivered in the Certification Kit</td>
<td>Software Configuration Indexes of SCADE Display KCG and SCADE Suite KCG</td>
<td>9.3</td>
</tr>
</tbody>
</table>
D  SCADE Suite Compiler Verification Kit (CVK) and User Context Verification Kit (UCVK)

D-1  Compiler Verification Kit (CVK)

D-1.1 CVK Product Overview

WHAT SCADE SUITE CVK IS
While SCADE Suite KCG qualification ensures that source code conforms to LLRs developed with SCADE Suite, the purpose of CVK is to verify that the C compiler correctly compiles the C code generated by KCG.

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.

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.

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

Figure D.1: Role of KCG and CVK in verification of user development environment
D-1.2 Motivation for Sample-Based Testing

The source code generated by SCADÉ 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 SCADÉ application is in the scope of the limits covered by CVK. SCADÉ 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 position paper [CAST-25] (§3.2.4.2) for compiler verification activities in the case of automatically generated code:

“The applicant performs normal testing activities on generated source code that comprises all source code programming constructs specified in the coding standards in order to demonstrate that the object code generated from this source code is correct and does not introduce erroneous code that is not traceable at the source code level”.

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 SCADE sample, covering all SCADE constructs, is built as material for code generation.
   b. Each elementary C construct and their combination are generated from a SCADE sample root node 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.
   c MC/DC coverage at Scade model level
      (complementary objective not requested by the
       CVK methodology).

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-1.4 Use of SCADE Suite CVK

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

• The CVK User’s Manual [CVK_UM] is an
  appendix of the customer’s verification plan, more
  precisely in the qualification plan of the user’s
  development environment.

• The CVK test suite is instantiated for the
  customer’s verification process, more precisely in
  the qualification process of one’s development
  environment, for the verification of the compiler.

Users must verify that the complexity of their
model (depth of expressions, data structures, and
call tree) is lower than the one of the model in
CVK. Otherwise, they shall either upgrade CVK
accordingly or decompose the SCADE model.

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

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

• The C sample is regenerated by KCG from the
  SCADE sample, with specified KCG options and
  is compared to the provided Reference C sample.
  Comparison is performed with the qualified Diff
  tool.

• From the C sample, the C compiler/linker
  generates an executable. Note that the C sample is
  always taken from the delivered reference sample,
  not from the regenerated C sample.

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

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

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

## D-2 User Context Verification Kit (UCVK)

### D-2.1 UCVK Overview

**WHAT UCVK IS**

UCVK is a test suite, whose purpose is to verify that:

- a compiler correctly compiles code generated by SCADE Display KCG.
- a HW/SW graphical platform provides correct rendering of SCADE Display objects, including in their static and dynamic behavior.

**WHAT UCVK IS NOT**

1. UCVK is NOT a validation suite of the C compiler (see Section D-1.1)
2. UCVK is NOT a means to validate the HW/SW platform for its general use.

**ROLE OF UCVK**

UCVK is a test suite: it is part of verification means of SCADE Display KCG users.

Figure D.5 shows the complementary roles of SCADE Display KCG and UCVK in the qualification of the development environment.
of the customer, including its use of the C compiler and the graphical libraries of the target platform.

**Figure D.5:** Role of KCG and UCVK in verification of user development environment

**UCVK SAMPLE CHARACTERISTICS**

The sample that is generated by the SCADE Display models of UCVK exhibits the following characteristics:

- It contains all individual C constructs and all SGL library calls [SGL_ID] that can be generated from SCADE Display KCG.
- It contains all C code constructs that can be generated from SCADE Display KCG.
- It contains all SGL library calls that can be generated for SCADE Display graphical objects.
- It contains all properties of primitives, variables and constants.

**D-2.2 Motivation for Sample-Based Testing**

The source code generated by SCADE Display KCG is a small subset of the C language with a low level complexity, augmented by the library calls to the functions of the SGL library [SGL_ID].

Regarding the generated C code, it has the following characteristics:

- It contains all individual C constructs and all SGL library calls [SGL_ID] that can be generated from SCADE Display KCG.
- It contains all C code constructs that can be generated from SCADE Display KCG.
- It contains all SGL library calls that can be generated for SCADE Display graphical objects.
- It contains all properties of primitives, variables and constants.

**D-2.3 Use of UCVK**

The following items are part of UCVK:

- A Specification file, BEHAVIOR.sgf, to be used as an input of SCADE Display KCG 6.0. This specification was built according to the rules described above (See UCVK sample characteristics in Section D-2.1).
- A Test Specification document describing the test suite to be executed and the expected results. It contains a description of the animation laws to be used on the target to execute the C code generated from the BEHAVIOR.sgf model.
These animation laws are such that, while running the test cases, 100% MC/DC structural coverage of the sample C source code is achieved.

- A description file in HTML format of BEHAVIOR.sgf
- A PC executable to be used as a reference for dynamic behavior analysis when executing the test suite on the target platform.

We now describe how UCVK has to be used with SCADE Display KCG 6.0 version.

The user launches the testing activity with the following steps:

- Generating the C code using SCADE Display KCG 6.0 from the BEHAVIOR.sgf file contained in the UCVK delivery.
- Compiling and integrating the generated source C code on the target environment.
- Compiling and integrating on target environment, the C code corresponding to animation laws contained in the UCVK delivery.
- Launching the executable code on the target environment.
- Comparing the dynamic behavior of the executable on target with the dynamic behavior of the PC executable dynamic behavior contained in the UCVK delivery.

The user makes the following verifications:

- Checking that SCADE Display KCG 6.0 is properly installed. An incomplete or corrupted installation is likely to produce incorrect results.
- Checking that the right version of SCADE Display KCG 6.0 is correctly installed.
- Performing a visual check of the display obtained in target environment:
  a according to acceptance criteria defined in the UCVK test specification;
  b comparing with images contained in the HTML model description (for checking the static behavior);
  c comparing with PC execution on host PC (for checking the dynamic behavior).
Index

A
Accuracy 54
ANAC 3
Architecture 34
ARP 4754
  overview 3
ARP 5056
  overview 3

C
C elementary constructs 103
C sample 102
C subset 102
CAAC 3
Causality 20
Clock 24
Coding process 39
Compiler Verification Kit 101
Concurrency 21, 25
Configuration Management 49
Coverage 73, 75
Coverage analysis
  test coverage 10
  with MTC 74
Coverage criteria
  structural coverage 11
Coverage resolution
  structural coverage 11
CPU 13, 62, 63, 66, 70
CVK 101
CVK test execution and MC/DC 102

D
Data typing 23
Dependency 21
Design process 33

E
Design standard 55
Design Verifier 59
Development assurance levels 5
Development processes 6, 7
Discrete control 22
DO-178B
  overview 4
  processes 6

F
EASA 3
Equations 19
EUROCAE 4

G
EUROCAE 4

H
High-level requirements 7, 8
HLR
  see High-level requirements

I
Initialization 21
Input/Output 35
Integral processes 6
Integration process 43
Interface 19

K
KCG 25, 55, 71, 72, 79, 85
  acronym 93
DO-178B qualification 97
SCADE Display 30
SCADE Display and SCADE Suite 68, 70
SCADE Display and UCVK 105
SCADE Suite and CVK 101

L
Layer 18, 25, 34, 55
LLR
  see Low-level requirements
Local variables 19
Logic 37, 38
Low-level requirements 7, 9
  development in SCADE 35

M
MC/DC 11
Model-based 25
Modified Condition/Decision Coverage
  see MC/DC
Modular 20
MTC 74, 75
Multitasking 47

O
OGL Tracer 70, 99
OGLX 47, 70, 99
Operator 19
Index

P
Partitioning 66
Planning processes 6
Plugs 18, 44, 85
Property 61

Q
QTE 72
acronym 93
Qualification 68, 97

R
Regulation 36
Requirements Management
Gateway 30
Requirements process 31
Robustness 38
RTCA 4
RTOS 45

S
SAE 3
SCADE State Machines 22
Scheduling 44
SGL 106
acronym 93
Simulation 58
Software architecture 7
Software Design Standards 9
Source code 9
SRATS 73
SSM 22
Stack usage analysis 65
Stack Verifier 66
Standards 68
Structural coverage 84
Symbology
layer 18, 55
System Engineering 16

T
TAS
acronym 93
Task integration
RTOS 45
Tasking 44
TCI
acronym 93
Teamwork 48
Test procedures 73
Test results 73
Testing 10, 51
Timing Verifier 62, 63
TORD
acronym 93
Traceability 41, 53

U
UCVK 101
acronym 93
Unintended functions 74, 78

V
Validation 51
Verification 51
Verification processes 8

W
WCET analysis 62
Contact Information

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

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

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

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

Copyright © 2012 Esterel Technologies SA. All rights reserved Esterel Technologies SA.