ESTEREL Technologies

Methodological Handbook

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

CONTACTS

Legal Contact

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

US Technical Support

Esterel Technologies Inc.
100 View Street, Suite 208
Mountain View, CA 94041
UNITED STATES
Phone: +1 650 641 3250

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
PO Box 7995
Crowthorne RG45 9AA
UNITED KINGDOM
Phone: +44 1344 780898

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 - Str. 13b
D- 85521 Ottobrunn - Riemerling
GERMANY
Phone: +49 89 608 75537

China

Esterel Technologies
No. 10-401, Shanghai Pudong Software Park
498, GuoShouJing Road
201203, Shanghai
P.R. CHINA
Phone: +86-21 5027 1120

Shipping date: February 2006 Revision: SC-HB-DO178B - SC/2012u6-KCG4.2

Abstract

This document addresses the issue of cost and productivity in the development of safe embedded software for avionics applications. Such projects, driven by the DO-178B guidelines, traditionally require very difficult and precise development and verification efforts. This handbook reviews the regulatory guidelines and then presents the optimization of the development and verification processes that can be achieved with the SCADE Suite methodology and tools. SCADE Suite supports the automated production of a large part of the development life-cycle elements. The effect of using SCADE Suite together with the qualified KCG 4.2 Code Generator will be 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

List of Figures

List of Tables

1. Document Background, Objectives, and Scope

1.1 Background

A traditional situation in the avionics industry is that the function and architecture of an embedded computer system (*i.e.*, Flight Control, Braking, Cockpit Display, etc.) are defined by system engineers; the associated control laws are developed by control engineers using some informal notation or a semi-formal notation mostly based on schema-blocks and/or state machines; and the embedded production software is finally specified textually and coded by hand in C and Ada by software engineers.

In this context, qualified automatic 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 control laws as described above, and to automatically generate the C code from this model using a qualified code generator, in the sense of DO-178B, thus bringing the following advantages to the development life cycle:

- When a proper modeling approach is defined:
  - It fulfills the needs of control engineers, typically using such notations as data flow diagrams and state machines.
  - It fulfills the needs of software engineers by supporting the accurate definition of the software requirements and 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 setting up efficient new processes to ensure that safety criteria are met.
- It saves coding time, as this is automatic.
- It saves a significant amount of verification time, as the use of such tools guarantees that the generated source code agrees with the model.
- It allows identifying problems earlier in the development cycle, since most of the verification activities can be done at model level.
- It reduces the change cycle time, since modifications can be performed at model level and code can automatically be regenerated.

1.2 Objectives and Scope

This document gives a careful explanation of the system and software life cycles as described in the ARP 4754 and DO-178B guidelines. It then explains how to use both the proper modeling techniques and automatic code generation from models to obtain drastic productivity improvements.

The document is organized as follows:

Section 2. This section provides an introduction to the regulatory guidelines of ARP 4754 and DO-178B that are used when developing

embedded avionics software. It then describes the main challenges in the development of safety-critical applications, in terms of specification, verification, and efficiency of the resulting software.

Section 3. This section presents an overview of the SCADE Suite methodology and tools. It first demonstrates how SCADE maintains the highest-quality standards, while reducing costs based on a "correct-by-construction" approach and the use of a qualified automatic code generator, according to the following points:

- A unique and accurate software description, which enables the prevention of many specification or design errors, can be shared among project participants.
- 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 SCADE tools, including the use of the KCG 4.2 qualified code generator. It also explains how SCADE-generated code can be integrated on target, including when it has to interact with an RTOS (Real Time Operating System).

Section 5. and Section 6. These sections present the verification activities that take place when SCADE is used, including model-level verification with the Simulator, the Design Verifier, and the Model Test Coverage tool, as well as specific verification activities aimed at detecting compiler errors.

Appendix A lists the references.

Appendix B is a glossary of terms.

Appendix C details the KCG 4.2 qualification process.

Appendix D details the Compiler Verification Kit (CVK).

2. Development of Safety-Related Airborne Software

2.1 ARP 4754 and DO-178B Guidelines

2.1.1 Introduction

The avionics industry requires that safety critical software be assessed according to strict certification authority[1] 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.

2.1.2 ARP 4754

ARP 4754 was defined in 1996 by the SAE (Society of Automotive Engineers).

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 guidance material in this document was developed in the context of Federal Aviation Regulations (FAR) and Joint Airworthiness Requirements (JAR) Part 25. In general, this 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 with in the DO-178B (RTCA)/ED12B (EUROCAE) document. Coverage of complex hardware aspects of design are dealt with in RTCA document DO-254.

2.1.3 DO-178B

DO-178B/ED-12 was first published in 1992 by RTCA (Requirements and Technical Concepts for Aviation) 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

1. For example, the United States Federal Aviation Administration (FAA), the European Aviation Safety Agency (EASA), Transport Canada, etc.

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.

2.1.4 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 are summarized in Figure 2.1.

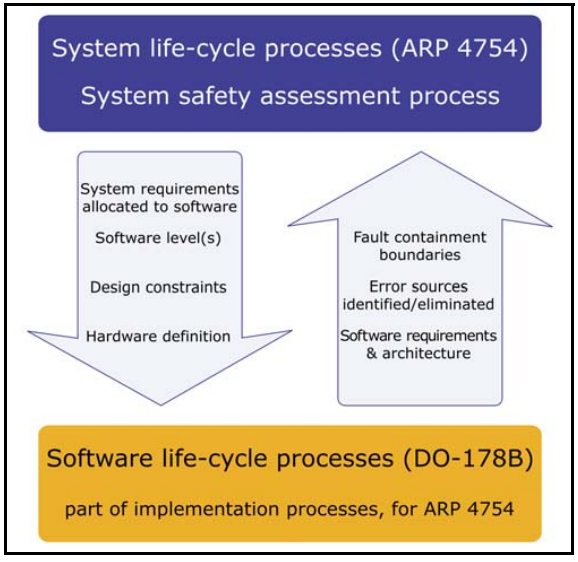


Figure 2.1: Relationship between ARP 4754 and DO-178B processes

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 software), DO-254 (for complex 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."

2.1.5 Development assurance levels

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

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

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

This handbook mainly targets level A, B and C software.

2.1.6 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 that appropriate objectives are defined. For instance:
  a Development assurance level of the software.
  b Design standards.
- Define procedures for the verification of the objectives. For instance:
  a Verify that design standards are met and that the design is complete, accurate, and traceable.
  b Develop and apply requirements-based test cases.

- 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.

2.1.7 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 processes that define and coordinate 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.

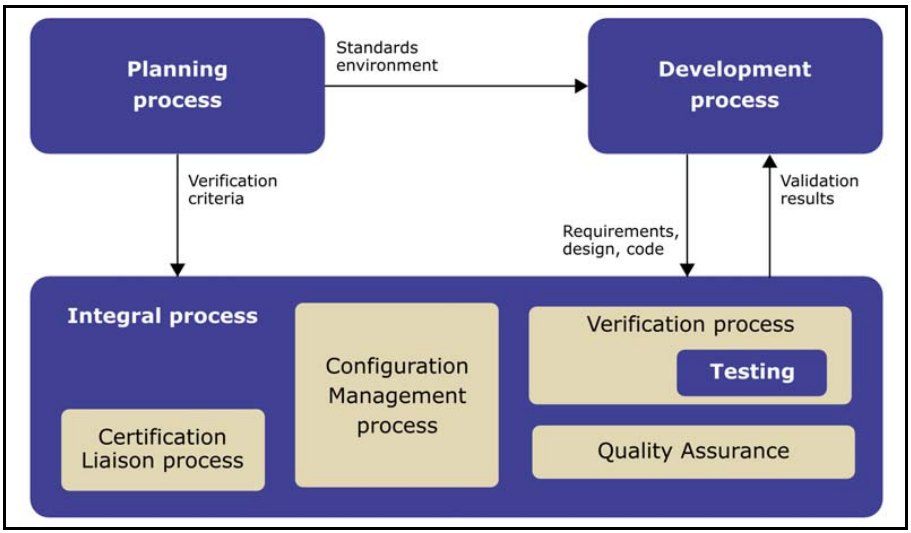


Figure 2.2: DO-178B life-cycle processes structure

In the remainder of this document we will 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 object code and builds up to the integrated system or equipment.

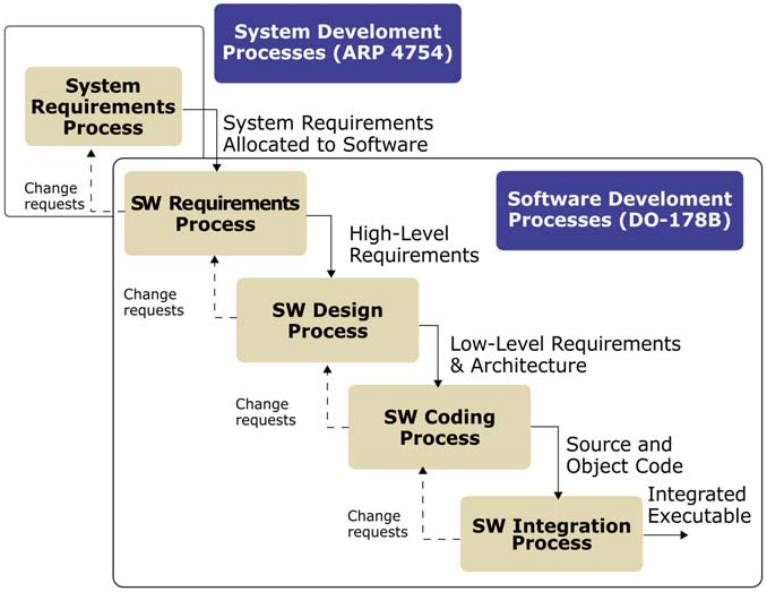


Figure 2.3: DO-178B development processes

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 are implemented as source code.

The source code is compiled and linked by the integration process up to 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 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 process; 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 and 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 that may have been introduced during the software design process. These reviews and analyses confirm that the software low-level requirements satisfy these objectives:

a Compliance with high-level requirements: the software low-level requirements satisfy the software high-level requirements.

b Accuracy and consistency

c 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 (such as bus loading), system response times, and input/output hardware.

d Verifiability: each low-level requirement can be verified.

e Compliance with the Software Design Standards (defined by the software planning process).

f Traceability: the objective is to ensure that all high-level requirements were taken into account in the development of the low-level requirements.

g Algorithm aspects: ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities (*e.g.*, mode changes, crossing value boundaries).

h The SW architecture is compatible with the HLR, it is consistent and compatible with the target computer, is verifiable, and conforms to standards.

i 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:

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

b Compliance with the software architecture: The source code matches the data flow and control flow defined in the software architecture.

c Verifiability: The source code does not contain statements and structures that cannot be verified, and the code does not have to be altered to test it.

d 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 low-level requirements.

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.

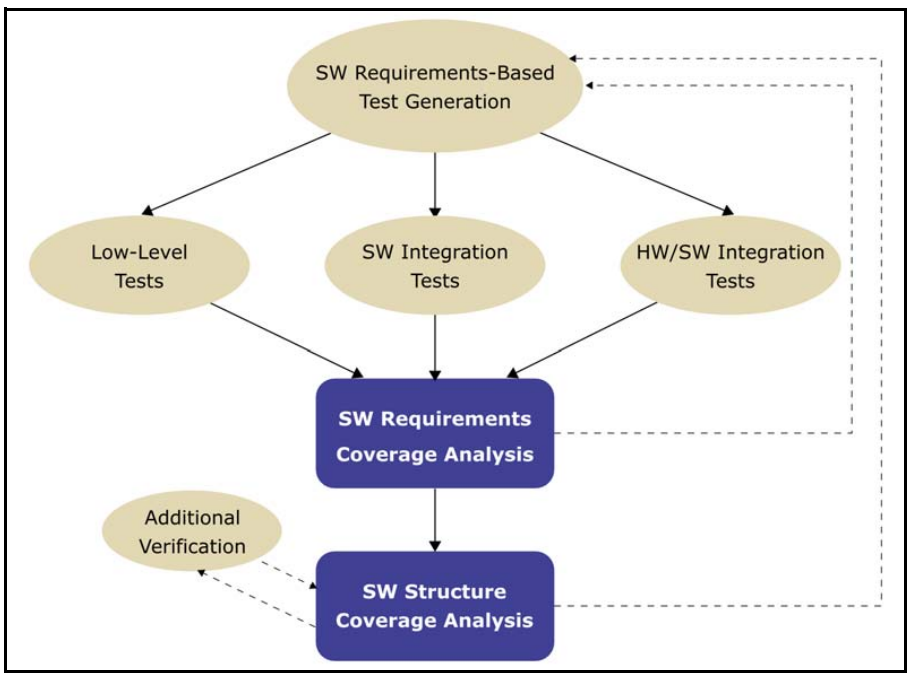


Figure 2.4: DO-178B testing processes

There are three types of testing activities:

- Low-level testing: to verify the implementation of software low-level requirements.
- Software integration testing: to verify the interrelationships between software requirements and components and to verify the implementation of the software requirements and software components within the software architecture.
- Hardware/software integration testing: to verify correct operation of the software in the target computer environment.

As shown in Figure 2.4, 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-248, FAQ#43). Note that requirements coverage is an absolute prerequisite to this step.

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 has been exercised.
- Level B: Decision coverage is required; this means that every decision has taken all possible outcomes at least once (*e.g.*, then/else for an "if" construct) and that every entry and exit point in the program has been 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 has been invoked at least once.
  - Every decision has taken all possible outcomes.
  - Each condition in a decision has been shown to independently affect that decision's outcome (this is 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

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

2.4 What Are the Main Challenges in the Development of Airborne Software?

This section introduces the main challenges that have to be faced when developing safety-related airborne software.

2.4.1 Avoiding multiple descriptions of the software

In such a development life cycle, the software is described in several phases and documents:

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

At each step, it is important to avoid as much as possible rewriting the software description.

This rewriting would not only be expensive, it would also be error-prone. And there is a major risk of inconsistencies between different descriptions. This necessitates devoting a significant effort to verifying compliance 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 and diagrams. It is an everyday experience that natural language is subject to 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 the dynamic behavior: for instance, how to 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).

The programmers generally have a limited understanding of the system, which makes it vulnerable to ambiguities in the specification. Moreover, the code they produce is generally not understandable by the author of the system or high-level requirements.

In the traditional approach, the combined risk of interpretation error and coding errors is so high that a major part of the life-cycle 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. It should be easy and efficient to retarget to a given processor.

2.4.5 Finding specification and design errors as early as possible

A significant number of specification and design errors are only detected during software integration testing.

One reason is that the requirement/design specification is often ambiguous and subject to interpretation. Another reason is that it is difficult for a human reader to understand details regarding dynamic behavior of the software without being able to exercise it. And the main reason is that, in a traditional process, this is the first time where one can exercise the software. This is very 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 and cost of updates

There are many sources of changes in the software, ranging from bug fixing, function improvement to 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-related airborne software is much higher than for other non safety-related commercial software. For Level A software, the overall verification cost (including testing) may account for up to 80% of the budget. Verification is also a bottleneck for the project completion. So, clearly, any change in the speed and/or cost of verification has a major impact on the project time and budget.

The objective of this document is to show how to retain a complete and thorough verification and validation process, while dramatically improving the efficiency of this process. The methods we will describe reach at least the level of quality achieved by traditional means, by optimizing the whole development process.

2.4.8 Providing an efficient way to store Intellectual Property (IP)

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

- Textual system and software requirements
- Graphical models of the software requirements (*e.g.*, regulation laws)
- Source code
- Test cases
- Other

3. Model-Based Development with SCADE Suite and KCG

3.1 What Is SCADE?

SCADE ORIGIN AND APPLICATION DOMAIN

The name SCADE stands for “Safety-Critical Application Development Environment.” This name is used both for the SCADE notation and for the SCADE Suite software development environment. Its purpose is to create a bridge between control engineering and software engineering activities.

SCADE has been designed from the beginning 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] in Appendix B. From the beginning, it has been designed with companies developing safety-critical software.

SCADE has been used from the start on an industrial basis for the development of safety-critical software such as flight control (Airbus, Eurocopter), nuclear power plant control (Schneider Electric), and railway switching systems (CSEE Transport).

SCADE addresses the applicative part of hard real-time software, as illustrated in Figure 3.1. This is usually the most complex and changeable aspect of software, containing complex decision logic, filters, and control laws. It typically represents 60% to 80% of the software embedded in an airborne computer.

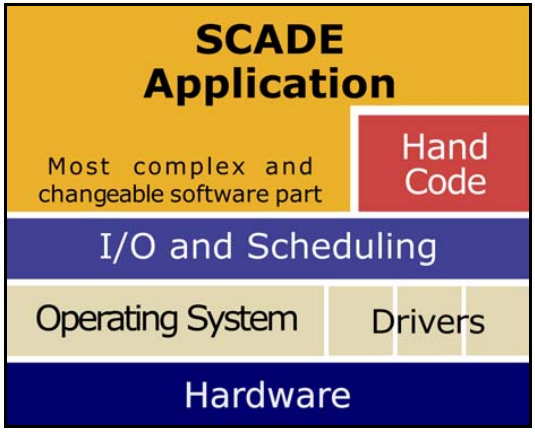
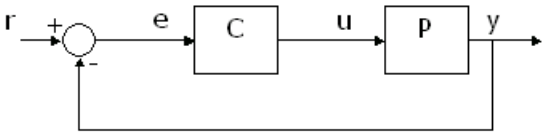


Figure 3.1: SCADE addresses the applicative part of software

A BRIDGE BETWEEN CONTROL ENGINEERING AND SOFTWARE ENGINEERING

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

- Control engineers describe systems and their controllers using block diagrams and transfer functions (s form for continuous time, z form for discrete time), as shown below in Figure 3.2.



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

Figure 3.2: Control engineering view of a Controller

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

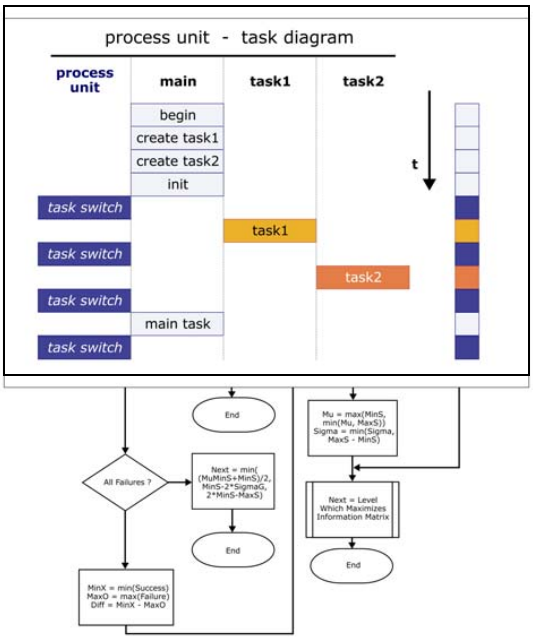


Figure 3.3: Software engineering view of a Controller

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

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

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

3.2 SCADE Modeling Techniques

3.2.1 Familiarity and accuracy reconciled

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

- Block diagrams to specify the algorithmic part of an application, such as control laws and filters.
- Safe State Machines (SSM) to model the behavior.

What the modeling techniques of SCADE add is a very rigorous view of these well-known but often insufficiently defined formalisms. SCADE has a formal foundation and provides a precise definition of concurrency; it ensures that all programs generated from SCADE behave deterministically.

SCADE allows for automatic generation of C code from these two formalisms.

We will now describe more precisely their characteristics and the way an application designed in SCADE executes on a target platform.

3.2.2 SCADE node

The basic building block in SCADE is called a node. A node is a user-defined function, built from lower-level nodes, down to predefined operators (*e.g.*, logical, arithmetic, delay, etc.) A node can be represented either graphically (see Figure 3.4), or textually (see Table 3.1 below).

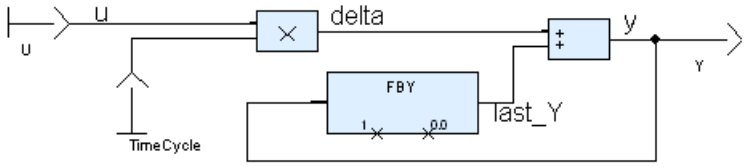


Figure 3.4: Graphical notation for an integrator node

A node is a functional module made of the following components:

Table 3.1: Components of SCADE functional modules: nodes

Component	Textual Notation for an Integrator Node	Graphical Notation
Formal interface	`node IntegrFwd( U: real ; hidden TimeCycle: real)` `  returns ( Y: real) ;`	Arrows

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

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

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.

The SCADE Editor supports a user-friendly structured editing mode for graphical and textual nodes.

A node is fully modular:

- There is a clear distinction between its interface and its body.
- There can be no side-effects from one node to another one.
- The behavior of a node does not depend on its context.
- A node can be used safely in several places in the same model or in another one.

3.2.3 Block 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 block diagrams, such as the one illustrated in Figure 3.5 below.

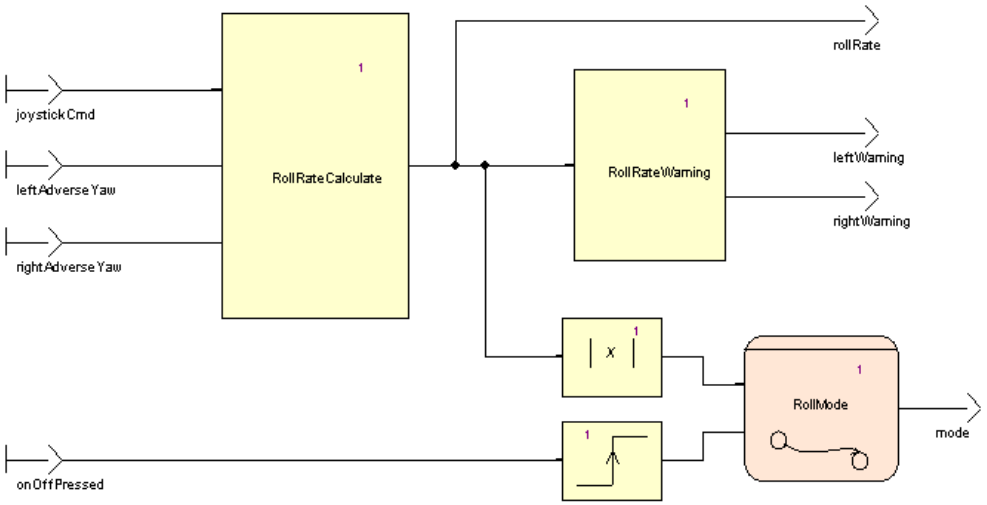


Figure 3.5: A SCADE block diagram for roll management

Boxes compute mathematical functions, filters, and delays, while arrows denote data flowing between the boxes. Blocks that have no functional dependency compute concurrently, and the blocks only communicate through the flows. Flows may carry numeric, Boolean, or discrete values tested in computational blocks or acting on flow switches.

SCADE blocks are fully hierarchical: blocks at a description level can themselves be composed of smaller blocks interconnected by local flows.

In Figure 3.5 above, the RollCalculate block is hierarchical, and one can zoom into it using the SCADE Editor. Hierarchy makes it possible to break design complexity by a divide-and-conquer approach and to design reusable library blocks.

SCADE is modular: the behavior of a node does not vary from one context to another.

The SCADE language is strongly typed, in the sense that each data flow has a type (Boolean, integer, real, arrays, etc.), and that type consistency in SCADE models is verified by the SCADE tools.

SCADE 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 3.6 below, where the "throttle" output depends on itself via the ComputeTargetSpeed and ComputeThrottle nodes. The SCADE semantic Checker detects this error and signals that this output has a recursive definition.

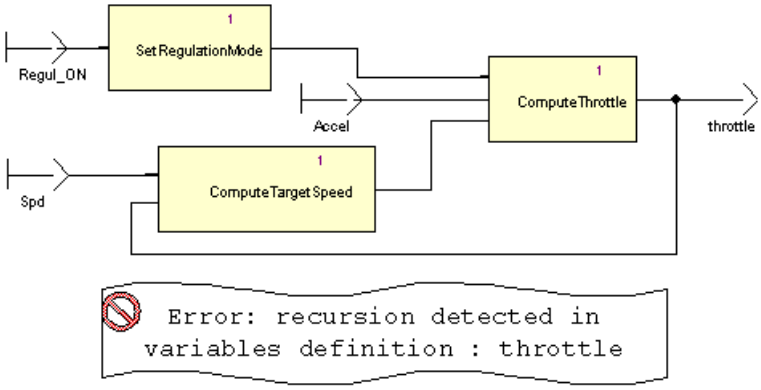


Figure 3.6: Detection of a causality problem

Inserting an FBY (delay) operator in the feedback loop solves the causality problem, since the input of the ComputeTargetSpeed block is now the value of "throttle" from the previous cycle, as shown in Figure 3.7.

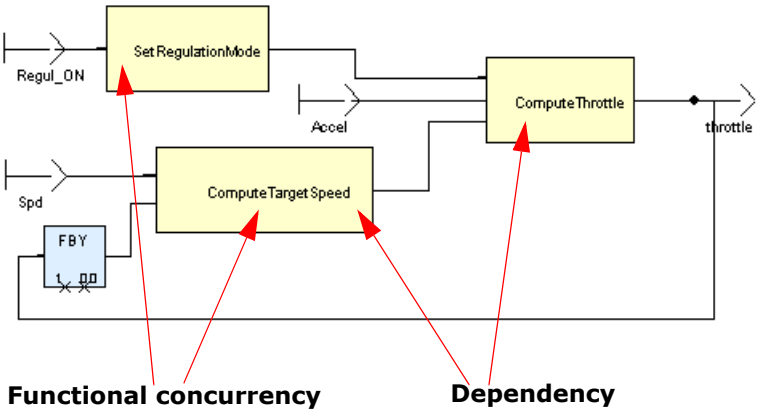


Figure 3.7: Functional expression of concurrency in SCADE

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

- Blocks SetRegulationMode and ComputeTargetSpeed are functionally parallel; since they are independent, the relative computation order of these blocks does not matter (because, in SCADE, there are no side-effects).
- ComputeThrottle functionally depends on an output of ComputeTargetSpeed. The SCADE 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 the 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 (the -> operator), as illustrated in Figure 3.8, which models a counter.

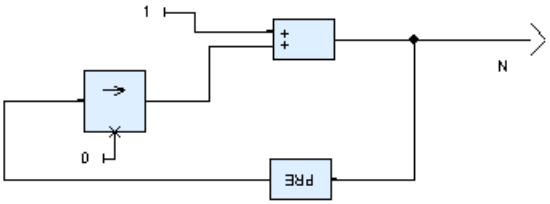


Figure 3.8: Initialization of flows

The second argument of the + operator is 0 in step 1 (the initial value), and the previous value of flow N in steps 2, 3, ... In the absence of explicit initialization, SCADE emits warnings. Mastering initial values is indeed a critical subject for critical embedded software.

3.2.4 Safe 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 the behavior varies qualitatively as a response to events. This is characteristic of modal human-machine interface, alarm handling, complex functioning mode handling, or communication protocols.

State machines have been very extensively studied in the past fifty years, and their theory is well understood. 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 has been introduced. SCADE hierarchical state machines are called Safe State Machines (SSMs).

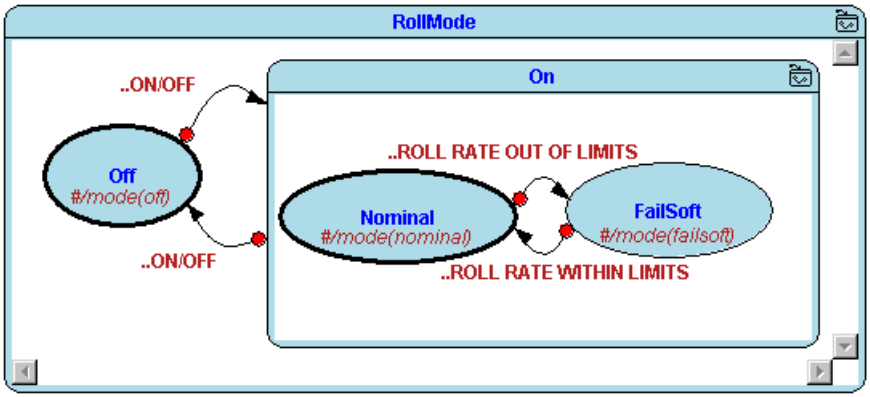


Figure 3.9: Safe State Machine for RollMode

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 methodological guidelines.

3.2.5 Mixed continuous/discrete control

Large applications contain cooperating continuous and discrete control parts. SCADE makes it possible to seamlessly couple both data flow and state machine styles. Most often, one includes SSMs into block-diagram design to compute and propagate functioning modes. Then, the discrete signals to which an SSM reacts and sends back, are simply transformed back and forth into Boolean data flows in the block diagram. The computation models are fully compatible. As an example, Figure 3.5 shows a data flow diagram where the RollMode block contains the Safe State Machine described in Figure 3.9.

3.2.6 Cycle-based intuitive computation model

The cycle-based execution model of SCADE is a direct computer implementation of the ubiquitous sampling-actuating model of control engineering. It consists of performing a continuous loop of the form illustrated in Figure 3.10 below. In this loop, there is a strict alternation between environment actions and program actions. Once the input sensors are read, the cyclic function starts computing the cycle outputs. During that time, the cyclic function is *blind to environment changes*.[2] When the outputs are ready, or at a given time determined by a clock, the output values are fed back to the environment, and the program waits for the start of the next cycle.

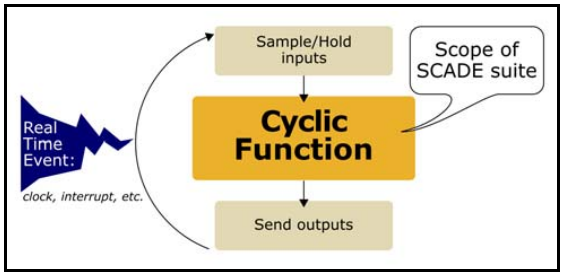


Figure 3.10: The cycle-based execution model of SCADE Suite

In a SCADE block diagram specification, each block has a so-called clock (the event triggering its cycles) and all blocks act concurrently. Blocks can all have the same clock, or they can have different cycles, which subdivide a master cycle. At each of its cycle, a block reads its inputs and generates its outputs. If an output of block A is connected to an input of block 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 SSMs have the very same notion of a cycle. For a simple state machine, a cycle consists of performing the adequate transition from the current state and outputting the transition output in the cycle, if any. Concurrent state machines communicate with each other, receiving the signals sent by other machines and

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

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

This cycle-based computation model carefully distinguishes between logical concurrency and physical concurrency. The application is described in terms of logically concurrent activities, block diagrams, or SSMs. 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.

3.2.7 SCADE data typing

The SCADE language is strongly typed.

The following data types are supported;

- Predefined types: Boolean, Integer, Real, 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 semantic Checker.

3.2.8 SCADE Suite as a model-based development environment

SCADE Suite is an environment for the development of safety-related avionics software. It supports a model-based development paradigm, as illustrated in Figure 3.11:

- The model is the software requirements: it is the unique reference in the project and it is based on a formal notation.
- Documentation is automatically and directly generated from the model: it is correct and up-to-date by construction.
- The model can be exercised by simulation using the same code as the embedded code.
- Formal proof techniques can be directly applied to the model to detect corner bugs or to prove safety properties.
- Code is automatically and directly generated from the model with the KCG 4.2 qualified automatic code generator: the code is correct and up-to-date by construction.

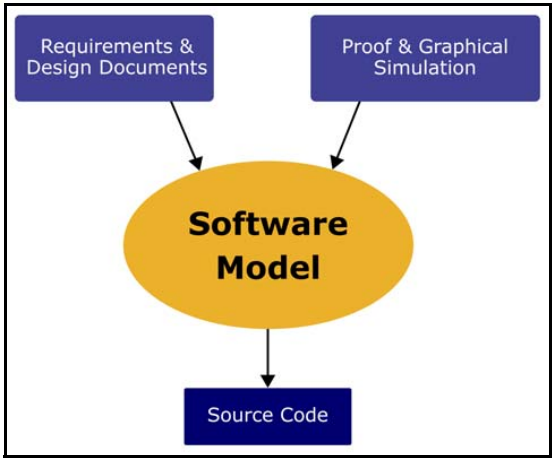


Figure 3.11: Model-based development with SCADE Suite and KCG 4.2

SCADE Suite applies the following "golden rules":

- Share unique, 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: Detect errors in early stages and/or write "correct-by-construction" descriptions.

SCADE Suite enables the saving of a significant amount of verification effort, essentially because it supports a "correct-by-construction" process.

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

BENEFITS OF THE "DO THINGS ONCE" PRINCIPLE

The SCADE model formalizes a significant part of the software architecture and system design. It 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, will share the SCADE model as a reference.

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

3.2.9 SCADE modeling and safety benefits

In conclusion to 3.2, we have shown that SCADE strongly supports safety at model level because of the following points:

- The SCADE modeling language was rigorously defined. Its interpretation does not depend on the reader or on a tool. It relies on more than twenty years of academic research. The semantic kernel of SCADE is very stable: it has not changed over the last 15 years.
- The SCADE modeling language is simple. It relies on very few basic concepts and simple combination rules of these concepts. There are no complex control structures like loops or gotos. There is no creation of memory at runtime. There

is no way to incorrectly access memory through pointers or an index out of bounds in an array.

- The SCADE modeling language contains specific features oriented towards safety: strong typing, mandatory initialization of flows, and so on.
- A SCADE model is deterministic. A system is deterministic if it always reacts in the same way to the same inputs occurring with the same timing. In contrast, a non deterministic system can react in different ways to the same inputs, the actual reaction depending on internal choices or computation timings. It is clear that determinism is a must for an aircraft: internal computation timings should not interfere with the flight control algorithms.
- The SCADE modeling language provides a simple and clean expression of concurrency at functional level (SCADE block diagrams are computed concurrently; data flows express dependencies between blocks). This avoids the traditional problems of deadlocks and race conditions.
- The Editor and Code Generators of SCADE Suite perform the complete verification of the language rules, such as type and clock consistency, or causality in SCADE models.

4. Software Development Activities with SCADE Suite

In this section we provide a more detailed view on 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 development flow and more traditional textual/manual flows. It has been observed on industrial projects that the fraction developed with SCADE typically ranges from 60% to 90% of the applicative part of the software.

Figure 4.1 shows the DO-178B development processes with those where SCADE is used highlighted in bold frame. Traceability between system requirements and software high-level and low-level requirements can be managed with a tool such as DOORS, thanks to the SCADE-DOORS link.

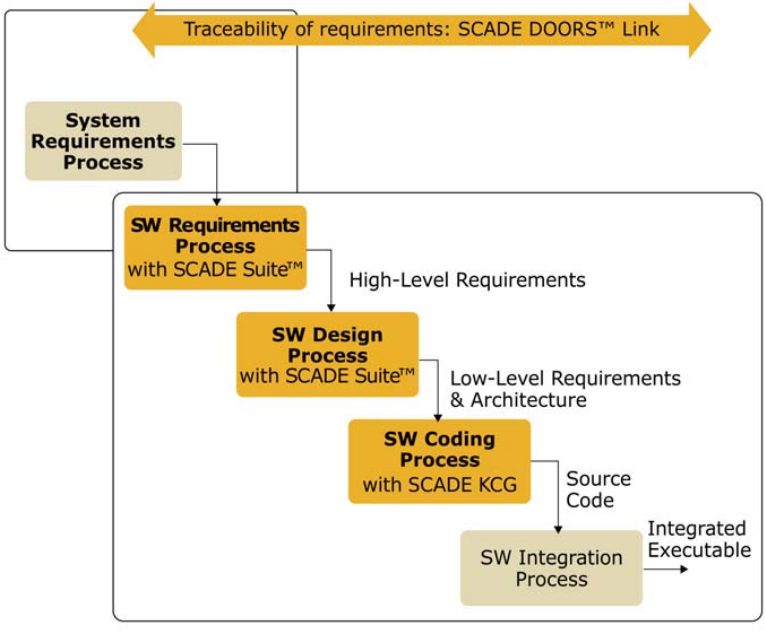


Figure 4.1: Software development processes with SCADE Suite

Some companies start using SCADE to define control laws during the system definition. 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 modeling is used extensively in the software design process to develop major parts of the low-level requirements and the architecture. From such SCADE models, KCG, the qualified SCADE Code Generator, can automatically generate C source code.

As shown in Figure 4.2, the HLRs that are described in SCADE are also LLRs. Many other HLRs, which are textual (light color fill), are refined in SCADE form in the design phase.

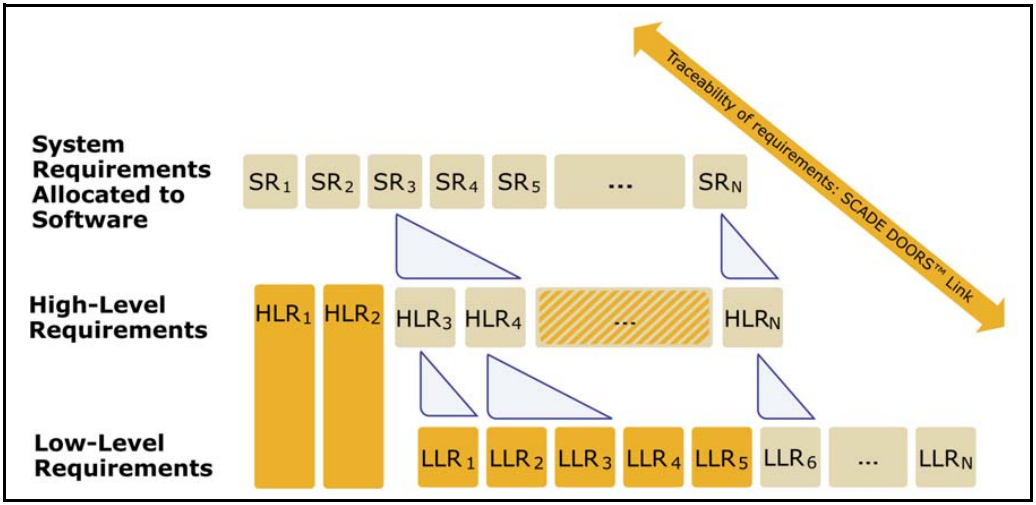


Figure 4.2: Development of high-level and low-level requirements with SCADE

4.2 Software Requirements Process with SCADE

In DO-178B terminology, the software requirements process produces the high-level requirements (HLR). Most of these high-level requirements are in textual form, illustrated with figures.

A partial SCADE model (see the example in Figure 4.3) 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 syntactic and semantic Checker.

- Prepare the framework for the design process. Having defined the top-level functions and their interfaces will compel the refinements to be consistent in terms of interfaces.

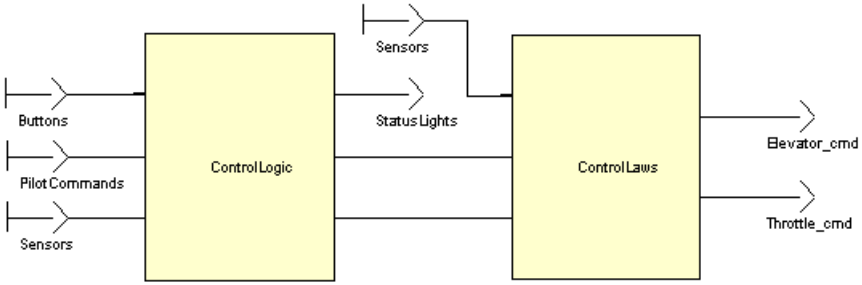


Figure 4.3: Top-level view of a simple flight control system

SCADE's flexible annotation feature allows attaching comments or more specific information (such as physical units) to the SCADE nodes, interfaces, and data types.

The document items generated from the SCADE model can be inserted or handled as an annex to the HLR document.

4.3 Software Design Process with SCADE

In DO-178B terminology, the software design process produces the architecture and the low-level requirements.

Figure 4.4 illustrates the design flow with SCADE that is detailed in the next sections.

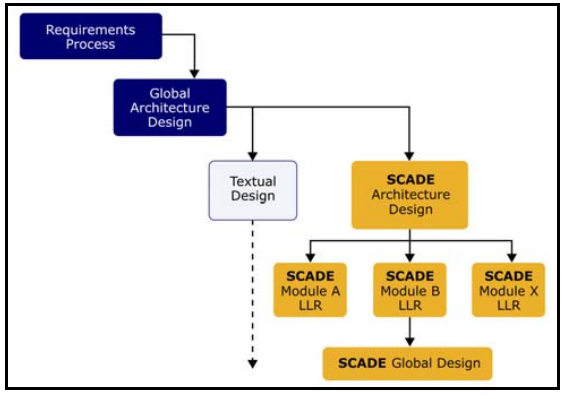


Figure 4.4: The software design process with SCADE

4.3.1 Architecture design

GLOBAL ARCHITECTURE DESIGN

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

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

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

SCADE is well suited to the functional parts of the software, such as logic, filtering, regulation. It is usually less well suited 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 architecture is composed of data type definitions, top-level nodes, and their connections, which ensure:

- Stability and maintainability: The team needs a stable framework during the first development as well as when there are updates.
- Readability
- Efficiency

There is no magic recipe to achieving a good 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, such as adding a sensor, an error case, and evaluate the robustness of the architecture to such changes.
- Retain the architecture that is robust to changes and minimizes the complexity of interconnections.

For example, the architecture shown in Figure 4.3 groups several sensors in one structured flow; it is therefore more maintainable than if each individual sensor value has its own input and flow throughout the model.

Note: If SCADE has already been used for the high-level requirements, then this has to be considered as the first candidate for the SCADE architecture, since it has the best direct traceability to the HLRs. That said, it is recommended that this architecture also be verified to ensure it has the right properties for maintainability.

4.3.2 SCADE low-level requirements development

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

The following sections provide some examples of SCADE modeling patterns. The “SCADE Design Guidelines” [SCADE_DGL] handbook provides more detailed and complete coverage of design guidelines.

INPUT/OUTPUT HANDLING

We assume that raw acquisition from physical devices and/or from data buses is done with drivers to feed SCADE 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 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 4.5

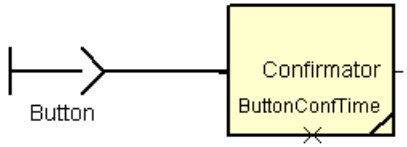


Figure 4.5: Inserting a Confirmator in a Boolean input flow

In a similar way, outputs to actuators have to be value-limited and rate-limited, which can be ensured by inserting Limiter blocks before the output, as shown in Figure 4.6 below.

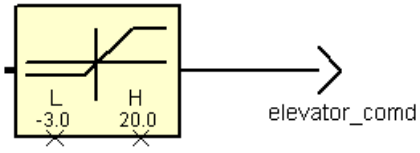


Figure 4.6: Inserting a Limiter in an output flow

Since the data flow is very explicit in a SCADE 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

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.

SCADE 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 $s=K1*u - K2* z^{-1}$, which means $s(k)=K1*u(k) - K2* s(k-1)$, this can be expressed in textual SCADE as s=K1*u-K2*pre(s) or graphically, as shown in Figure 4.7 below.

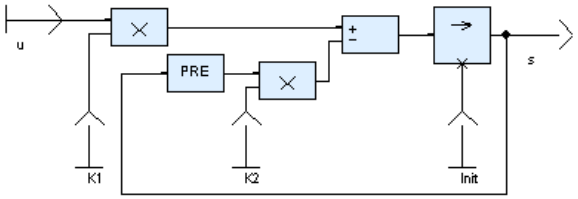


Figure 4.7: A first order filter

SCADE can implement both Infinite Impulse Response (IIR) filters and Finite Impulse Response (FIR) filters. In an 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 a filtering or regulation algorithm with SCADE:

a Develop this algorithm directly in graphical or textual SCADE.

b Develop it by reusing library blocks such as “first order filter,” “integrator,” etc. These library blocks are themselves developed with SCADE.

Using library blocks 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 comprise an Integrator.
- It enforces consistency throughout the project.
- It factors the code.

DECISION LOGIC

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

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

SCADE 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 Confirmator 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 shown in Figure 4.8.
- Conditional activation of nodes depending on Boolean conditions.
- Safe State Machines (SSM), as shown in Figure 4.9.

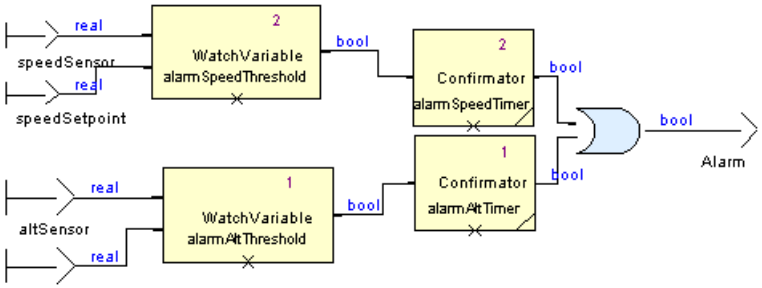


Figure 4.8: Alarm detection logic

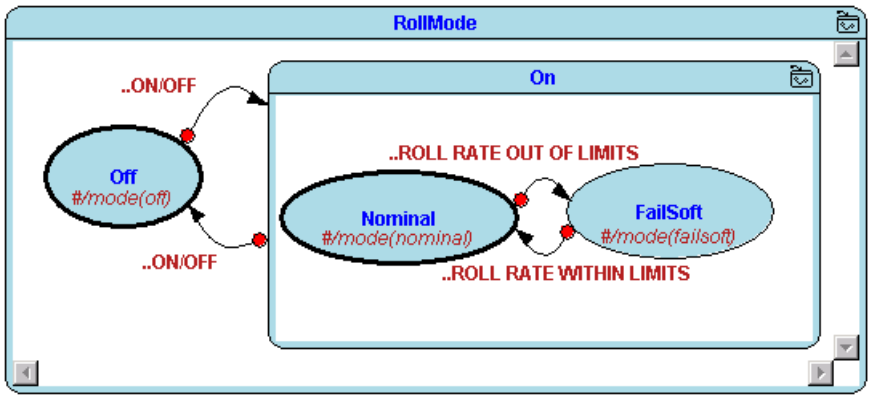


Figure 4.9: Safe State Machine for RollMode management

Which technique to use for decision logic?

When starting with SCADE, 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 combinatorial logic: simply use a logical expression in the data flow. A state machine that just jumps to state Xi when condition Ci is true, independently of the current state, is a degenerated one and does not deserve to be state machine.
- Does the state have a strong qualitative influence on the behavior? This is in favor of a state machine.
- Is there a hierarchy in the states? This is in favor of SSM, at least in the requirements phase. However, with the current version of the SSM code generation chain, it is recommended to limit

the depth of the state hierarchy in order to ease verification of the generated code (see §4.4.2).

To express concurrency:

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

Last but not least, pack, use, or reuse behavior that you have captured into blocks inserted into higher-level data flow nodes. For instance, the design of the alarm manager in Figure 4.8 uses threshold detectors and confirmators.

ROBUSTNESS

Robustness issues must be addressed at each level. We recommend that robustness be addressed differently at the design and coding levels.

- Design Level

  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). There should be no exception mechanisms 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). And 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 blocks, such as guarded division, voters, confirmators, and limiters. Their presence in the diagrams is very explicit for the reader. It is also recommended to use 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

The SCADE Code Generator automatically generates the complete C code that implements the software system design and module design defined in SCADE for both data flows and state machines (see Figure 4.10). It is not just a generation of skeletons: the complete dynamic behavior is implemented.

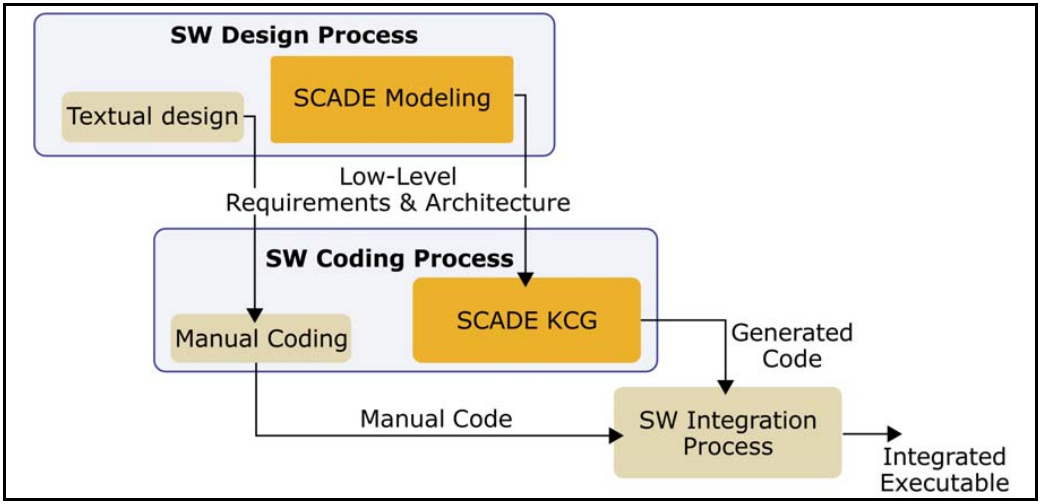


Figure 4.10: The software coding process with SCADE

Let us now distinguish two cases of using automatic code generation: data flows and SSMs.

4.4.1 Code generation from SCADE data flow diagrams

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

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

- The code is portable: it is ISO-C compliant and it performs no operating system call.
- The code structure reflects the model architecture (by function or by blocks, depending on code generation options).
- The code is readable and traceable to the model through the use of names and annotations.
- Memory allocation is fully static (no dynamic memory allocation).
- There is no recursion and no looping (except a few local, fixed-size loops).
- Execution time is bounded.
- The code is decomposed into elementary assignments to local variables (this restricts use of the optimization options of the C compiler + SCADE KCG option).
- Every C variable is assigned only once. This is called “Single Static Assignment (SSA)” and it reduces compilation complexity.
- Expressions are explicit-parenthesized.

- No dynamic address calculation is performed (no pointer arithmetic).
- It contains no array indexing (since there are no arrays, except for FBY).
- 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 the SCADE data flow model is illustrated in Figure 4.11 below.

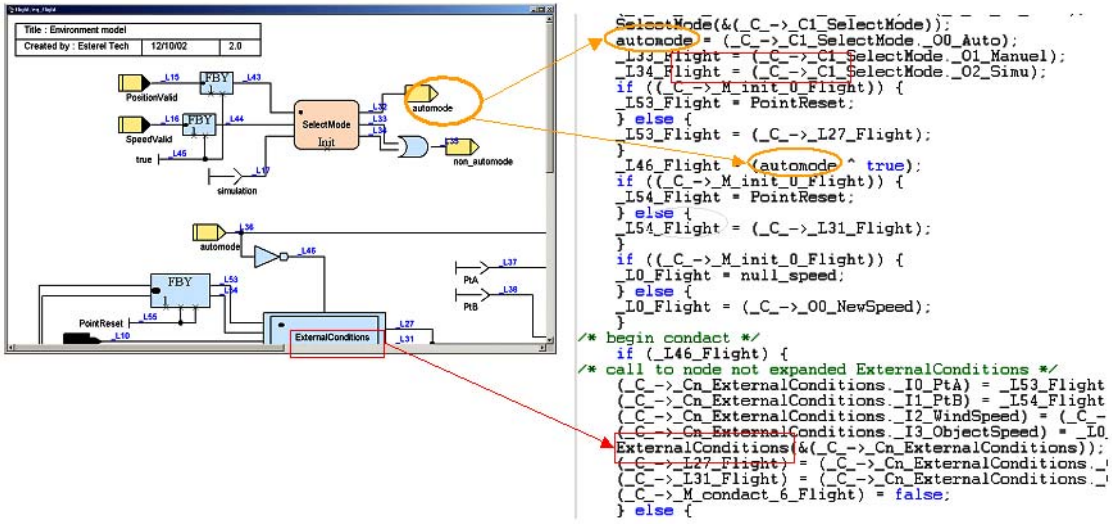


Figure 4.11: SCADE data flow to generated C source code traceability

Various code generation options can be used to tune the generated C code to particular target and project constraints. Basically, there are two ways to generate code from a SCADE node:

- 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 4.12.

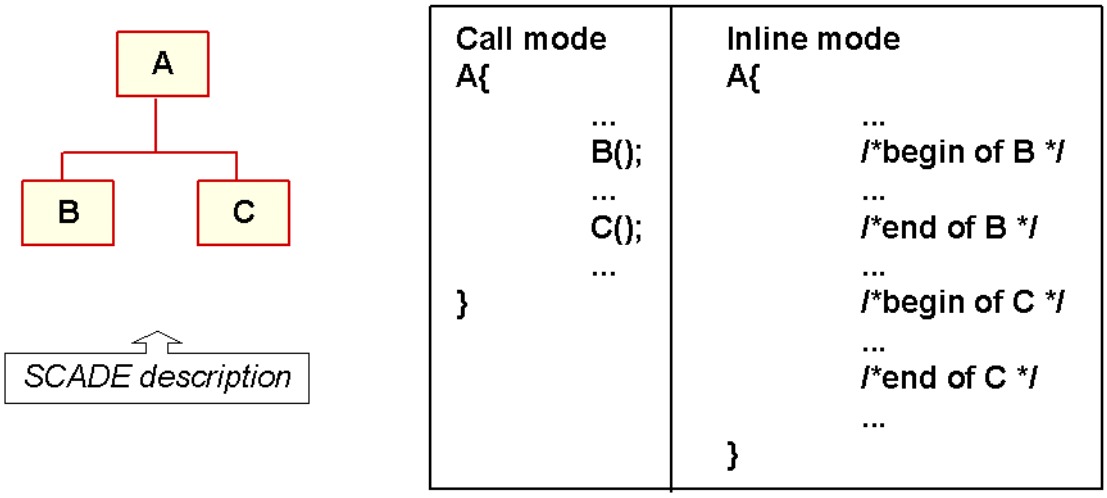


Figure 4.12: Comparing Call and Inline modes

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

CONTROL FLOW

Traditional design and programming is error-prone for the control flow. There are frequent errors related to:

- Loop termination
- Computation order
- Deadlocks
- Race conditions (a result depends on computation timing of parallel tasks/threads)

With SCADE, the approach is different:

- The designers develop the SCADE model focusing on functions; they need not spend time analyzing the dependencies and developing the sequencing.
- When the KCG analyzes a SCADE model, it generates a computation order based on the functional dependencies.
- Every data element is computed at the right time, once and only once.
- There are no loops and no “goto” constructs.
- Concurrency is expressed functionally, but the generated code is sequential and contains no tasking overhead.

4.4.2 Code generation from SCADE SSMs

Figure 4.13 describes the automatic C code generation chain from SSM designs. The SSM node is first translated into a SCADE data flow node, then the KCG 4.2 Code Generator is used.

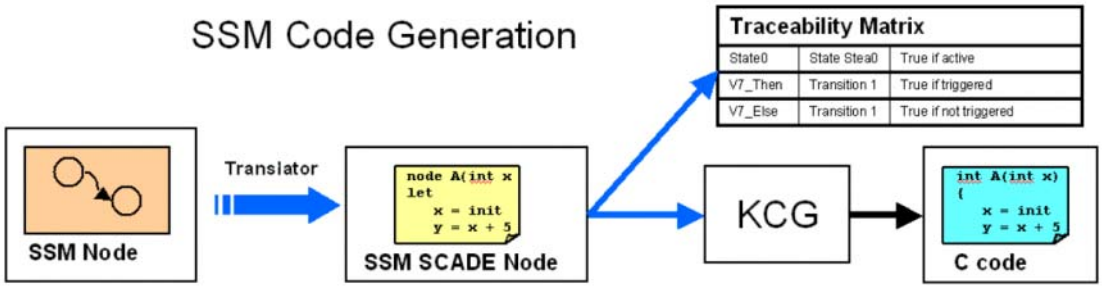


Figure 4.13: C code generation from SSMs

The code has the same properties as code generated from data flow diagrams.

Traceability of the code to the SCADE SSM model is illustrated in Figure 4.14 below. It relies on the use of names, annotations, and a traceability matrix. Variable names identify the inputs, outputs, and state names. Specific comments identify the transition behavior.

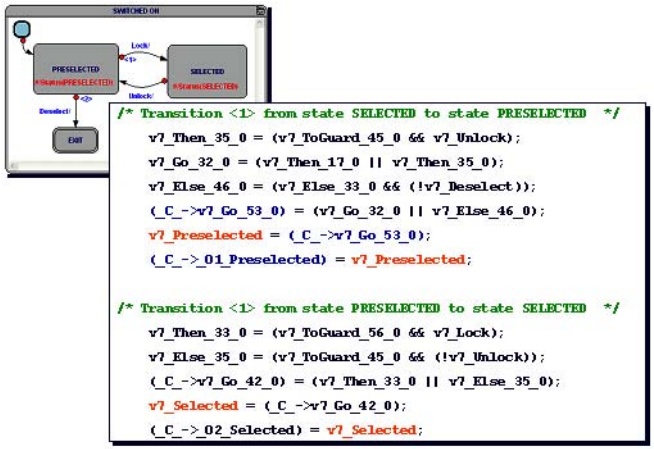


Figure 4.14: SCADE SSM to generated C source code traceability

The way to accommodate concurrent designs in a SCADE model, while keeping the necessary traceability between model and code, is to use data flows to handle concurrency. This is shown in Figure 4.15 below, where there are three buttons in parallel.

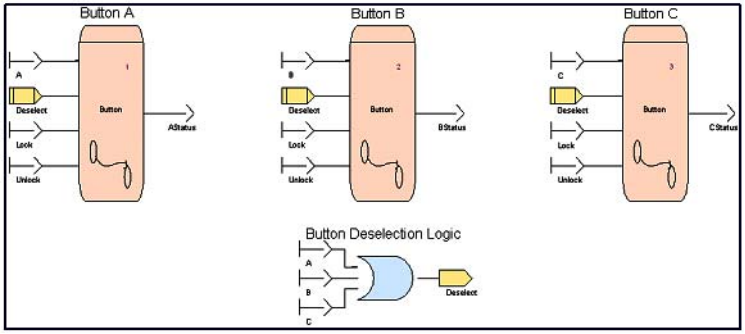


Figure 4.15: Three buttons in a parallel data flow

4.5 Software Integration Process

4.5.1 Integration aspects

Integration of SCADE code concerns:

- Scheduling
- Input/output
- Integration of external data types and constants
- Integration of external functions

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

4.5.3 Integration of external data and code

SCADE allows using external data types and functions. In the model, they have to be declared as “imported,” and for functions, their interface also has to be declared. Examples of such functions are trigonometric functions, byte encoding, and checksum. At integration time, these functions have to be compiled and linked to the SCADE-generated code. The SCADE

Simulator automatically compiles and links external code when the path names of the source files are given in the project settings.

4.5.4 SCADE scheduling and tasking

Scheduling actually has to be addressed from the preliminary design phase, but for the sake of simplicity we describe it here.

First, we recall the execution semantics of SCADE, and then we examine how to model and implement scheduling of a SCADE model in single or multirate mode, in single tasking or in multitasking mode.

SCADE EXECUTION SEMANTICS

SCADE execution semantics is based on the cycle-based execution model that we described in Section 3.2.6. This model can be represented with Figure 4.16.

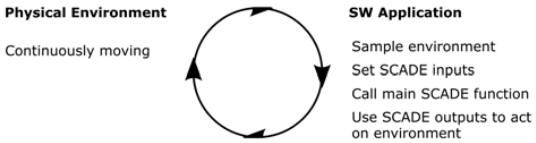


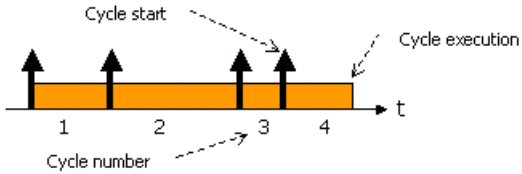
Figure 4.16: SCADE execution semantics

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

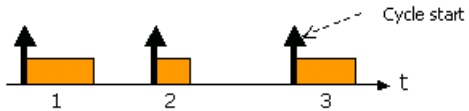
BARE SYSTEM IMPLEMENTATION

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

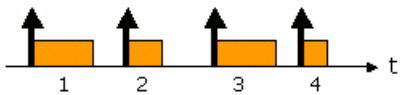
- Polling: a new cycle is started immediately after the end of the previous one in an infinite loop.



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

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

SINGLE-TASK INTEGRATION OF SCADE FUNCTION WITH AN RTOS

A SCADE design can easily be integrated in an RTOS in the same way that it is integrated in a general-purpose C code, as shown in Figure 4.17. The infinite loop construction 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.

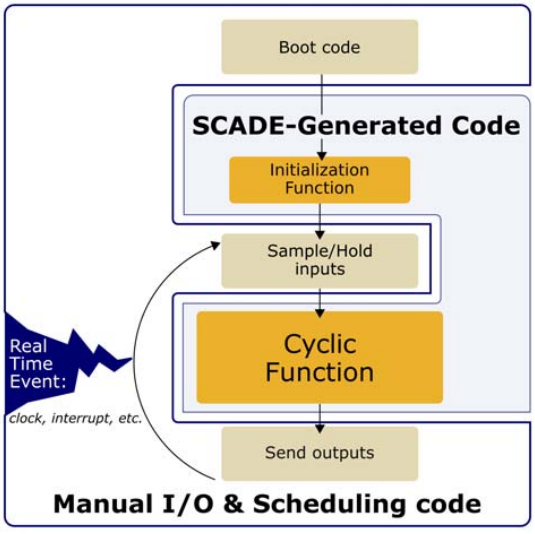


Figure 4.17: SCADE code integration

This architecture can be designed by hand for any RTOS.

Note that concurrency is expressed functionally in the model and that the Code Generator takes into account the data flow 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 it does not interfere with the SCADE code.

MULTIRATE, SINGLE-TASK APPLICATIONS

SCADE 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 a *condact*[3] operator can do this. Slowest rates will be derived from the fastest rate, which is always the top-level rate. This ensures a deterministic behavior.

The following application has two rates: Sys1, which is as fast as the top-level, and Sys2, which is four times slower, as shown in Figure 4.18 below.

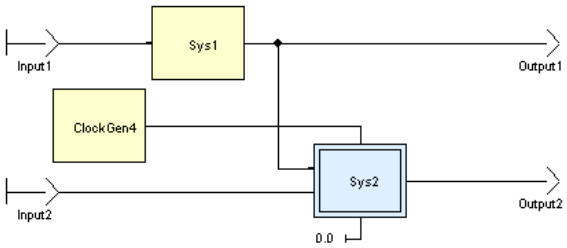


Figure 4.18: Modeling a birate system

3. A condact operator has an input clock (on top) that is 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. When an operator is put under condact, its execution only occurs when a given condition is true.

The schedule of this application will be as shown in Figure 4.19 below:

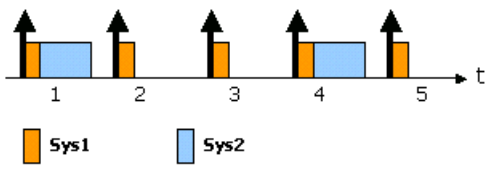


Figure 4.19: Timing diagram of a bi-rate system

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

The solution consists of splitting the slow part into several little slow parts and distributing their execution on several fast rates. This is a safe 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 4.20:

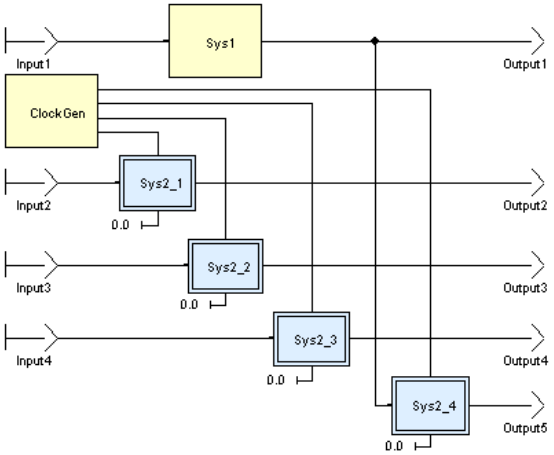


Figure 4.20: Modeling distribution of the slow system over four cycles

The slow part, Sys2, is split into four subsystems. These subsystems are executed sequentially, one after the other, in four cycles, as shown in Figure 4.21 below:

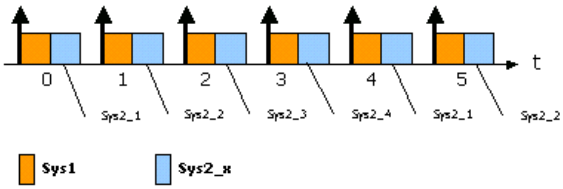


Figure 4.21: Timing diagram of the distributed computations

Note that Sys1 execution time can be longer than with the previous design. This means that a slower, but cheaper, 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 construction 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, respecting data flow execution order.
  - SCADE simulation and proof are valid for the generated code.
  - Same code interface as a mono-rate application.
- Constraints:
  - Need to know 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 has been used for fairly large industrial systems. There are situations where implementation of the SCADE 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 [Caspi-2004] for details in Appendix A).

4.6 Teamwork

To work 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 modules (called “nodes” in SCADE) and there are no side-effects from one node to another.

A typical project organization is shown in Figure 4.22:

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

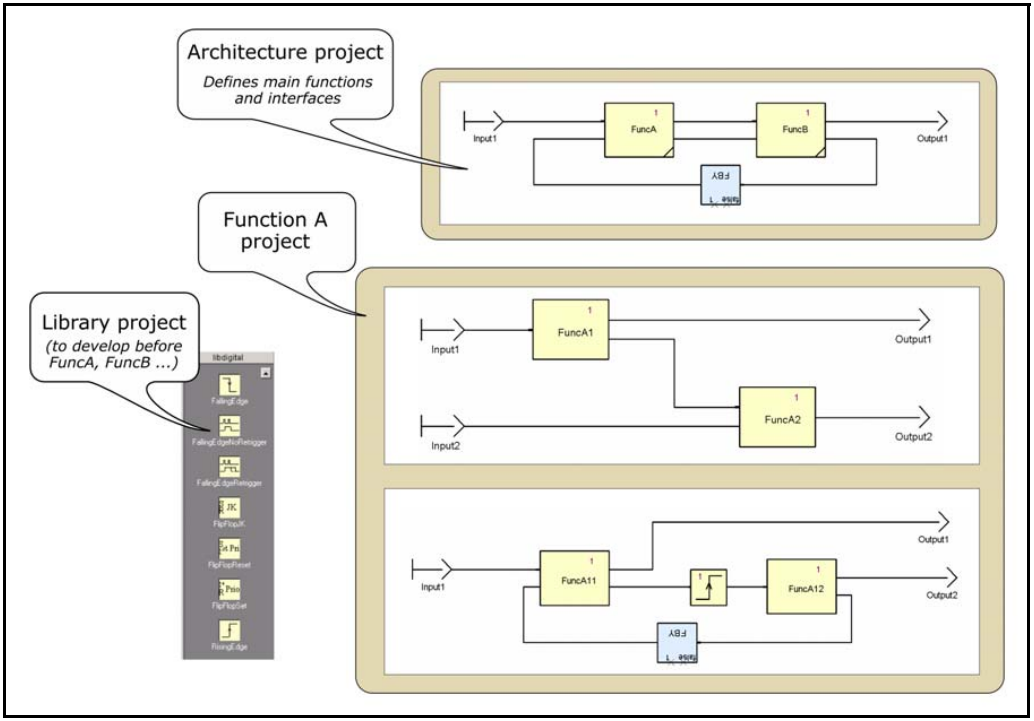


Figure 4.22: Typical teamwork organization

At each step, the team can verify in a mouse click that the subsystem remains consistent with the interface. Later, the integration of those parts in a larger model can be achieved by linking these “projects” to the larger one. At any stage, the SCADE semantic Checker verifies the consistency of this integration in a mouse click.

All these data have to be kept under strict version and configuration management control. SCADE can be integrated with the customer’s configuration management system via SCCI™ (Microsoft 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 libraries can store definitions of nodes and/or data types, which can be reused in several places. These range from basic nodes such as latches or integrators to complex, customer-specific systems.

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 typically satisfied through a combination of review, 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, when the implementation does not conform to the requirements.

5.2 Verification of the SCADE High-Level Requirements

5.2.1 Verification objectives for the HLR

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

Table 5.1: DO-178B Table A-3

	Objective
1	Software high-level requirements comply with system requirements.
2	Software high-level requirements are accurate and consistent.
3	Software high-level requirements are compatible with target computer.
4	Software high-level requirements are verifiable.
5	Software high-level requirements conform to standards.
6	Software high-levels requirements are traceable to system requirements.
7	Algorithms are accurate.

For those elements of the SCADE model that are developed during the requirements phase, they have to be verified against the objectives of DO-178B Table A-3.

They also have to 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

This compliance is verified by peer review. At this stage, the SCADE model is usually incomplete and composed primarily of top-level nodes. The meaning of these nodes is described textually, either in the body of the textual document, or as textual annotations of the SCADE model.

ACCURACY AND CONSISTENCY

Again, since the model at this stage is incomplete, verification is mostly based on review. Some consistency checks of both the interface and the connections are automated by the SCADE Checker.

COMPATIBILITY WITH TARGET COMPUTER

There is nothing specific to SCADE at this stage.

VERIFIABILITY

The SCADE model identifies the top-level functions and describes the functional breakdown and data flows between top-level functions. This description is verifiable.

COMPLIANCE WITH STANDARDS

The SCADE notation has precise syntactic and semantic rules (*e.g.*, data type consistency) defined in the SCADE language reference manual. Compliance with this standard can be verified by the SCADE syntactic and semantic checkers. Note that a model that has been created with the SCADE Editor is syntactically correct automatically.

TRACEABILITY TO SYSTEM REQUIREMENTS

Traceability can be managed with the SCADE DOORS Gateway or by using annotations of the SCADE model elements to reference the system requirements (see §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.

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

	Objective	Verification Method
1	Software high-level requirements comply with system requirements.	Review
2	Software high-level requirements are accurate and consistent.	Review
3	Software high-level requirements are compatible with target computer.	Review
4	Software high-level requirements are verifiable.	Review
5	Software high-level requirements conform to standards.	SCADE syntax Checker
6	Software high-level requirements are traceable to system requirements.	Establish traceability with DOORS Link or with annotations
7	Algorithms are accurate.	N/A (No SCADE algorithm at this stage)

5.3 Verification of the SCADE Low-Level Requirements and Architecture

5.3.1 Verification objectives

The complete SCADE model has 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 the SCADE model is 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).

Table 5.3: DO-178B Table A-4

	Objective
1	Low-level requirements comply with high-level requirements.
2	Low-level requirements are accurate and consistent.
3	Low-level requirements are compatible with target computer.
4	Low-level requirements are verifiable.
5	Low-level requirements conform to standards.
6	Low-levels requirements are traceable to high-level requirements.
7	Algorithms are accurate.
8	Software architecture is compatible with high-level requirements.

Table 5.3: DO-178B Table A-4 (Continued)

	Objective
9	Software architecture is consistent.
10	Software architecture is compatible with target computer.
11	Software architecture is verifiable.
12	Software architecture conforms to standards.
13	Software partitioning integrity is confirmed.

5.3.2 SCADE model accuracy and consistency

The syntactic and semantic Checkers of SCADE Suite 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 the SCADE Suite Editor.

5.3.3 Compliance with design standard

The SCADE Language Reference Manual defines the design standard for the SCADE architecture and LLRs: it defines precisely the syntactic and semantic rules that a SCADE model has to follow. The SCADE syntactic and semantic Checkers included in KCG verify compliance with this standard.

5.3.4 Traceability from SCADE LLR to HLR

Traceability from SCADE LLRs to the HLRs can be efficiently supported with a tool such as DOORS™ Link. This tool imports the SCADE structure into DOORS. Then, in the DOORS environment, this structure can be handled like any other requirements hierarchy. It can then be browsed; links with the HLR can be established and analyzed; coverage matrices can be generated. With a mouse click on a SCADE element in DOORS, this element is automatically selected in SCADE Editor, as shown in Figure 5.1 below.

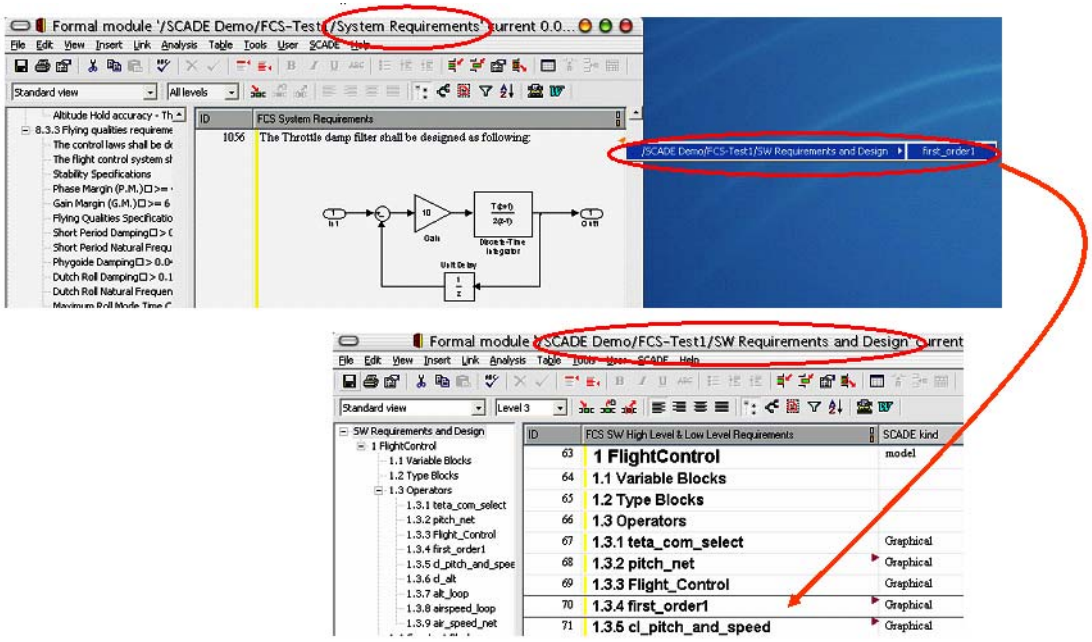


Figure 5.1: Traceability between SCADE LLR and HLR using DOORS™ Link

5.3.5 Verifiability

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

5.3.6 Compliance with high-level requirements

To verify compliance of the SCADE model with the HLR, there are three complementary techniques:

- Peer review
- 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 data of the SCADE model can be automatically generated. SCADE notation has several advantages compared to textual notation:

- The description is not subject to interpretation. This is because the SCADE notation has formal definition.
- The description is complete. Incompleteness is detected by the SCADE semantic Checker.
- Its graphical form is simple and intuitive.

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

SCADE SIMULATION

It is helpful to dynamically exercise the behavior of a SCADE specification to better verify how it functions. As soon as a SCADE model (or pieces of it) is available, it can be simulated with the SCADE Suite Simulator, as shown in Figure 5.2. 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. Note that all simulation scenarios, like all testing activities, have to be based on the software high-level requirements.

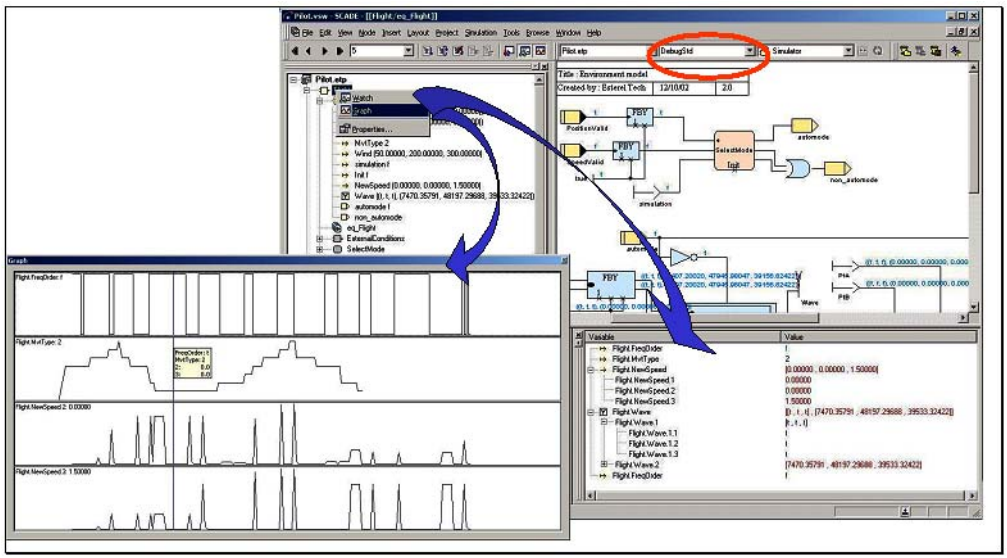


Figure 5.2: Simulation makes it possible to "play with the software specification"

Simulation supports the detection of assertion violation, which is a powerful help during the verification of robustness.

SCADE SUITE FORMAL VERIFICATION WITH DESIGN VERIFIER

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

Testing activities, including SCADE simulation, let you test and verify the correctness of the design. However, with testing, one is never 100% sure that the design is correct, as one usually never tests all possible scenarios.

Formal verification of computer systems is a set of activities consisting of 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 LLR in 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. Analyze mathematically and/or by state space exploration the validity of the safety properties.

Let us take a simple example. Assume we have a landing gear control system, which may trigger a landing gear retraction command. Assume that we want 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"
```

We would express in a SCADE node the safety property shown in Figure 5.3 below, reflecting the property above. This node is called an observer.

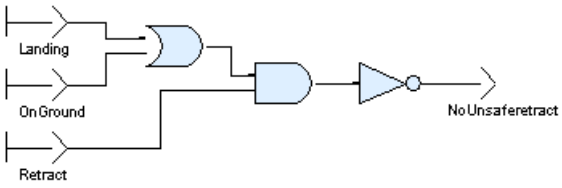


Figure 5.3: Observer node containing landing gear safety property

Then, we would connect the observer node to the controller in a verification context node, as shown in Figure 5.4 below.

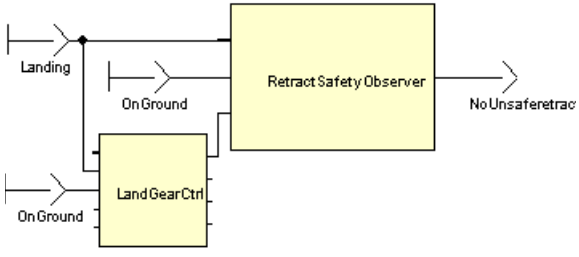


Figure 5.4: Connecting the observer node to the landing gear controller

Traditionnally, 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, which does not require a lot of time to use and also enables increased confidence in the system. To meet this challenge, the SCADE Suite Design Verifier offers to a wide range of users a solution for easy access to formal verification, which can rely 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 you want your design to fulfill.
- Property Verification: This is a push-button feature of the SCADE application, which basically provides a yes/no answer. Moreover, in the case of a no answer, SCADE lets you discover in an automatic and user-friendly way why a no answer was reached.

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 have to 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 5.5 represents the Design Verifier workflow. It consists of successive tasks that may be iterated. There are three kinds of tasks:

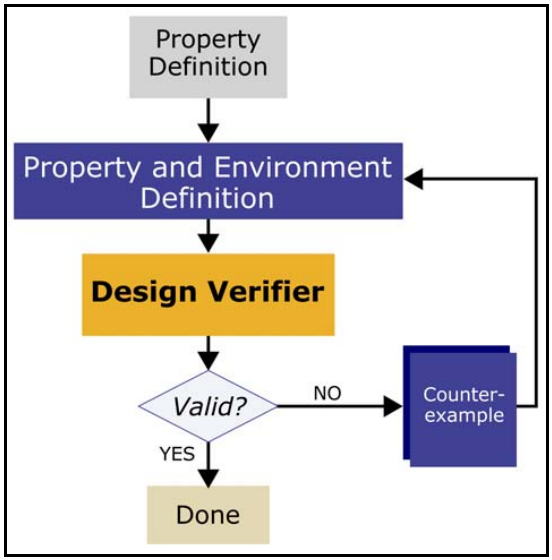


Figure 5.5: Design Verifier workflow

- Property Definition: This task consists of extracting properties from the high-level requirements to check with Design Verifier.
- Property and Environment Specification: This task consists of formally describing, as SCADE observer properties, the requirement extracted as properties in SCADE. 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 in the communication between the Safety Assessment Process and the System Development Process.

Typically, safety properties can be directly expressed from the FHA (Functional Hazard Assessment) and from the PSSA (Preliminary System Safety Assessment) phases, as defined by ARP 4754. Then, by verifying that the software model respects these properties, this can feed the SSA (System Safety Assessment) process.

The SCADE 5.1 Design Verifier will also support automatic detection of potential division by zero and overflow/underflow throughout the model.

5.3.7 Partitioning

SCADE introduces no specific risks, but provides no partition mechanism. Partitioning is beyond the scope of SCADE. It has to be ensured by low-layer hardware and software mechanisms such as memory partitioning and interrupt service routines.

5.3.8 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

	Objective	Verification Method
1	Low-level requirements comply with high-level requirements.	Review, simulation, formal verification
2	Low-level requirements are accurate and consistent.	Ensured by notation and semantic checks
3	Low-level requirements are compatible with target computer.	SCADE computational model uses no target-specific resource. Remains to be verified: memory and CPU consumption.
4	Low-level requirements are verifiable.	Ensured by formality of SCADE notation
5	Low-level requirements conform to standards.	SCADE syntax and semantic Checker
6	Low-levels requirements are traceable to high-level requirements.	DOORS Link or annotations
7	Algorithms are accurate.	SCADE formal description is unambiguous. Numerically sensitive algorithms have to be analyzed by simulation and numerical analysis techniques.
8	Software architecture is compatible with high-level requirements.	Review
9	Software architecture is consistent.	SCADE semantic Checker
10	Software architecture is compatible with target computer.	SCADE computational model uses no target-specific resource. Remains to be verified: memory and CPU consumption.
11	Software architecture is verifiable.	Ensured by SCADE notation.
12	Software architecture conforms to standards.	SCADE syntax and semantic Checker.

Table 5.4: DO-178B Table A-4 Objectives Achievement (Continued)

	Objective	Verification Method
13	Software partitioning integrity is confirmed.	SCADE introduces no specific risk, but provides no partition mechanism; traditional method has to be used.

5.4 Verification of Coding Outputs and Integration Process

5.4.1 Verification objectives

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

Table 5.5: DO-178B Table A-5

	Objective
1	Source code complies with low-level requirements.
2	Source code complies with software architecture.
3	Source code is verifiable.
4	Source code conforms to standards.
5	Source code is traceable to low-level requirements.
6	Source code is accurate and consistent.
7	Output of software integration process is complete and correct.

5.4.2 Impact of code generator qualification

The KCG can be qualified as a development tool because it has been developed by Esterel Technologies to fulfill the DO-178B requirements for Level A development tools (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 Generator.

SOURCE CODE COMPLIES WITH SOFTWARE ARCHITECTURE

This is ensured by the qualification of the Code Generator.

SOURCE CODE IS VERIFIABLE

By specification of the Code Generator, the generated code reflects the model and is verifiable. The qualification of the Code Generator 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 constructs have to be implemented in C. The qualification of the Code Generator ensures that this standard is respected.

SOURCE CODE IS TRACEABLE TO LOW-LEVEL REQUIREMENTS

By specification, the generated code has a simple, readable structure, traceable to the model by names and by comments. The qualification of the Code Generator ensures that this is respected.

SOURCE CODE IS ACCURATE AND CONSISTENT

The specification of the Code Generator defines accurate and consistent code, reflecting accurate and consistent input models. The qualification of the Code Generator ensures that this is respected.

OUTPUT OF THE SOFTWARE INTEGRATION PROCESS IS COMPLETE AND CORRECT

This verification is achieved by the combined testing process (see Section 5.5).

5.4.3 Verification summary

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

Table 5.6: DO-178B Table A-5 Objectives Achievement

	Objective	Verification Method
1	Source code complies with low-level requirements.	Replaced by KCG qualification
2	Source code complies with software architecture.	Replaced by KCG qualification
3	Source code is verifiable.	Replaced by KCG qualification

Table 5.6: DO-178B Table A-5 Objectives Achievement (Continued)

	Objective	Verification Method
4	Source code conforms to standards.	Replaced by KCG qualification
5	Source code is traceable to low-level requirements.	Replaced by KCG qualification
6	Source code is accurate and consistent.	Replaced by KCG qualification
7	Output of software integration process is complete and correct.	See Combined Testing Process

5.5 The Combined Testing Process

5.5.1 Verification objectives

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

Table 5.7: DO-178B Table A-6

	Objective
1	Executable object code complies with high-level requirements.
2	Executable object code is robust with high-level requirements.
3	Executable object code complies with low-level requirements.
4	Executable object code is robust with low-level requirements.

Table 5.7: DO-178B Table A-6 (Continued)

	Objective
5	Executable object code is compatible with target computer.

5.5.2 Divide-and-conquer approach

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 KCG qualification and from the characteristics of the generated code:

1 Compliance of the SCADE model with the HLRs is, to a large extent, verified at the model level.

2 Compliance of the source code with the SCADE LLRs is ensured by KCG qualification.

3 For the verification of source to object code transformation, the characteristics of the generated source code allow a sample-based approach for low-level testing of the source code generated by KCG.

5.5.3 Combined testing process organization

This section describes the combined testing process. Appendix D provides details about the justification of the sample-based approach and about the process to build the sample.

Figure 5.6 summarizes the combined testing process.

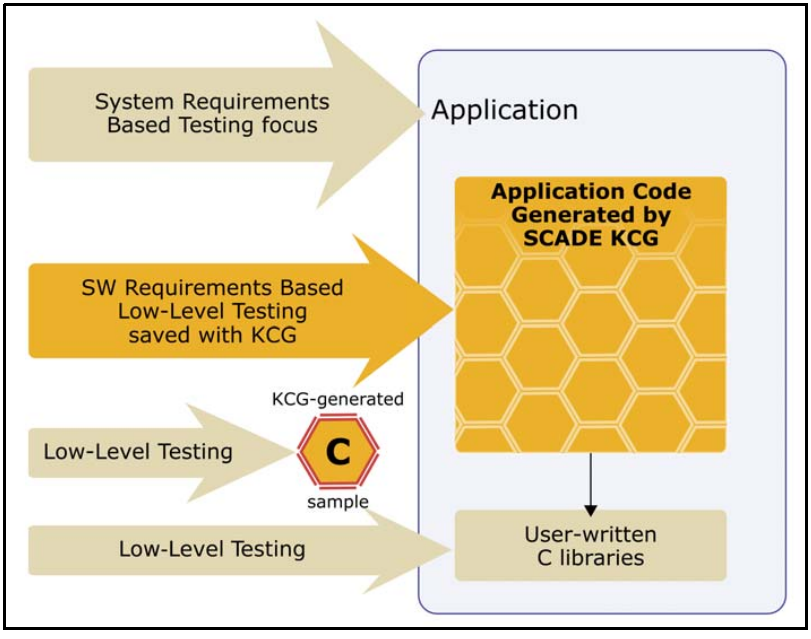


Figure 5.6: The combined testing process with KCG

The combined testing process is organized in the following way:

1 The coding 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 will be used for hand code and KCG code in the project.
- Analysis of this object code is performed according to CAST Paper P-12[CAST-12] to demonstrate that if any object is not directly traceable to source code, then this object code is correct (note that this activity is independent from SCADE).

2 For the functions that are hand-coded in the source code language, all verification activities are performed on the complete code:

- The user performs classical verification activities (source code review, all level testing, and structural coverage analysis at the source code level) for all the code.

3 For the source code automatically generated by 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.

4 For the whole application:

- The user performs extensive system requirements-based software and hardware/software integration testing. “Extensive” means that all system requirements allocated to software are covered by those tests. As explained in §5.5.5, the coverage of both HLRs and LLRs has to be achieved by the combination of target and host testing.

If the combination of all the above activities does not detect any errors in the object code, then we can have sufficient confidence that the compiler did not introduce undetected errors in the code generated by KCG for that application.

5.5.4 Verification summary

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

	Objective	Verification Method
1	Executable object code complies with high-level requirements.	Set of tests covering normal HLR conditions (representative SCADE simulation and target testing) + KCG qualification + compiler verification
2	Executable object code is robust with high-level requirements.	Set of tests covering abnormal HLR conditions (representative SCADE simulation and target testing) + KCG qualification + compiler verification
3	Executable object code complies with low-level requirements.	KCG qualification + compiler verification
4	Executable object code is robust with low-level requirements.	Test arithmetic exception handler and/or impose usage of robust arithmetic blocks
5	Executable object code is compatible with target computer.	Target resource usage analysis

6. Verification of the Verification Activities

6.1 Verification Objectives

We now have to assess how well the above-mentioned verification has been performed. Table 6.1 summarizes the verification objectives from DO-178B.

Table 6.1: DO-178B Table A-7

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

6.2 Verification of Test Procedures and Test Results

TEST PROCEDURE CORRECTNESS

The following categories of test procedures have to be reviewed:

- High-level requirements-based test procedures on the Simulator, if verification credit is sought. In this case, representativeness has to be demonstrated.
- Low-level test procedures of the C sample.
- System or high-level requirements-based test procedures on the target.

TEST RESULT CORRECTNESS AND DISCREPANCY EXPLANATION

The results of the above-mentioned test 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 done on the SCADE part are based on HLR(s) and will contain a link to the HLR(s) that they verify. The analysis of these

links will confirm full coverage of the HLR by the test cases; otherwise, test cases have to be complemented.

If representativity of the simulation is ensured, then coverage has to be ensured by the union of test cases performed on the target and on the Simulator. Otherwise, all HLRs have to be fully covered by target testing. Note that scenarios can be transferred from/to the Simulator to/from the target test environment.

6.4 LLR Coverage Analysis with MTC

6.4.1 Objectives and coverage criteria

This section addresses the coverage of the SCADE LLRs. Model coverage analysis is a means of assessing how far the behavior of a model has been explored. It is complementary to traceability analysis and high-level requirements coverage analysis.

Model coverage verifies that every element of the SCADE model (which represents a software design feature) has been dynamically activated when the high-level requirements 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. The 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 basically must reflect it. 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. In a word, 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 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 (SCADE form). It may involve some dynamic behavior.

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

Coverage criteria have been a research topic for some time, but most available results concern sequential programming, which is by far the dominant programming technique:

- Control flow coverage criteria are the most widely used. In that category, [DO-178B] has selected statement coverage, condition/decision coverage and MC/DC (Modified Condition/Decision Coverage) depending on the safety integrity level.
- Even most data flow coverage criteria found in the literature have been primarily designed for sequential programs. They focus on the definitions and uses of a variable: where is a variable defined/redefined; has it always been defined before use; which definition has been activated before use?

These existing criteria are a valuable source of inspiration for SCADE model coverage. However, in order to define relevant criteria for SCADE models, we need to take into account the characteristics of SCADE:

- A SCADE model describes the functionality of software, while a C program describes its implementation. As it will be 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 flow, while most programming languages and their criteria are sequential.
- Every SCADE node 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 definitions/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 tools verify such rules.

In order to define proper criteria, we have taken 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.

Operators are the building blocks of SCADE models. A SCADE model is a network of operators, where data flows from one operator to another through connections. Each operator is characterized by a set of features, which are characteristic of the operator’s elementary functions.

The most basic feature for an operator corresponds to the fact that an operator is activated. This criterion is similar to the

procedure call criterion for code. In a SCADE network, all operators in a node N are systematically activated as soon as node N is activated. Activation of a node is determined by its activation condition, if there is one; otherwise it is activated at each cycle. So, this criterion is very weak, and thus we propose more relevant criteria.

The following example illustrates the kind of coverage criterion that we propose. It is based on the characteristic features of an operator and can be recorded for every instance of this operator during the execution of a model.

- 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.

A reasonable criterion corresponds to covering the following features:

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

6.4.2 LLR coverage analysis with SCADE Suite MTC

SCADE MTC (Model Test Coverage) is a module of SCADE Suite, which allows 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 has been simulated. 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.

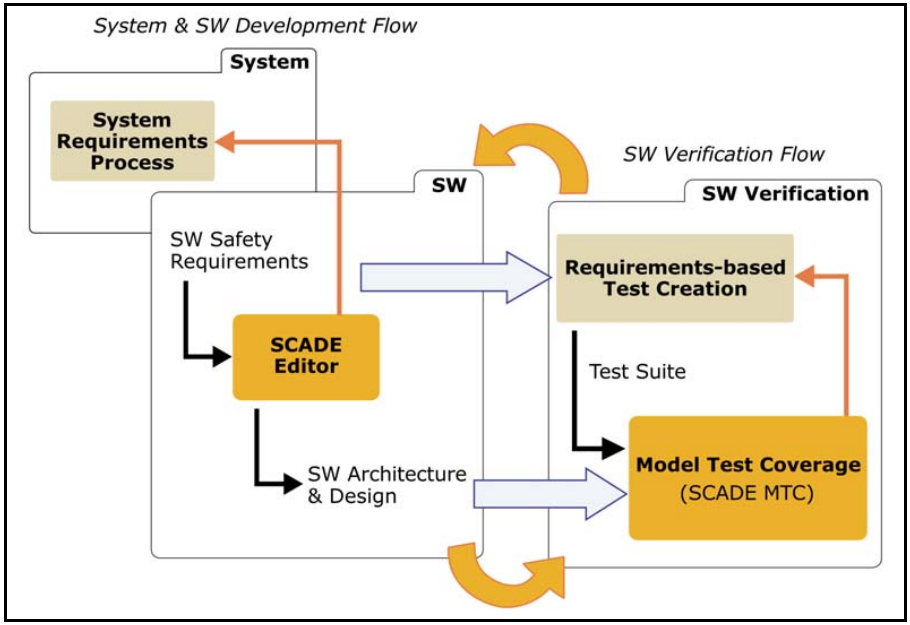


Figure 6.1: Position of SCADE Suite Model Test Coverage (MTC)

The use of SCADE MTC is decomposed in the following phases:

1. Model Coverage Acquisition: Running test cases in the SCADE 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: 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 in both.

Figure 6.2 below 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.

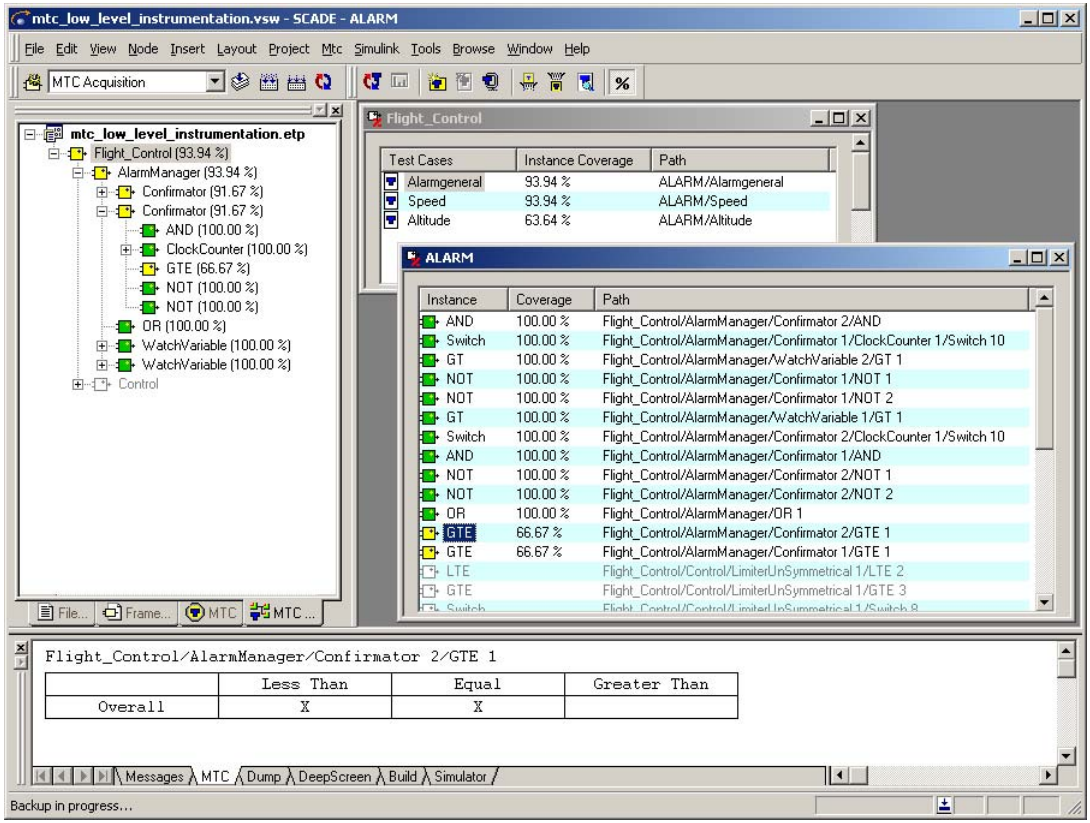


Figure 6.2: Using SCADE Suite Model Test Coverage (MTC)

Let us 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 of adding missing requirements-based test cases.

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

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

4 Deactivated parts in the SCADE model: In that case, resolution may consist of explaining the reason why a deactivated feature remains in the design.

EXAMPLE 1: INSUFFICIENT TESTING

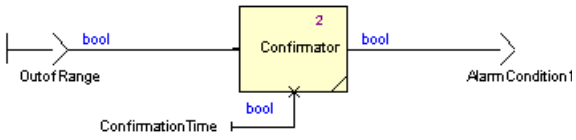


Figure 6.3: Non activated Confirmator

- Analysis: The Confirmator in Figure 6.3 was not raised during testing activities. Analysis concludes that the requirement is correct but testing is not sufficient.
- Resolution: Develop additional tests.

EXAMPLE 2: LACK OF ACCURACY IN THE HLR

The Integrator in Figure 6.4 was never reset during the tests: Is the “reset” behavior an unintended function?

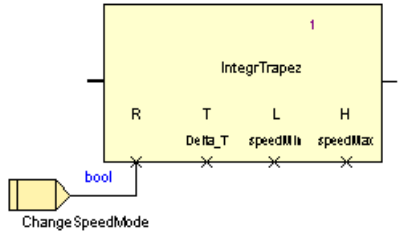


Figure 6.4: Uncovered “reset” activation

- Analysis: Resetting the filter here is a correct 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 can be introduced both during the design process and during the coding process. Structural code coverage analysis is needed to detect both categories.

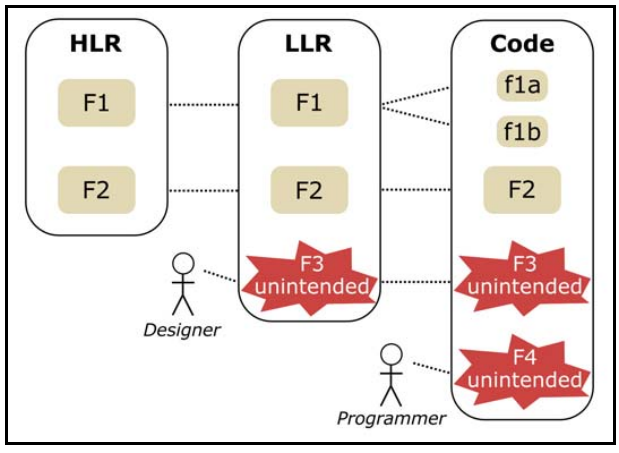


Figure 6.5: Sources of unintended functions in a traditional process

When using MTC and KCG as illustrated on Figure 6.6, unintended functions are detected and eliminated at the 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.

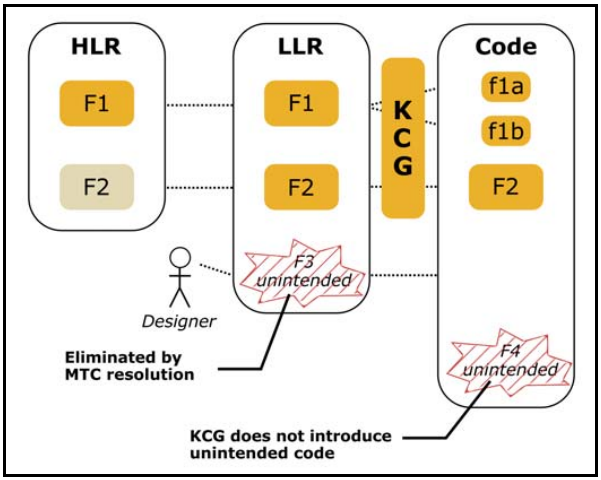


Figure 6.6: Elimination of unintended functions with MTC and KCG

6.5 Structural Coverage of the Source Code

6.5.1 Control structure coverage

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

Structural coverage has to be verified both on:

- the complete manual code;
- the C sample (see 5.5 The Combined Testing Process).

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

For a given implementation of a low-level requirement at a node, model coverage does not necessarily guarantee that every piece of generated code for that low-level requirement has been exercised in that context. This is

similar to a situation that may occur with run-time 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 will be 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]. While not a necessary part of the argument, analysis is facilitated by the simple nature of the generated code in the primitive blocks. Most blocks are implemented with three or fewer lines of C source. No block comprises more than twelve lines of C.

6.5.2 Data coupling and control coupling

DO-178B requires that test coverage of the data and control coupling are achieved. 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.*"

DATA COUPLING

Data coupling verification is based on the analysis of integration regarding:

- Interfaces between modules
- Handling of global data
- Input/output buffers sizing
- Etc.

The verification of data coupling is done as follows:

- Data coupling between elements of the SCADE model is verified automatically with the SCADE semantic Checker. SCADE KCG ensures that this consistent data coupling at the model level is correctly reflected in the C code.
- Data coupling between SCADE and manual code is verified manually in the traditional way.

CONTROL COUPLING

Control coupling verification is based on the analysis of integration regarding:

- Execution of call sequences
- Analysis of scheduling
- Analysis of WCET
- Etc.

The verification of control coupling is done as follows:

- Control coupling between elements of the SCADE model is verified automatically with the SCADE semantic Checker. SCADE KCG ensures that consistent control coupling at the model level is correctly reflected in the C code.
- Control coupling between SCADE and manual code is verified manually in the traditional way.

Moreover, MTC coverage verifies that control coupling and data coupling have been effectively exercised by test cases.

6.6 Summary of Verification of Verification

Table 6.2 summarizes verification objectives and methods for the verification activities.

Table 6.2: DO-178B Table A-7 Objectives Achievement

	Objective	Verification Method
1	Test procedures are correct.	Test procedure review. N.B.: concerns both host and target.
2	Test results are correct and discrepancies are explained.	Test results review. N.B.: concerns both host and target.
3	Test coverage of high-level requirements is achieved.	HLR coverage by combination of simulation and target tests.

Table 6.2: DO-178B Table A-7 Objectives Achievement (Continued)

	Objective	Verification Method
4	Test coverage of low-level requirements is achieved.	Model Test Coverage
5	Test coverage of software structure (modified condition/decision) is achieved.	MC/DC on C sample. MTC on SCADE model.
6	Test coverage of software structure (decision coverage) is achieved.	Decision coverage on C sample. MTC on SCADE model.
7	Test coverage of software structure (statement coverage) is achieved.	Statement coverage on C sample. MTC on SCADE model.
8	Test coverage of software structure (data coupling and control coupling) is achieved.	Semantic check of SCADE model. Manual verification of integration with manual code.

Appendixes and Index

A References

[C. André] "Representation and Analysis of Reactive Behaviors: A Synchronous Approach," proc. CESA'96, IEEE-SMC, Lille, France (1996).

[Amey] "Correctness by Construction: better can also be cheaper," Peter Amey, Crosstalk, the Journal of Defense Software Engineering, March 2002.

[ARP4754] "Certification considerations for highly integrated or complex aircraft systems," Society of Automotive Engineers, 1996.

[G. Berry] "The Foundations of Esterel. In "Proofs, Languages, and Interaction, Essays in Honour of Robin Milner," G. Plotkin, C. Stirling and M. Tofte, ed., MIT Press (2000).

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

[CAST-12] "Guidelines for Approving Source Code to Object Code Traceability," CAST-12 Position Paper, December 2002.

[NASA_MCDC] "A Practical Tutorial on Modified Condition/Decision Coverage," Kelly J. Hayhurst (NASA), Dan S. Veerhusen (Rockwell Collins), John J. Chilenski (Boeing), Leanna K. Rierson (FAA).

[DO-178B/ED-12B] "Software Considerations in Airborne Systems and Equipment Certification," RTCA/EUROCAE, December 1992.

[DO-248B] Final report for clarification of DO-178B "Software Considerations in Airborne Systems and Equipment Certification," RTAC Inc, October 2001.

[DO-254] "Design Assurance Guidance for Airborne Electronic Hardware," RTCA Inc.

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

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

[Order8110.49] "Software Approval Guidelines," FAA Order, February 6, 2003.

[Pilarski] "Cost effectiveness of formal methods in the development of avionics systems at Aerospatiale," François Pilarski,, 17th Digital Avionics Conference, November 1-5, 1998, Seattle, WA.

[SCADE_Lang] "SCADE Language Reference Manual," Esterel Technologies 2003.

[SCADE_DGL] "SCADE Design Guidelines," Esterel Technologies, 2005 (to be published).

B Acronyms and Glossary

ACRONYMS

A/C	Aircraft
COTS	Commercial Off-The-Shelf.
CVK	Compiler Verification Kit
DER	Designated Engineering Representative
DV	SCADE Design Verifier
EUROCAE	European Organization for Civil Aviation Equipment
FAA	Federal Aviation Administration
HLR	High-level Requirement
KCG	SCADE Qualified Code Generator
LLR	Low-level Requirement.
JAA	Joint Aviation Authorities
JAR	Joint Aviation Requirements
MC/DC	Modified Condition/Decision Coverage
N/A	Not Applicable
RTCA	Requirements and Technical Concepts for Aviation, RTCA, Inc.
SCADE	Safety Critical Application Development Environment
SQA	Software Quality Assurance
SW	Software

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 has been 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, safety-related requirements, and system architecture.
Host computer	The computer on which the software is developed.
Independence	Separation of responsibilities, which ensures the accomplishment of objective evaluation. (1) For software verification process activities, independence is achieved when the verification activity is performed by a person(s) other than the developer of the item being verified, and a tool(s) may be used to achieve an equivalence to the human verification activity. (2) For the software quality assurance process, independence also includes the authority to ensure corrective action.
Integral process	A process 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 has been 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 has been 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 KCG 4.2

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.

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 has been 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 KCG 4.2

The SCADE KCG 4.2 Code Generator has been 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:
  - Reference Manual of the SCADE and Lustre languages.
  - Requirements Data of SCADE/KCG 4.2.
- The Tool Qualification Plan (TQP) presents the strategy and the organization for the Qualification of the KCG 4.2 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.

C-3 SCADE KCG 4.2 Life-Cycle Documentation

Table C.1 describes the documents that must be considered by the Certification Authorities during the qualification audit of KCG 4.2.

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 KCG 4.2 Qualification Kit that is delivered to SCADE users.

Documents that are "*Available*" can be audited by the Certification Authorities.

Table C.1: Documents required for SCADE KCG 4.2 qualification audit by Certification Authorities

Data	DO-178B 7 FAA Requirement	SCADE Suite™ KCG Package	DO-178B Reference
Tool Qualification Plan	Submit	Tool Qualification Plan of KCG	12.2.3.a(1), 12.2.3.1, & 12.2.4
Tool Operational Requirements	Available	- Version Content - Software requirements data of KCG, S2L, and L2C - Reference Manual of SCADE & Lustre Languages	12.2.3.c(2) & 12.2.3.2
Tool Accomplishment Summary	Submit	Tool Accomplishment Summary of KCG	12.2.3.c(3) & 12.2.4
Tool Verification Records (for example, test cases, procedures, and code)	Available	Accessible at Esterel Technologies premises	12.2.3
Tool Qualification Development data (for example, requirements, design, and code)	Available	Accessible at Esterel Technologies premises	12.2.3
Software Configuration Index	Submit	Software Configuration Index of KCG	9.3

D The Compiler Verification Kit (CVK)

D-1 CVK Product Overview

WHAT CVK IS

CVK is a test suite, whose purpose is to verify that a compiler correctly compiles code generated by SCADE KCG.

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

WHAT 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, proper number 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.

ROLE OF CVK

CVK is a test suite: it is part of verification means of the KCG users.

Figure D.1 shows the complementary roles of KCG and CVK in the qualification of the customer's development environment from Esterel Technologies to the customer.

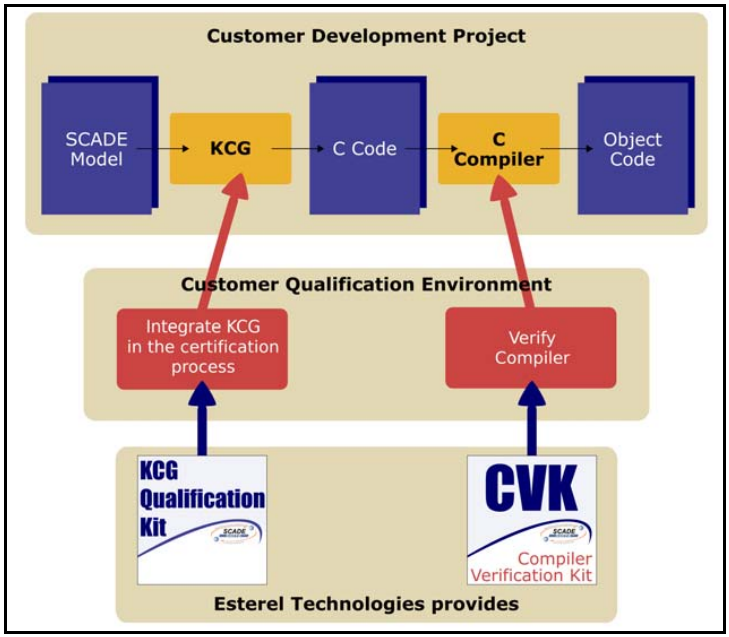


Figure D.1: Role of KCG and CVK in the qualification of the customer's development environment

CVK CONTENTS

The CVK product is composed of the following items:

- A CVK User's Manual containing:
  - Installation and use instructions
  - Description of the underlying methodology
  - Product structure description
  - SCADE models description
  - C sample description
  - Test cases and procedures description
- The SCADE-generated C sample to verify the C compiler.
- The associated SCADE models. This enables customers to generate the C sample with KCG in their environment and allows them to compare the generated code with the reference code provided in the CVK test suite and verify their KCG installation.
- Test cases to cover the SCADE C subset with an MC/DC of 100% coverage for all KCG settings.
- Automated test procedures for Windows platform.

C SAMPLE CHARACTERISTICS

The C sample exhibits the following characteristics:

- It contains all individual C constructs that can ever be generated by KCG 4.2 from a correct SCADE model.

- It contains the C code that is generated for each SCADE operator.
- It contains supplementary combinations for the specific nesting of C constructs that can be produced. Examples:
  - Up to five operands for operators such as +, as in “A + B + C + D + E” or “(A x B) + (C x E)” etc.
  - Using selection in a deep structure, as in “A.B.C.D.E”.
  - A node with 800 inputs.

D-2 Motivation for Sample-Based Testing

The source code generated by KCG is a small subset of C with a low level of complexity:

- It is composed of a sequence of basic blocks and/or calls to C functions implementing children nodes, as shown in Figure D.2 when expansion is not used.
- Memory allocation is fully static (no dynamic memory allocation).
- There is no recursion and no looping (except a few local, fixed-size loops).
- The code is decomposed into elementary assignments to local variables (this restricts use of the optimization options of the C compiler + SCADE KCG option).
- There is only single static assignment (SSA) to a C variable within a C function.
- Expressions are explicitly parenthesized.
- No dynamic address calculation is performed (no pointer arithmetic).
- It contains no array indexing (since there are no arrays, except for FBY).
- 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.

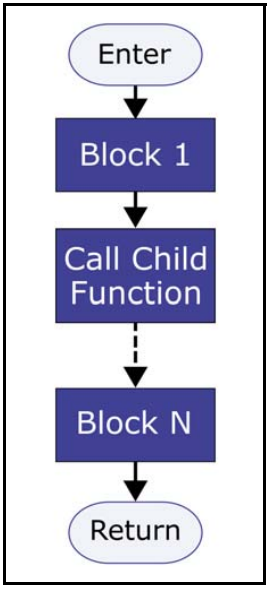


Figure D.2: Sequential structure of the generated code (without expansion)

The basic blocks composing the software do not contain branching outside themselves. Table D.1 below summarizes which blocks may or not contain control structures that are local to their enclosing blocks. Generally, these control structures are not nested; they are put in sequence. The only exception is generated when expanding a node under condact or following a

"when" operator; this requires verifying that the call depth of the expanded subtree is lower than the call depth of the one used with CVK..

Table D.1: Type of code generated from SCADE operators

SCADE Construct	Control Structure in the Generated Block
Predefined arithmetic operators	None
Structured data handling	None
Comparison operators	None
Boolean operators (and, or, xor, not, sharp)	None in source code. Local branching in the machine code, if bitwise option is not used.
Selection operator (if, case)	Local to the block
->, pre, fby	Local to the block
"condact" operator	Local to the block if not expanded. "if" nested down to the next unexpanded node otherwise.
"when" operator	"if" nested down to the next unexpanded node.

These building blocks define a set of equivalence classes. Thus, a representative sample of those building blocks, covering these equivalence classes, is used to verify that the compiler correctly compiles these building blocks into machine code.

This approach is similar to the approach proposed in [CAST-12] for the analysis of source to object code traceability, in the sense that verification can be performed on a sample when there are strict coding standards, restricting the programming language to a subset:

"*The applicant may elect to impose coding standards throughout the program that might limit the number of programming operations and libraries to a smaller subset of the programming language and compiler-provided library functions. If evidence can be provided that the operational program and library functions fully comply with the coding standards, then the certification authority may accept analysis that is done on the subset. Once all of the existing code can be shown to be in compliance with the coding standards, one or more tests combining all of the constructs specified in the coding standards can be produced and compiled. To establish the validity of the test programs, the applicant should document the results of a comparison between the results of the analysis and some representative code from the actual operational program. The representative code should be chosen from complex functions where a higher probability of added, untraceable functionality might exist (for example, software handling arrays, potential hidden calls to libraries, exception handlers, traps or loop constructs).*"

D-3 Strategy for Developing CVK

Figure D.3 summarizes the strategy for developing and verifying CVK.

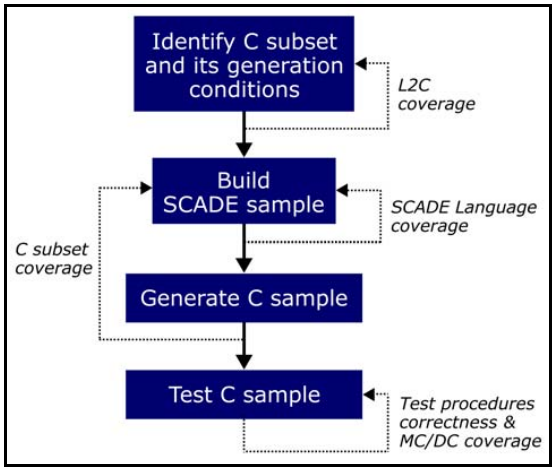


Figure D.3: Strategy for developing and verifying CVK

CVK has been built in the following way:

1 Identify the C subset generated by KCG by analyzing the L2C requirement specification (L2C is the back-end of KCG generating the C code). Coverage of L2C is verified. Elementary C constructs are identified in terms of the C-ISO standard.

2 Build the C sample.

   a A SCADE sample covering all SCADE constructs is built as material for code generation.

   b Each C sample is obtained by combining a SCADE sample item with appropriate KCG options.

   c Coverage of the C subset by the C sample is verified. Coverage of the SCADE language is also verified as a secondary objective, although it is not required for the verification of the compiler itself. If necessary, additional items are added in the SCADE sample and/or additional KCG options are exercised.

3 Develop a set of input vectors and expected output vectors to test C sample. These tests are executed on a host platform to verify:

   a Conformance of outputs

   b MC/DC coverage of the code

D-4 Use of CVK

CVK is used as follows (Figure D.4):

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

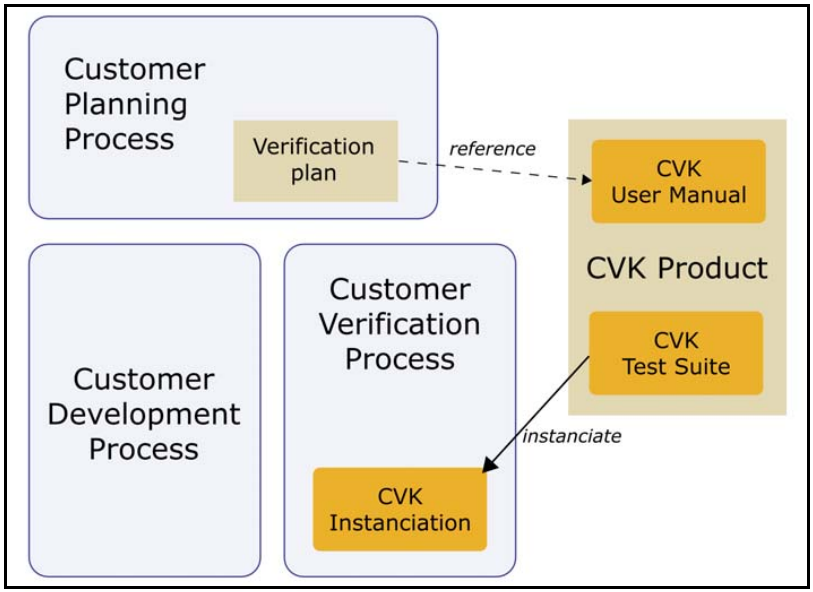


Figure D.4: Use of CVK items in the customer's processes

Figure D.5 details the role of CVK items (colored boxes) in the verification of the compiler:

- The C sample is generated by KCG from the SCADE sample with specified KCG options. This is done on the host computer.
- From the C sample, the C cross-compiler/linker generates an executable. This can be done on the host.
- The executable is run on the target processor. It reads input vectors and computes output vectors.
- The output vectors are compared to reference vectors, based on the requirements, namely the semantics of the C sample, which is the same as the semantics of the SCADE model. This can be done on the host computer.

If there is any difference between the collected results and the reference results, then analysis has to be conducted to find the origin of the difference(s). If it is an error in the use or contents of CVK, then this has to be fixed. If it is due to an error in the compiler, then this probably means that this compiler has to be rejected, since the SCADE-generated code is very simple, and if the compiler is not able to compile it correctly, it means it is definitely not reliable.

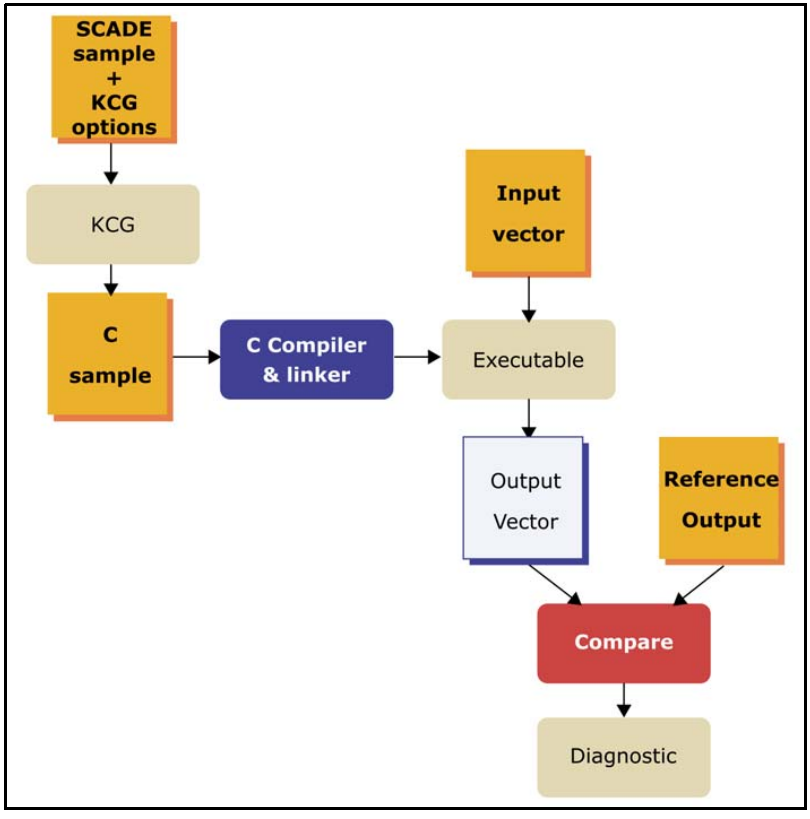


Figure D.5: Position of CVK items in the compiler verification process

The cross-compiler/linker has to be run with the same options as for the manual code and as for the rest of the KCG-generated code.

Index

Index

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