# Efficient Development of Airborne Software with SCADE Suite™

***Abstract***

*This white paper addresses the issue of cost and productivity improvement in the development of safety-critical software for avionics systems. Such developments, driven by the ED-12/DO-178B guidelines traditionally require very difficult and precise development and verification efforts. This paper first reviews traditional development practices and then the optimization of the development process with the SCADE Suite methodology and tool. SCADE Suite supports "correct by construction" and automated production of the life cycle elements. The effects on savings in the development and verification activities are presented in detail. Industry examples demonstrate the efficiency of this approach, and business benefits are analyzed.*

**Authors**

Jean-Louis Camus (Esterel Technologies)
Jean-Louis.Camus@esterel-technologies.com
Bernard Dion (Esterel Technologies)
Bernard.Dion@esterel-technologies.com

# Table of Contents

# List of Figures

# 1 Executive Summary

Companies developing avionics software are facing a challenge. Safety is an absolute requirement. This makes the development of such systems very expensive, as shown by the figures below, observed for avionics software:

- The average development and test of 10K Lines of Code (KLOC) level B software is 16 person-years
- The cost of a minor bug is in the range $100K-$500K
- The cost of a major bug is in the range $1M-$500M

The growing complexity of those systems increases the cost and time for their development to a level that conflicts with business constraints such as time-to-market and competitiveness.

This paper addresses the issue of productivity in the development of software for civil aircrafts, as specified in the guideline ED-12/DO-178B and explores the nature of these costs and how to reduce them by adopting efficient methods and tools.

First, an introduction to DO-178B guidelines is provided, to illustrate how safety objectives lead to high costs when traditional development processes are used. In particular, verification activities are analyzed: including testing, analysis and review requirements. Also explored is the process for change or error correction. It is shown that these verification activities are the cost-driver and the bottleneck in project schedule.

Next, SCADE Suite, an encompassing method and tool, is described. SCADE Suite maintains the highest quality standards while reducing costs based on a “correct by construction” approach SCADE Suite provides:

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

Industrial application examples at Eurocopter and Airbus are described, which demonstrate the efficiency of the approach.

Finally, business benefits of the investment in SCADE Suite are analyzed. They include a 50% cost reduction compared to the traditional approach, a dramatic reduction in the time to implement a change (from weeks to days), and a significant improvement in the ability to reuse components, all leading to a high competitive advantage.

Details of effort reduction are given in appendix.

# 2 The Challenge of Developing Software for Safety-Critical Systems

Companies developing avionics software are facing a challenge. Safety is not an option but an absolute requirement. Traditionally this has made the development of such systems very expensive, as shown by the figures below, observed for avionics software:

- The average development and test of 10K Lines of Code (KLOC) level B software is 16 person-years.
- The average avionics software is 46% over budget and 35% behind schedule.
- The time to market is 3-4 years.
- The cost of a minor bug is in the range $100K-$500K.
- The cost of a major bug is in the range $1M-$500M.

To compound this, the amount and complexity of software increase every year. As an example, the progression of software size for the Airbus family of avionics software is shown below:

| Aircraft | A310 (70’) | A320 (80’) | A340 (90’) |
|---|---|---|---|
| Number of digital units | 77 | 102 | 115 |
| Volume of onboard software in Mbytes | 4 | 10 | 20 |

The decades of the ‘70’s, 80’s, and 90’s have also seen increasing competitive pressures and an explosion of costs.

In order to complete projects within cost and schedule constraints, companies developing safety-critical software feel they have had no choice but to innovate in their development processes.

In this paper we address the issue of productivity in the development of airborne software, as guided by ED-12/DO-178B. We will identify the areas of cost and explore how to reduce them by adopting efficient methods and tools.

# 3 The ARP 4754 and ED-12/DO-178B Guidelines for the Development of Safety-Critical Systems and Software

***This chapter introduces ED-12B/DO-178B and reviews the activities performed over the development cycle when following ED-12/DO-178B guidelines. It also defines some of the terminology particular to DO-178B. We will then analyze why traditional development is so expensive.***

## 3.1 What Are DO-178B and ARP 4754?

The avionics industry requires that safety-critical software be assessed according to strict United States Federal Aviation Administration (FAA) and Europe Joint Aviation Authority (JAA) guidelines before it may be used on any commercial airliner. ARP 4754 and DO-178B are guidelines, which are used both by the companies developing airborne equipment and by the certification authorities.

### 3.1.1 ARP 4754

ARP 4754 [ARP4754] 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 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, 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 RTCA document DO-178B and its EUROCAE counterpart, ED-12B. Coverage of complex hardware aspects of design are dealt with in RTCA document DO-254 [DO-254] .

### 3.1.2 DO-178B

ED-12/DO-178B [DO-178B/ED-12B] was first published in 1992 by EUROCAE (a non-profit organization addressing aeronautic technical problems) and RTCA (Requirements and Technical Concepts for Aviation). It was written by a group of experts from certification authorities and companies developing airborne software. It provides guidelines for the production of software for airborne systems and equipment. The objective of the guideline is to assure 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 that indicate that the objectives have been satisfied.

### 3.1.3 Relationship between ARP 4754 and DO-187B

ARP 4754 and DO-178B are complementary guidelines:

- ❑ ARP 4754 provides guidelines for the system level processes.
- ❑ DO-178B provides guidelines for the software development processes.

The information flow between the system processes and software processed is summarized by Figure. 1.

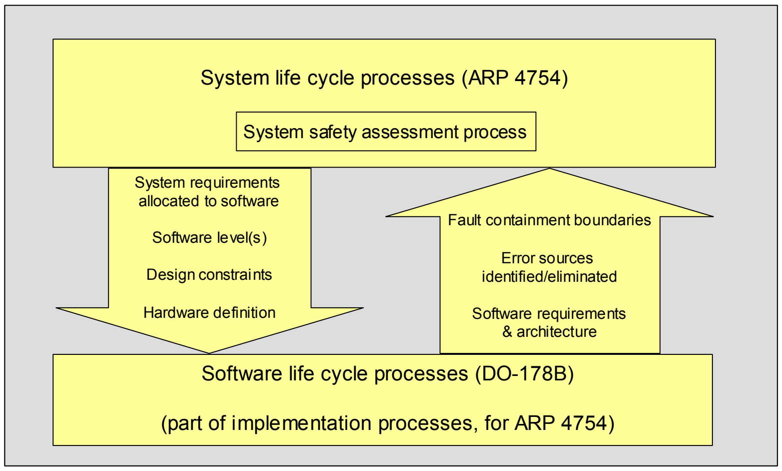

System life cycle processes (ARP 4754)
System safety assessment process
System requirements allocated to software
Software level(s)
Design constraints
Hardware definition
Fault containment boundaries
Error sources identified/eliminated
Software requirements & architecture
Software life cycle processes (DO-178B)
(part of implementation processes, for ARP 4754)

**Figure. 1 Relationship between ARP4754 and DO-178B Processes**

ARP 4754 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.*

### 3.1.4 Development Assurance Levels

ARP4754 defines guidelines for the assignment of so called "Development Assurance Levels" to the system, to its components, and to software, related to the most severe failure condition of the corresponding part.

ARP4754 and ED-12/DO-178B define in common five "Development Assurance Levels":

| Level | Effect of Anomalous Behavior |
|---|---|
| A | Catastrophic failure condition for the aircraft (ex: aircraft crash) |
| B | Hazardous/severe failure condition for the aircraft (ex: several persons could be injured). |

| Level | Effect of Anomalous Behavior |
|---|---|
| C | Major failure condition for the aircraft a (ex: flight management system could be down, the pilot would have to do it manually). |
| D | Minor failure condition for the aircraft (ex: some pilot-ground communications could have to be done manually). |
| E | No effect on aircraft operation or pilot workload (ex: entertainment features may be down). |

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

### 3.1.5 Objectives are the Essence of DO-178B

The essence of DO-178B is the formulation of appropriate objectives, and the verification that these objectives have been achieved. The authors of DO-178B 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 may vary over time, with the evolution of methods, techniques and tools. DO-178B never states that one should use design method X, or 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. The development assurance level of the software.
  b. The design standards.
- ❑ Define procedures for the verification of the objectives. The achievement of the objectives form transition criteria from one activity to another. For instance:
  a. Design review.
  b. Software integration testing.
- ❑ Define procedures for verifying that the above-mentioned verification activities have been performed satisfactorily. For instance:
  a. Remarks of document reviews have been answered.
  b. Structural coverage of code is achieved.

### 3.1.6 DO-178B Processes Overview

ED-12/DO-178B structures activities as a hierarchy of “processes”, depicted by Figure 2. The term “process” will appear 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. This process is beyond the scope of this paper.
- 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 ensures 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.

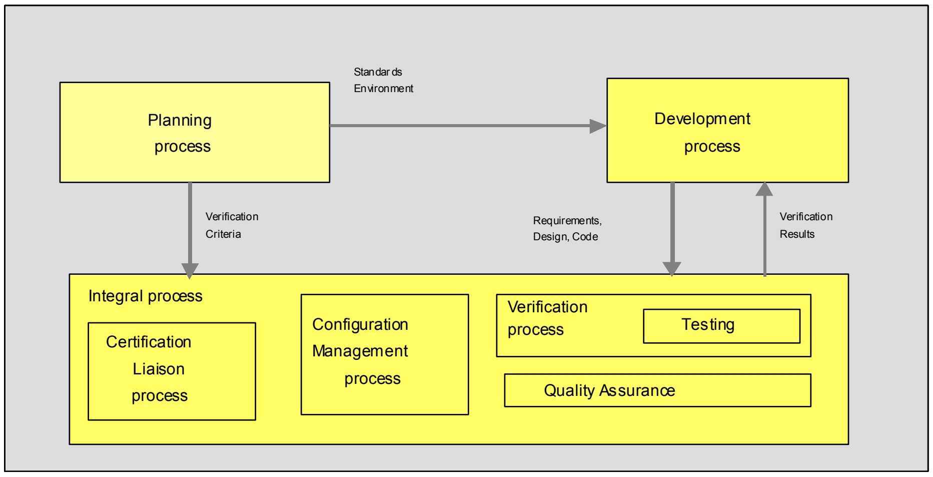


**Figure 2 DO-178B Life-cycle processes structure**

In the remainder of this document we will focus on the development process and on the corresponding planning and verification activities.

## 3.2 DO-178B Development Processes

The software development processes, as depicted by Figure 3 are composed of:

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

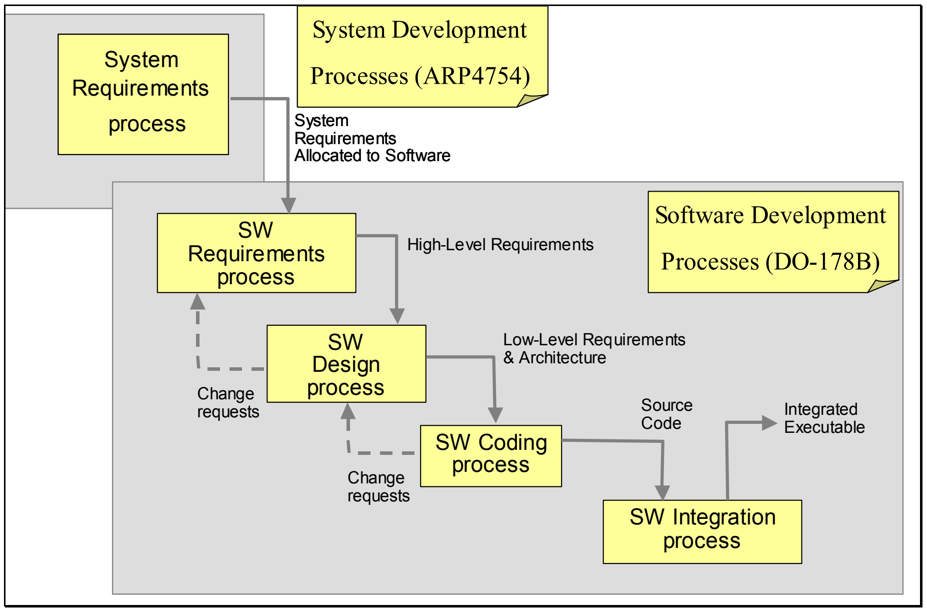

System Development Processes (ARP4754)
System Requirements process
System Requirements Allocated to Software
Software Development Processes (DO-178B)
SW Requirements process
High-Level Requirements
SW Design process
Low-Level Requirements & Architecture
Change requests
SW Coding process
Change requests
Source Code
SW Integration process
Integrated Executable

**Figure 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, 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 communications mechanisms, and software components. If the system contains "deactivated" code (see glossary), 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.

## 3.3 DO-178B Verification Processes

### 3.3.1 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 latter 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, while analyses are more detailed and should be reproducible (ex: conformance to coding standards).

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

### 3.3.2 Reviews and Analyses of the High-Level Requirements

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

a. Compliance with the system requirements.

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

c. Compatibility with target computer.

d. Verifiability: Each HLR has to be verifiable.

e. Conformance to standards, as defined by the planning process.

f. Traceability with the system requirements.

g. Algorithm accuracy.

### 3.3.3 Reviews and Analyses of the Low-Level Requirements

The objective of these reviews and analyses is to detect and report requirements 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: Each low-level requirement is accurate and unambiguous and the low-level requirements do not conflict with each other.
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. Conformance to standards: The Software Design Standards (defined by the software planning process) were followed during the software design process, and deviations from the standards are justified.
f. Traceability: Ensure that the high-level requirements were developed into the low-level requirements.
g. Algorithm aspects: Ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities (ex: mode changes, crossing value boundaries).
h. The SW architecture is compatible with the HLR, consistent, compatible with the target computer, verifiable, and conforms to standards.
i. Software partitioning integrity is confirmed.

### 3.3.4 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 conformance to 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, and 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. Conformance to 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 software low-level requirements were developed into Source Code.
f. Accuracy and consistency: The objective is to determine the correctness and consistency of the Source Code, including 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 interrupt conflicts.

### 3.3.5 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 the errors which could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed.

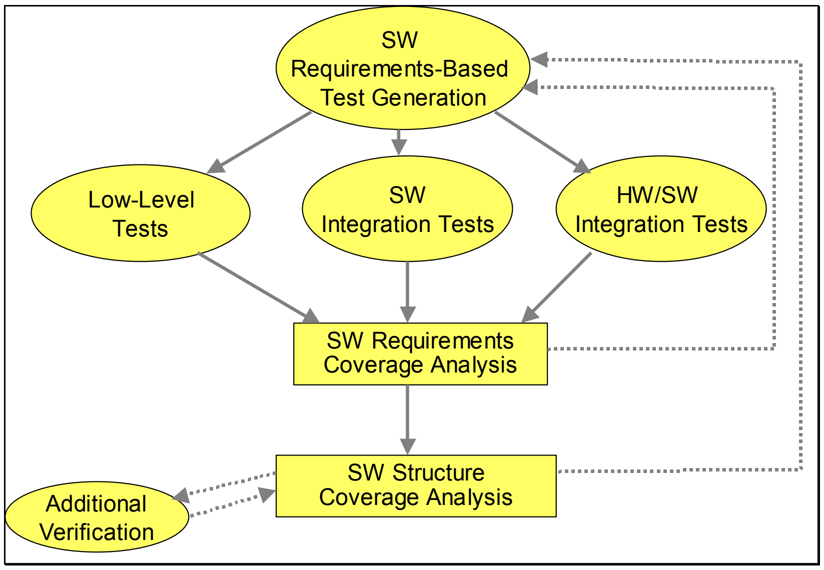

SW Requirements-Based Test Generation
Low-Level Tests
SW Integration Tests
HW/SW Integration Tests
SW Requirements Coverage Analysis
SW Structure Coverage Analysis
Additional Verification

**Figure 4 DO-178B Testing processes**

There are 3 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 by Figure 4, DO-178B imposes that all test cases are requirements-based; that means that test procedures have to be written from the specifications, not from the code. This is even imposed for low-level testing.

### Test Coverage Analysis

Test coverage analysis is a two-step process:

- Requirements-based test coverage analysis determines how well the requirements-based testing covered the software requirements
- Structural coverage analysis determines which code structures were exercised by the requirements-based test procedures.

### 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 need for re-verification
- If it is deactivated code (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: 100% statement coverage is required, which means that every statement in the program has been exercised.
- Level B: 100% decision coverage is required. That means that every decision has taken all possible outcomes at least one (ex: then/else for an if construct) and that every entry and exit point in the program has been invoked at least one.
- Level A: 100% MC/DC (Modified Condition/Decision Coverage) is required for level A software, which 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 fragment:

```
If A OR (B AND C)
Then do something
Else do something else
Endif
```

requires 4 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 |

## 3.4 Why is Traditional Development of Safety-Critical Software so Expensive?

*Traditional development of safety critical software leads to high costs, as shown by the figures given in chapter 2. In this section, we give some of the reasons that lead to these costs. We show that these costs are not due to DO-178B, but due to the way software is developed.*

### 3.4.1 Multiple Descriptions

In a traditional development process, the software is described in several places:

- ❑ System requirements allocated to software.
- ❑ Software high-level requirements.
- ❑ Software low-level requirements.
- ❑ Software source code.

At each step, the description of the software is rewritten into another form.

This rewriting is not only expensive, it is error-prone. There is a major risk to have inconsistencies between these different descriptions. This results in a huge amount of effort being devoted to the verification of the compliance of each level to the previous level. The purpose of many of the activities described in DO-178B is to detect the errors introduced during transformations from one written form to another.

### 3.4.2 Ambiguity and Lack of Accuracy of Specifications

Requirements and design specifications are traditionally written in 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 several parallel sentences containing “before X” or “after Y”?

### 3.4.3 Manual Coding

Coding is the last transformation in the traditional development process.

It takes as input the last formulation in natural language (or pseudo code). The programmer generally has a limited understanding of the system, which makes him vulnerable to ambiguities in the specification.

He produces code, which is difficult/impossible to understand by the author of the 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’s verification effort is consumed by code testing.

### 3.4.4 Late Detection of Specification and Design Errors

Many specification and design errors are only detected during software integration testing.

One reason is that the requirements/design specification is often ambiguous and subject to interpretation. The other reason is that it is difficult for a human reader to understand details, regarding dynamic behavior without being able to exercise it.  In a traditional process, the first time one can exercise the software is during integration. This is 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.

### 3.4.5 Complexity of Updates

There are many sources of changes in the software, ranging from bug fixing to function improvement or the introduction of new functions.

When something has to be changed in the software, all products of the software life cycle have to be updated consistently and all verification must be performed accordingly.

### 3.4.6 The Stakes of Verification Efficiency

The level of verification for avionics software is much higher than for other commercial software. For Level A software, the overall verification cost (including testing) may account for up to 80% of the budget [Amey] 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 paper is to show how to retain a complete and thorough verification process but dramatically improve the efficiency of the process. The methods we will describe achieve at least the level of quality achieved by traditional means by optimizing the whole development process.

# 4 Model Based Development with SCADE Suite

## 4.1 What is SCADE Suite?

### 4.1.1 A Model Based Development Environment

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

It supports a model-based development paradigm, as illustrated by Figure 5.

- ❑ The model is the software specification: It is the unique reference in the project.
- ❑ Documentation is automatically 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 applied to the model to detect corner bugs or prove safety properties.
- ❑ Code is automatically generated from the model with the Qualifiable Code Generator: The code is correct and up-to-date by construction.

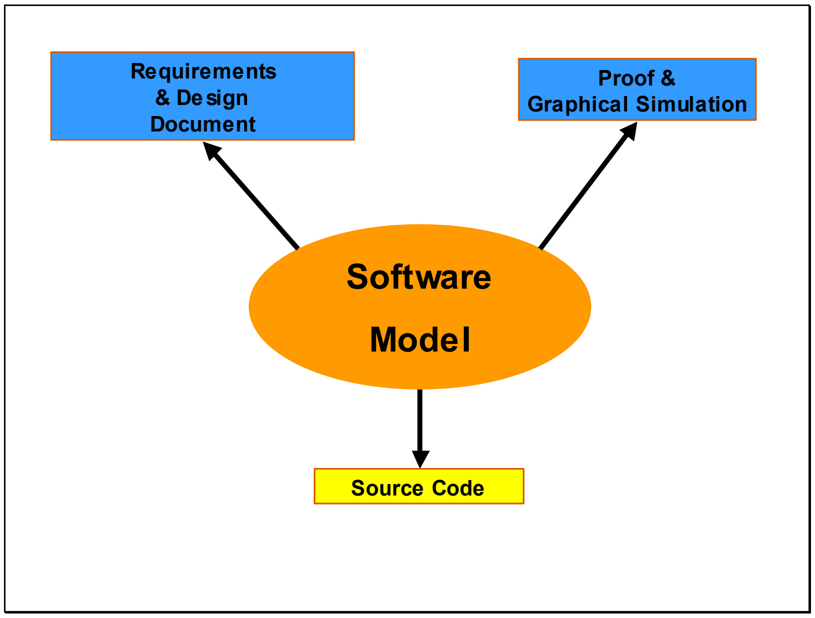


**Figure 5 Model-based development with SCADE**

SCADE Suite applies the "golden rules":

- **Share unique, accurate specifications**
- **Do things once**: Do not rewrite descriptions from one activity to the other, for instance between system design and software requirements, between HLR and LLR, LLR and code.
- **Do things right:** Detect errors in early stages and/or write "correct-by-construction" descriptions.

### 4.1.2 A Qualifiable Development Environment (QDE)

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

Tools have to be "qualified" when processes described in DO-178B are eliminated, reduced or automated (DO-178B, section 12.2). SCADE Suite has been specified and developed with qualification objectives. This goes far beyond careful development of the tools. It also requires appropriate definition of the modeling techniques, and of the generated code characteristics such as traceability and safety. Appendix D provides details about the qualification of the code generator.

The next section shows how SCADE Suite can be used in the development and verification process.

### 4.1.3 SCADE Suite Position in the Development Flow

SCADE Suite can be used in the following activities, as shown in green on Figure 6:

- Definition of High-level and Low-level Requirements:
  - The Editor: Supports the edition, documentation and verification of models.
  - The Simulator: Supports interactive or batch simulation of a model.
  - The Design Verifier: Supports corner bug detection and formal verification of requirements.
  - The Simulink to SCADE Gateway: supports translation of discrete time Simulink models to SCADE, and S-function generation from SCADE.
- The Code Generators: Automatically generate C or ADA code from the model. One of them (KCG) is qualifiable as a development tool for Level A software.
- The SCADE-DOORS gateway supports traceability management between SCADE and other documents such as requirements or test plans.

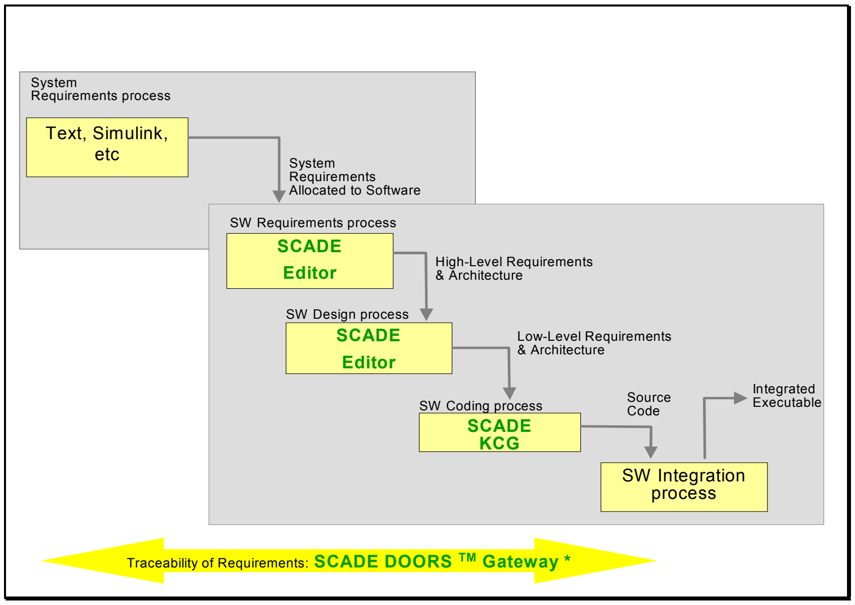


**Figure 6 SCADE Suite position in the DO-178B development processes**

## 4.2 SCADE Modeling Techniques

SCADE Suite uses two specification formalisms familiar to control engineers: block diagrams for continuous control and state machines for discrete control. What the synchronous modeling techniques of SCADE Suite add is a very rigorous view of these well-known but often insufficiently defined formalisms. This view includes a precise definition of concurrency and a proof that all SCADE programs behave deterministically.

### 4.2.1 Block Diagrams for Continuous Control

By *continuous control*, we mean sampling sensors at regular time intervals, performing signal-processing computations on their values, and outputting values often using complex mathematical formulae. Data is continuously subject to the same transformation. In SCADE, continuous control is graphically specified using block diagrams such as the one depicted in Figure 7.

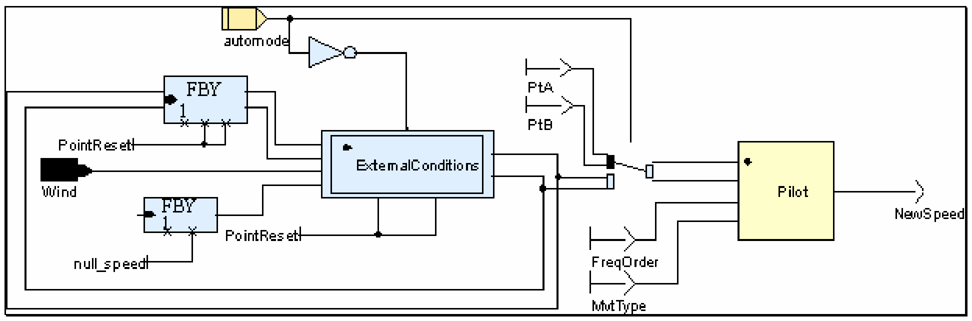


**Figure 7 A SCADE Suite block diagram**

Boxes compute mathematical functions, filters, and delays, while arrows denote flows of data between the boxes. Data flow between blocks that continuously compute their outputs from their inputs. All blocks compute concurrently, and the blocks only communicate through the flows. To add some flexibility in functioning modes control, some flows may carry 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 7, the *ExternalConditions* block is hierarchical, and one can zoom into it with the editor. Hierarchy makes it possible to break design complexity by a divide-and-conquer approach and to design reusable library blocks. Because of support for hierarchy, the set of primitive blocks can remain very small: there is no need to write complex blocks directly in C or ADA, since defining them hierarchically from smaller blocks is semantically better, much more readable, and just as efficient. Compared to other block-diagram formalisms, hierarchy in SCADE is purely architectural and does not imply complex hierarchical evaluation rules: a hierarchical block occurring in a higher-level block is simply replaced by its contents, conceptually removing its boundaries.

### 4.2.2 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, e.g. value threshold detection. Discrete control is when the behavior keeps changing, a characteristic of modal human-machine interface, alarm handling, complex functioning mode handling, or communication protocols.

State machines have been very extensively studied in the last 50 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, the initial one being Statecharts [D. Harel]. The Esterel Technologies hierarchical state machines are called Safe State Machines (SSMs), see

Figure 8 for an example. These evolved from the Esterel programming language [G. Berry] and the SyncCharts state machine [C. André] model. SSMs have proved to be scalable in large avionics systems.

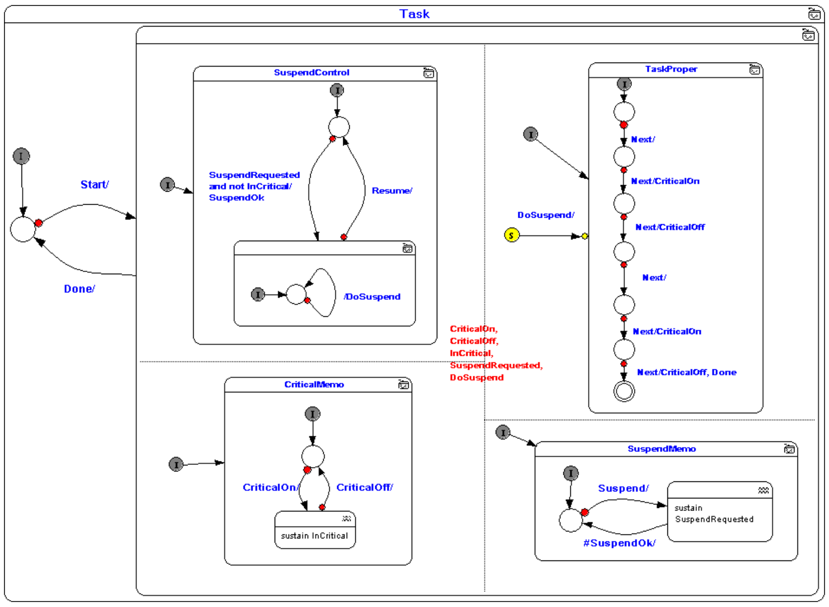


**Figure 8 Hierarchical and concurrent state machines**

SSMs are hierarchical and concurrent. States can be either simple states or macro states, themselves recursively containing a full SSM or a concurrent product of SSMs. 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 pre-empted whichever state they were in. Concurrent state machines communicate by exchanging signals, which may be scoped to the macro state that contains them.

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

### 4.2.3 Mixed Continuous / Discrete Control

Large applications contain cooperating continuous and discrete control parts. SCADE Suite makes it possible to seamlessly couple both data flow and state machine styles. Most often, one includes SSMs into block-diagrams design to compute and propagate functioning modes. Then, the discrete signals to which a SSM reacts and which it sends back are simply transformed back-and-forth into Boolean data flows in the block diagram. The computation models are fully compatible.

### 4.2.4 The Cycle-Based Intuitive Computation Model

The cycle-based model is a direct computer implementation of the ubiquitous sampling-actuating model of Control Engineering. It consists of performing a continuous loop of the form pictured in Figure 9. In this loop, there is a strict alternation between environment actions and program actions. Once the input sensors are read, the program starts computing the cycle outputs. During that time, the program is *blind to environment changes*. When the outputs are ready, or at a given time determined by a clock, the output values are fed back to the environment, and the program waits for the start of the next cycle.

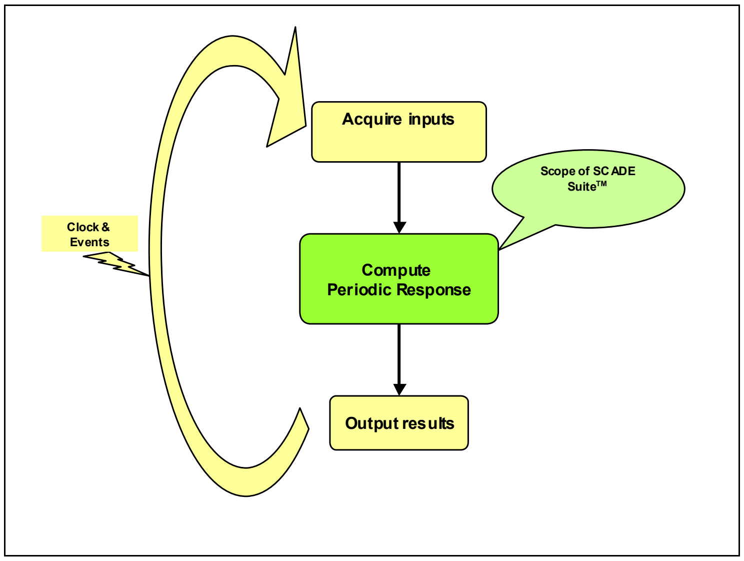


**Figure 9 The cycle based model**

In a SCADE block diagram specification, each block has a cycle and all blocks act concurrently. Blocks can all have the same cycle, 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 two connected blocks 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 *synchronous semantics.*

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 synchronously with each other, receiving the signals sent by other machines and possibly sending signals back. Finally, block diagrams and SSMs in the same design also communicate synchronously at each cycle.

Notice that this cycle-based computation model carefully distinguishes between *logical concurrency* and *physical concurrency*. The application is described in terms of logically concurrent activities, block-diagram or SSMs. Concurrency is resolved at compile-time, and the generated code remains standard sequential and deterministic C or ADA code, all contained within a very simple subset of these languages. 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.

### 4.2.5 Determinism

*Determinism* is a key requirement of most embedded applications. A system is deterministic if it always reacts in the same way to the same inputs occurring with the same timing. On the contrary, a non-deterministic system can react in different ways to the same inputs, actual reaction depending on internal choices or computation timings. It is clear that determinism is a must to drive a car or a plane: internal computation timings should not interfere with the driving algorithms. The plane should not decide by itself to go right or left. The same applies to man/machine interface and alarm handling.

## 4.3 The Development Processes with SCADE Suite

*In this section we present how SCADE Suite can be used for development activities. We will illustrate this with examples taken from the ACS, a toy example. The verification activities are described in the following section.*

### 4.3.1 The ACS Example

#### 4.3.1.1 Introduction

We will illustrate the SCADE Suite development process with a small example: the Altitude Control System.

*Note: this example has been derived from an altitude control system produced by the SafeAir project[1]. It is an element of a case study and was intended to be used for training. The original specification can be retrieved as a public document from the SafeAir WEB site (www.safeair.org).*

The Altitude Control System (ACS) of the A/C is defined as all the hardware and software necessary to control the given 2-dimensional speed/altitude of the A/C. The heart of the ACS is the Flight Control Computer (FCC).

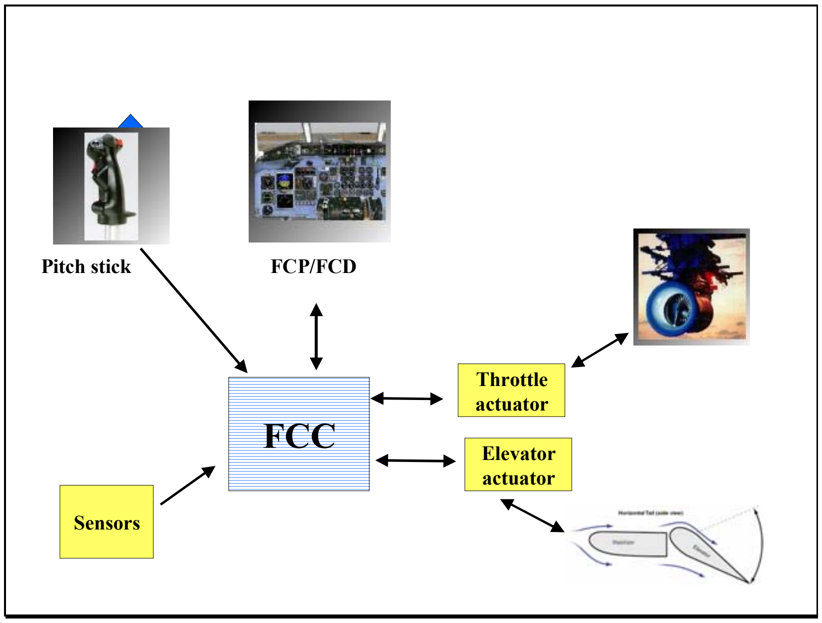


**Figure 10 The Altitude Control System**

#### 4.3.1.2 Inputs

The flat list of inputs is summarized in the table below

| name | type | units | range | purpose |
|---|---|---|---|---|
| **Phase_Button** | bool | | | Rising edge means switch to next phase |
| **Mode_Switch** | bool | | | True = Manual stick command mode |
| **Pitch_Cmd** | real | degrees | -5 to +20 | Pitch Stick command |
| **Throttle_Command** | real | | 0 to 1 | Manual throttle command |
| **Speed_Setpoint** | real | Knots | 120-180 | Speed setpoint<br>*Note: the hardware driver ensures that this data is updated only when the knob is pressed.* |

[1] The SafeAir project (IST 1999-10913) was partly funded by the European Community.

| name | type | units | range | purpose |
|---|---|---|---|---|
| **Speed_Meas** | real | Knots | 0-200 | A/C velocity |
| **Alt_Setpoint** | real | feet | 1000 to 40 000 | altitude setpoint<br>*Note: the hardware driver ensures that this data is updated only when the knob is pressed.* |
| **Alt_Meas** | real | feet | 0 to 50 000 | A/C altitude |

The following inputs are grouped into structures:

**Sensors** is structured data composed of:

- Speed_Meas: measured speed.
- Alt_Meas: measured altitude.

**Pilot_Commands** is structured data composed of:

- Pitch_Cmd (from the pitch stick).
- Man_throttle Throttle_Command from the pilot throttle stick.
- Speed_Setpoint from the FCP knob.
- Alt_Setpoint from the FCP knob.

#### 4.3.1.3 Outputs

The flat list of outputs is summarized in the table below.

| name | type | units | range | purpose |
|---|---|---|---|---|
| **Elevator_cmd** | real | degrees | -20 to +20 | Commanding the elevator actuator |
| **Throttle_cmd** | real | | 0 to 1 | Commanding the throttle actuator |
| **Alt_Lights** | enum | | Green(0), amber(1), red(2) | Indicate altitude threshold |
| **Spd_Lights** | enum | | Green(0), amber(1), red(2) | Indicate speed threshold |
| **Phase_Lights** | enum | | PARK, T_OFF, M_CRS, LD | Indicate flight phase |
| **Mode_Light** | bool | | | True= AUTO |

The structure "**Status_Lights**" groups the following data:

- Phase_Lights.
- Mode_Lights.
- Alt_Lights.
- Speed_Lights.

### 4.3.2 High and Low-Level Requirements Specification

The starting point for our example are the systems requirements allocated to software ($SR_1$, …, $SR_n$). This includes the functional requirements of the software, its performance requirements and safety-related requirements.

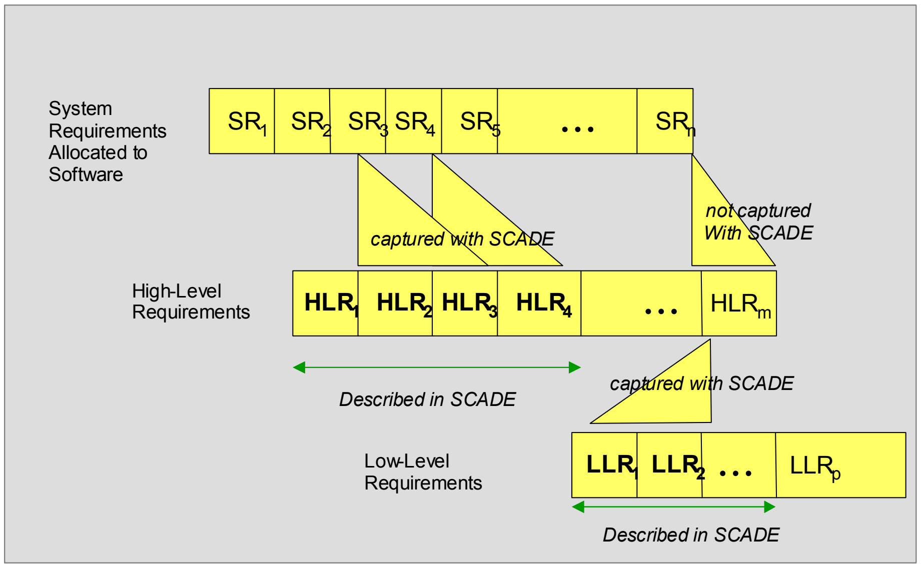


**Figure 11 From System to High and Low-level Requirements**

Within the **requirements process**, we may have manual formalization from some of the system requirements to high-level requirements described in SCADE. Other high-level requirements are not described in SCADE and are still described in a natural language or some other notation.

More precisely, a system requirement ($SR_i$) may be formalized into several high-level requirements ($HLR_i$), as is the case with $SR_3$ that is refined into $HLR_2$ and $HLR_3$. Furthermore, a given high-level requirement may participate in several system requirements, as is the case of $HLR_3$ with $SR_3$ and $SR_4$. Finally, there may also exist some "derived" high-level requirements that are not directly obtained by formalization of a system requirement, but may occur due to implementation or safety constraints.

Within the **design process**, we will only have to consider those high-level requirements that were not described in SCADE. For those, we may have a manual translation from high-level requirements to low-level requirements expressed in SCADE. Some of the low-level requirements may still be expressed in the form of pseudo-code or any other kind of low-level description, as it would be typically the case for low-level executive functions close to hardware.

The documentation of the modules, their interfaces, data types and flows can be generated and inserted in the Software Requirements Document.

#### 4.3.2.1 SCADE Artifacts for the High-level Requirements of the ACS

In this example, the SCADE model developed during the requirements phase serves as a means of formalizing the main structures of the software and is a part of the software high-level requirements document. The rest of high-level requirements would be text and figures.

At this stage, the SCADE model is an incomplete model. It will be complemented during the design phase.

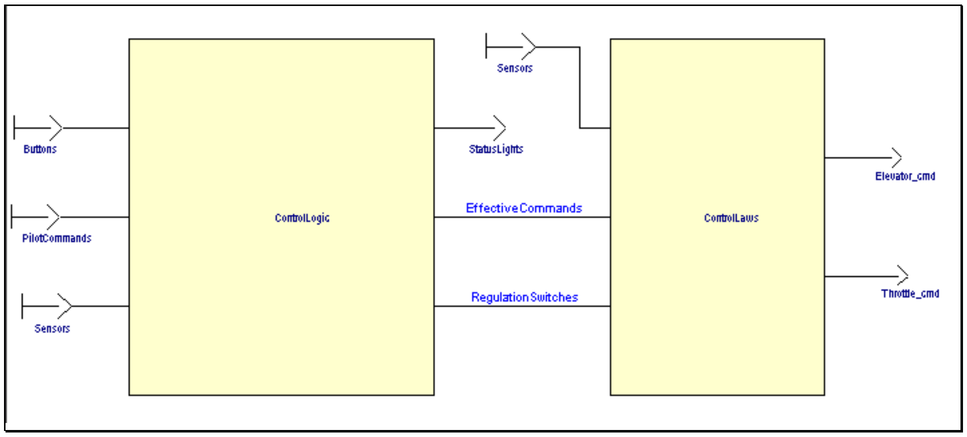
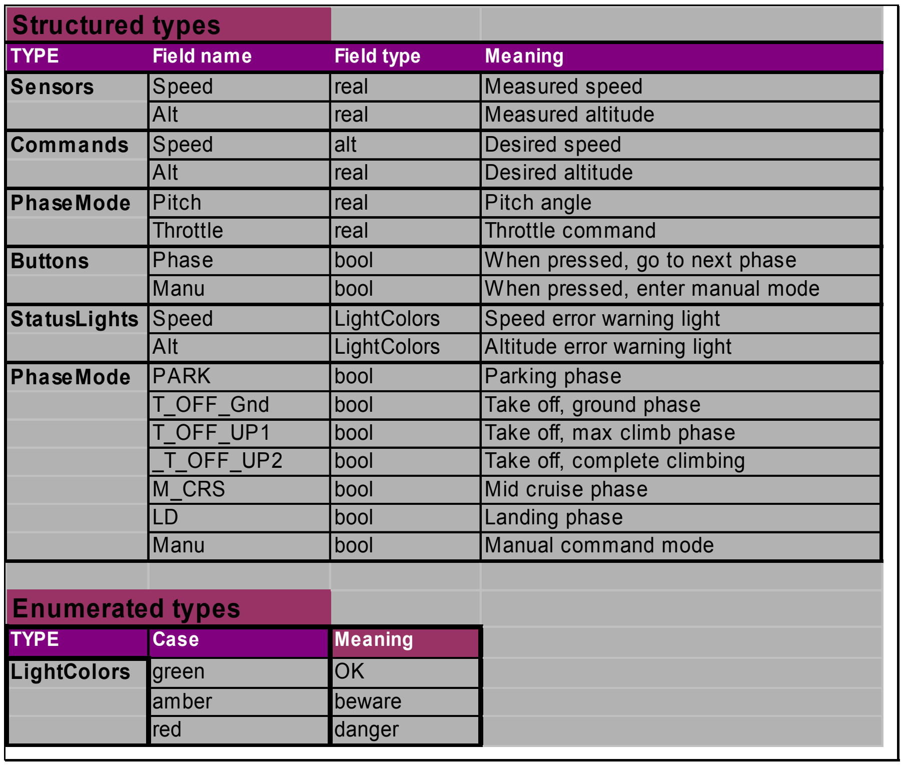


**Figure 12 Top level functions and interfaces of the ACS**

It formalizes the top-level functions, the interfaces of the system and of its high-level functions, and the data flow between those functions. Note that these flows are strongly typed and structured, using the data types definitions given below. This will provide a clean framework for the design.



**Structured types**

| TYPE | Field name | Field type | Meaning |
|---|---|---|---|
| **Sensors** | Speed | real | Measured speed |
| | Alt | real | Measured altitude |
| **Commands** | Speed | alt | Desired speed |
| | Alt | real | Desired altitude |
| **PhaseMode** | Pitch | real | Pitch angle |
| | Throttle | real | Throttle command |
| **Buttons** | Phase | bool | When pressed, go to next phase |
| | Manu | bool | When pressed, enter manual mode |
| **StatusLights** | Speed | LightColors | Speed error warning light |
| | Alt | LightColors | Altitude error warning light |
| **PhaseMode** | PARK | bool | Parking phase |
| | T_OFF_Gnd | bool | Take off, ground phase |
| | T_OFF_UP1 | bool | Take off, max climb phase |
| | _T_OFF_UP2 | bool | Take off, complete climbing |
| | M_CRS | bool | Mid cruise phase |
| | LD | bool | Landing phase |
| | Manu | bool | Manual command mode |

**Enumerated types**

| TYPE | Case | Meaning |
|---|---|---|
| **LightColors** | green | OK |
| | amber | beware |
| | red | danger |

**Figure 13 Structured data types**

State machines are used to explicit the states of the system and how it reacts to events. In the example below, this corresponds to parking, take off (on ground, 1st phase, 2nd phase), mid cruise, and landing.

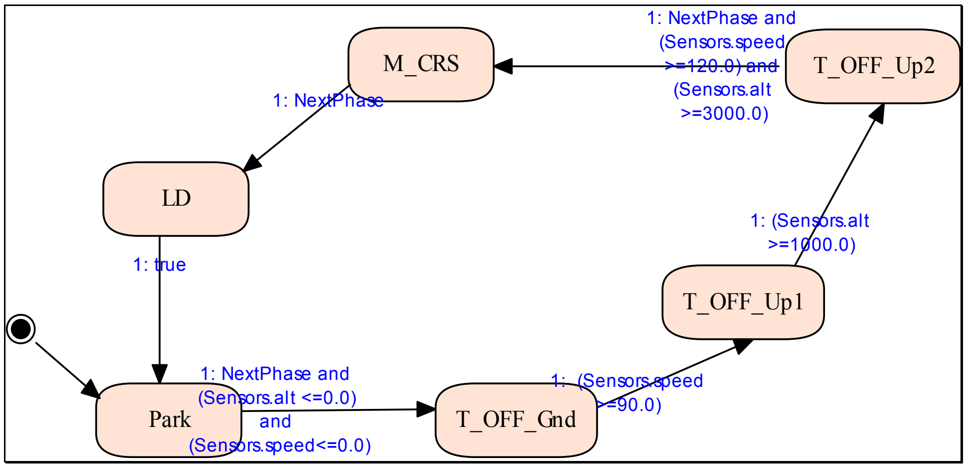

M_CRS
T_OFF_Up2
LD
T_OFF_Up1
Park
T_OFF_Gnd
1: NextPhase and (Sensors.speed >=120.0) and (Sensors.alt >=3000.0)
1: NextPhase
1: true
1: (Sensors.alt >=1000.0)
1: NextPhase and (Sensors.alt <=0.0) and (Sensors.speed<=0.0)
1: (Sensors.speed >=90.0)

**Figure 14 Main states of the ACS**

### 4.3.2.2 SCADE Artifacts for the Error Display

It is possible to directly translate some of the system functions into SCADE. For instance, the following system requirement defines the color of a light as a function of the altitude error (difference between measured altitude and set-point):

| \|ALT_error\| | Measured altitude background display |
|---|---|
| More than 1000 ft | RED |
| Between 100 ft and 1000 ft | AMBER |
| Less than 100 ft | GREEN |

**Figure 15 Altitude error color code display**

The following simple diagram, which factors two similar requirements, could directly express this function: one for the altitude, the other for the speed. Note that is typically a matter of project/company culture: some would prefer to do that during requirements, others during design.

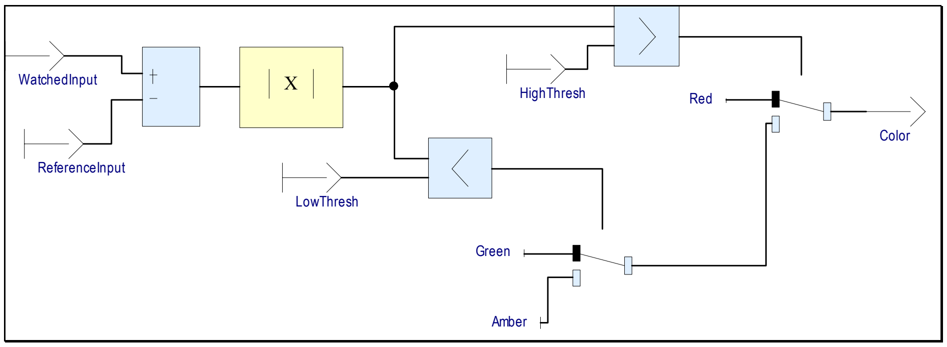


**Figure 16 Error color code computation module**

### 4.3.2.3 SCADE Artifacts for the Auto Mode Speed Definition

Here is a case where a two-step formalization is shown.

We had the following system requirement

- SYSREQ 21: «… automatic (AUTO) operational mode in which the A/C keeps its current speed (the value measured and identified upon entrance to this mode) ».

This sentence is rather complex, and we first decompose it in the HLR document:

- SWHLR 61: Memorize speed measure upon entrance in AUTO mode as *Mspeed.*
- SWHLR 62: Maintain A/C speed at *MSpeed* during Auto mode.

By the way, we realize that it would be dangerous to memorize a value that is not a correct for a set point. So, we add a derived requirement, which will be fed back to the system design.

- SWHLR 63: limit the speed value to memorize in the range [120,180].

Then, during the design activity, we formalize SWHLR61 and SWHLR63 in the following SCADE diagram. The speed from the sensors is first limited in the range 120 to 180, and this value is ready for memorization. The memorization (MEM block) occurs when the A/C enters (rising edge) the Auto mode.

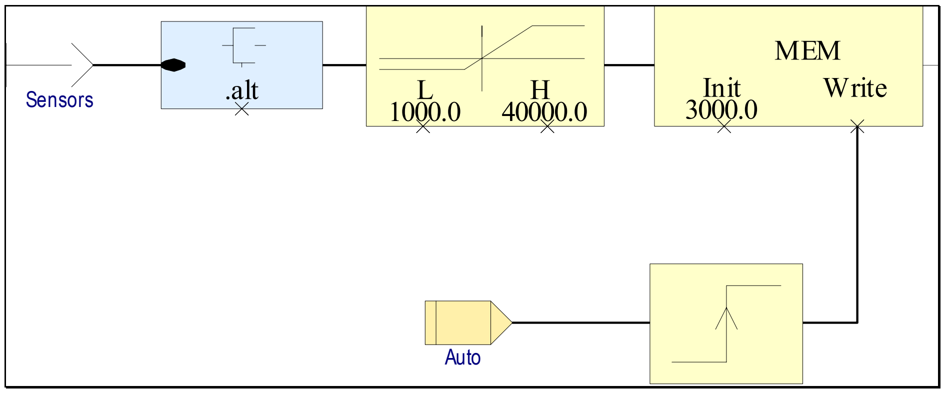


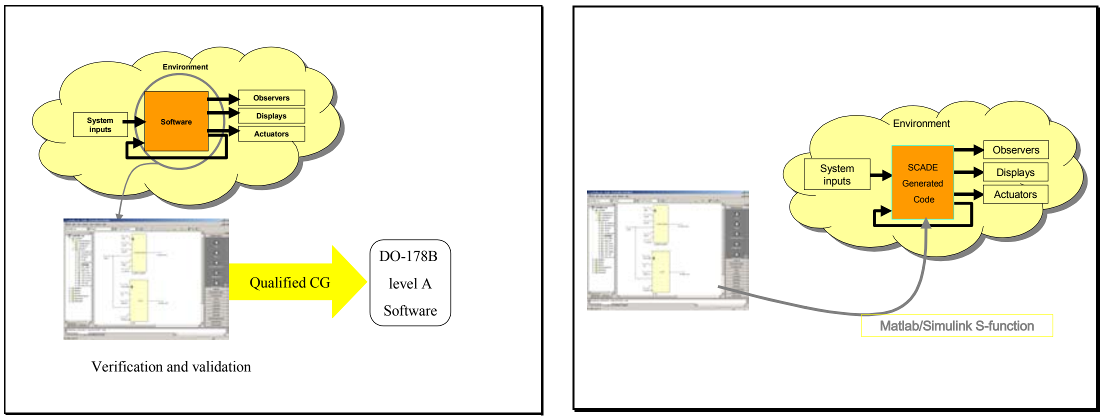
**Figure 17 Capturing the speed set point for auto mode**

#### 4.3.2.4 Filling the Gap between System Design and Software

For systems containing regulation and/or filtering functions, a model of the system dynamic behavior is often developed during the system design phase. This model usually contains 2 parts: the controller (ex: an engine speed regulator) and its environment (ex: the engine). The equipment implements the controller, and its software implements the functional behavior of this controller. Thus, the model of the controller is a requirement specification of the software.

When a tool such as MATLAB/Simulink$^{TM}$ is used to describe the controller, its transformation into a SCADE model is natural, since they are both based on data flow block diagrams. The Simulink Gateway even automates this task to a large extent. This gateway saves rewriting the description of the controller when going to software development.

Going from Simulink to SCADE improves the accuracy of the description to the level required by ED-12/DO-178B. In particular, data types and clocks have to be made more accurate. SCADE also gives access to formal verification.



*From Simulink to SCADE and to software development* *Validation of the software in a model of its environment*

**Figure 18 From system analysis in Simulink™ to software with the SCADE Simulink™ Gateway**

Moreover, the user can benefit from the qualified Code Generator:

- The generated code can be executed in the Simulink model of its environment, to validate its behavior in a model of its environment.
- The generated code can be downloaded to a prototype of the target, or the final target possibly with level A quality objectives, as a result of the qualified code generator (see section on code generation).

Below is a simple example with a piece of the control loop of the ACS.

Control engineers have provided the definition of a control loop in the form of control engineering block diagrams (Figure 19).

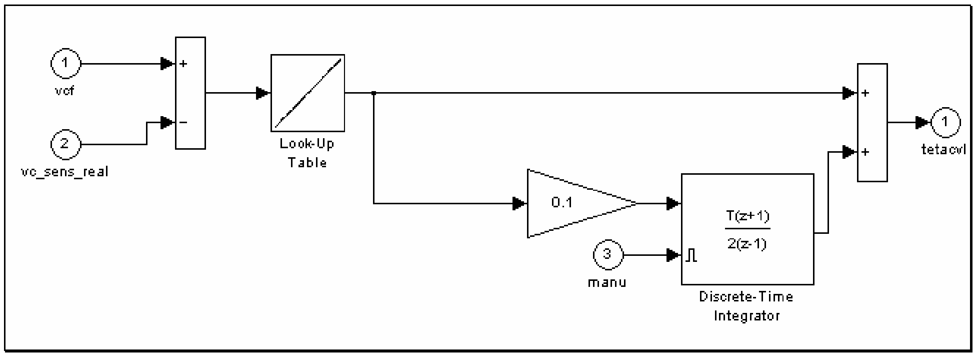


**Figure 19 Original control law given by control engineers**

The translation of the control engineering diagrams to SCADE (Figure 20) is quite easy, either manually for small models, or by using the Simulink to SCADE gateway for large models. When formalizing in SCADE, it may be necessary to specify details, such as initial conditions to remove any remaining ambiguity.

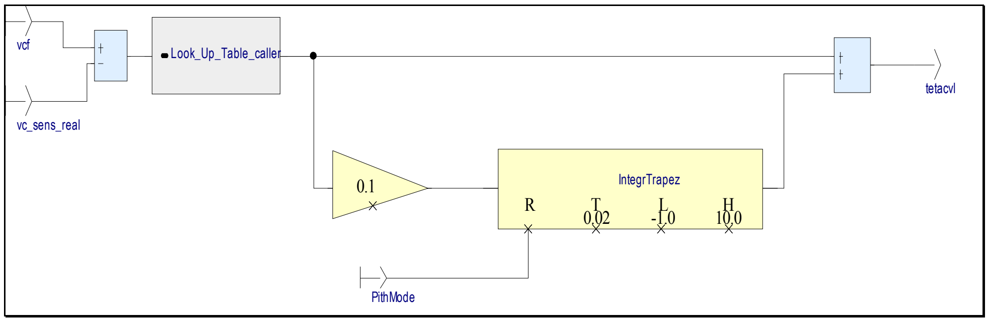


**Figure 20 Control loop element in SCADE**

#### 4.3.2.5 Benefits of the "Do Things Once" Principle

The SCADE model formalizes a significant part of the (high and low-levels) requirements. It is written and maintained once in the project, and shared among team members. Expensive and error prone rewriting is thus avoided, and interpretation errors are minimized. All members of the project team, from the specification team to the review and testing team will share this reference.

This formal definition can even be used as a contractual requirements 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.

#### 4.3.2.6 Teamwork and Reuse

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 clean distinction between interfaces and contents of modules (called "nodes" in SCADE) and there are no side effects from one node to another. So, a large software model can be split into several smaller SCADE models, with well-defined interfaces, thus forming a structured framework for the project. Each team member can work on a specific part. At each step, he can verify in a mouse click that he remains consistent with that interface. Later, integration of those parts in a larger model can be achieved by linking these "projects" in 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 the SCCI interface (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.

### 4.3.3 Coding and Integration

The SCADE Code Generators automatically generate the complete C or ADA code implementing the requirements and architecture defined in SCADE. It is not just a generation of skeletons: the complete dynamic behavior is implemented.

Various code generation options can be used to tune the generated code to the users needs; for example:

1) Generate one C function for a node or inline code of that node.
2) Pass external structured data by copy or by reference.

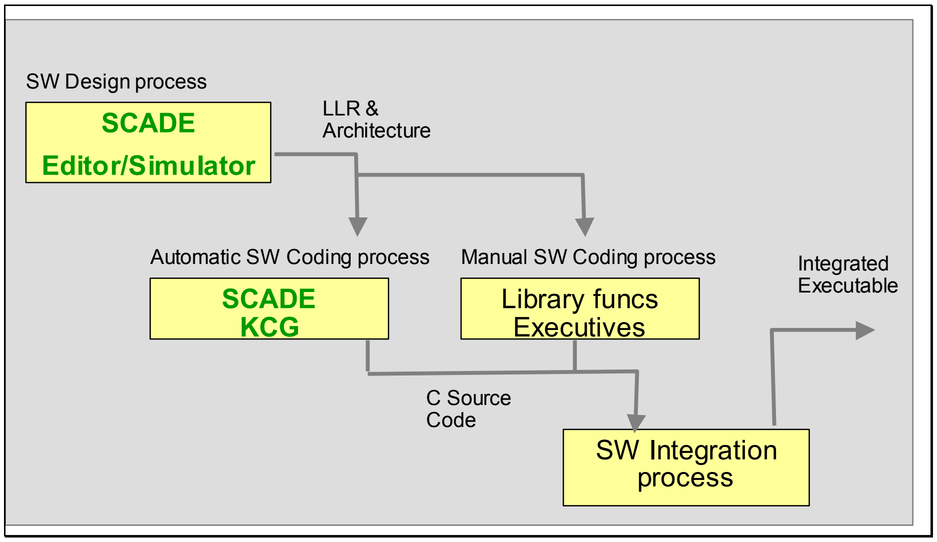


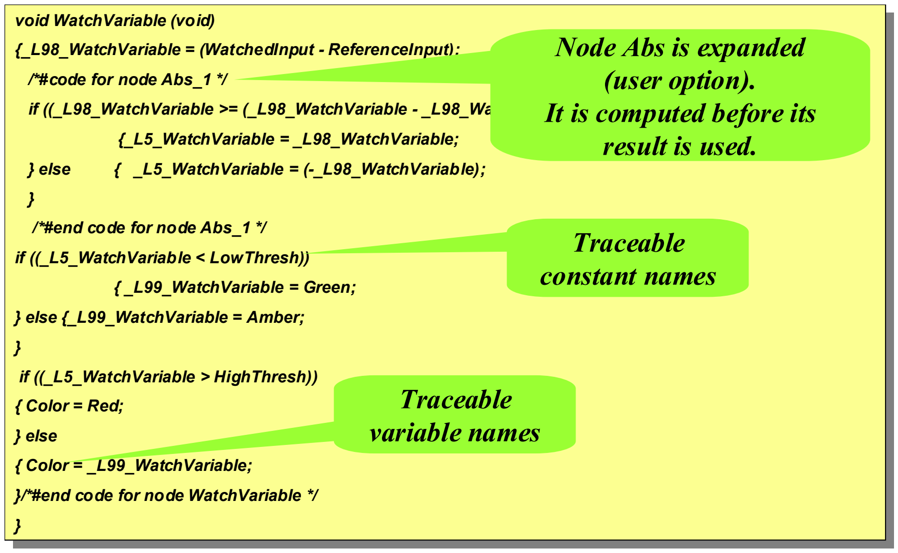
**Figure 21 Coding and integration**

**Legacy code**

Legacy code can be integrated easily as imported nodes and imported data.

**Scheduling**

The only scheduling code that the user has to write is the periodic call to the SCADE root function, typically based on the real time clock. The code generator, based on the data flow, automatically computes all of the internal scheduling of the model. It is a deterministic, sequential scheduling of the code. There is no overhead due to scheduling and communication, which one would have if the model pieces were implemented as tasks managed by an operating system. The generated code is both deterministic and efficient.



```
void WatchVariable (void)
{_L98_WatchVariable = (WatchedInput - ReferenceInput);
  /*#code for node Abs_1 */
  if ((_L98_WatchVariable >= (_L98_WatchVariable - _L98_Wa
                {_L5_WatchVariable = _L98_WatchVariable;
  } else       {  _L5_WatchVariable = (-_L98_WatchVariable);
  }
   /*#end code for node Abs_1 */
if ((_L5_WatchVariable < LowThresh))
                { _L99_WatchVariable = Green;
} else {_L99_WatchVariable = Amber;
}
 if ((_L5_WatchVariable > HighThresh))
{ Color = Red;
} else
{ Color = _L99_WatchVariable;
}/*#end code for node WatchVariable */
}
```

Node Abs is expanded (user option). It is computed before its result is used.
Traceable constant names
Traceable variable names

**Figure 22 Sample generated code**

**Safety**

The generated code is safe: there is no pointer arithmetic, no dynamic memory allocation, no operating system call; the only loops, which are for delays or array handling have a fixed length.

The generated code is traceable to the model: nodes, variables, constants are traceable by name and/or comments as shown on Figure 22.

## 4.4 The Verification Processes with SCADE Suite

*In this section we describe the verification activities that are performed when using SCADE Suite, without detailing the simplified/suppressed verification activities. Appendix C explains how this conforms to DO-178B. System level verification/validation activities such as stability of a control law are beyond the scope of software validation and of this paper.*

### 4.4.1 Verification of Requirements Consistency

First, one must check the consistency of the requirements. The syntactic and semantic checker of SCADE Suite performs 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", i.e. of production/consumption rates of data.

It is also possible to add custom verification rules, using the programmable interface of the SCADE editor.

### 4.4.2 Validation of Requirements with respect to System Requirements

The SCADE requirements have to be reviewed for conformance with system requirements. The SCADE Suite report generator ensures that the documentation is up-to-date. The advanced "find" feature of the editor facilitates an in-depth review.

A requirements management tool such as DOORS may also help in managing traceability with other life cycle data, such as textual requirements or test cases. Since SCADE and DOORS are integrated, the traceability of SCADE with textual requirements is simple and efficient.

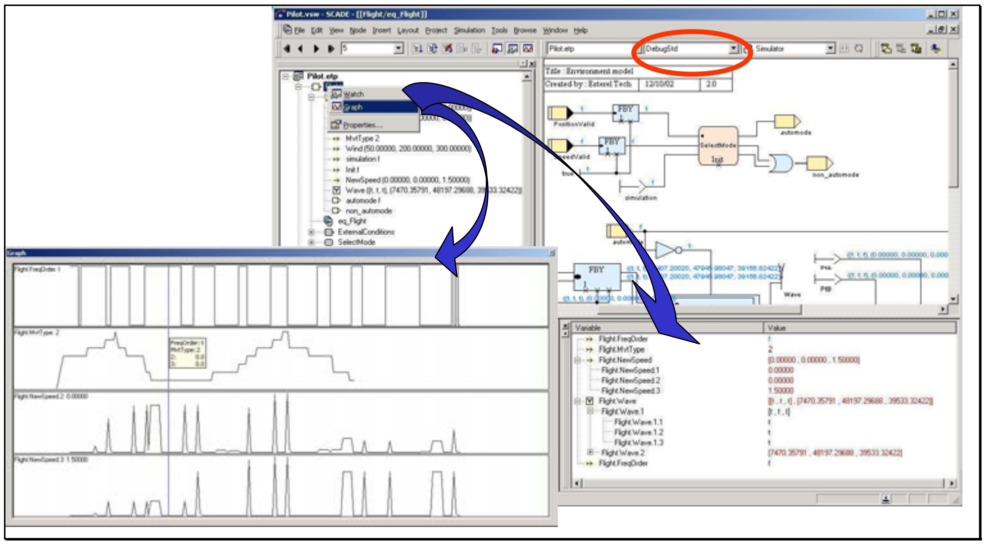


**Figure 23 Simulation allows to "play the requirements"**

It is also helpful to exercise dynamically the behavior of that specification, to better understand how it behaves. As soon as a SCADE model (or pieces of it) is available, it can be simulated with the SCADE simulator. Simulation can be interactive or 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 system requirements.

### 4.4.3 Verification of Robustness

Robustness and safety must be addressed at each level, as explained by ARP 4754. We recommend that robustness be addressed differently at the requirements and coding level.

At the requirements/design level, the specification should explicitly identify and manage robustness of the software with respect to invalid input data. This requires techniques such as voting, confirmation and range checking. At the requirements level, one should explicitly manage the ranges of variables. For instance, one should use generally integrators with a limiter. Or, if there is a division, the case where the divider is zero has to be managed explicitly at the requirements level, in the context calling the division: the division should only be called when the divider is not zero, and the action to be taken when the divider is zero in a foreseeable situation has to be defined by the writer of the specification, not by the programmer.

On the contrary, if an attempt to divide by zero happens at run time in spite of the above-mentioned design principles, this has to be handled as an abnormal situation, caused by a defect in the software specification, or by a hardware failure. The detection of the event can be typically part of the arithmetic library (the implementation of that library is generally target dependant). The action to be taken (ex: raise an exception and disconnect the computer) has to be defined as a global design decision for the computer.

It is easy to define libraries of robust blocks, such as 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 (in particular fixed point, if the application uses this technique) on the host and on the target, with libraries that have the same behavior.

Requirements analyses and reviews have to include the above-mentioned rules. These rules ensure that robustness is effectively managed in the software specifications and in the libraries, rather than being spread over the code.

### 4.4.4 Verification of the Source Code

The code generator is a "development tool", in DO-178B terminology. That means that a failure in the code generator may introduce an error in the code that will be embedded.

As explained in appendix C, the qualification of a tool may reduce the requirement around the verification of its output. There are two ways of using a SCADE Code Generator for the development of software, and presenting it accordingly to the certification authority:

a) **Unqualified**: the code generator is just a way of writing the code more effectively. But no reduction of verification is possible. The code is verified as if it was written manually. That means among others reading the code and ensuring its structural coverage during testing.

b) **Qualified**: the qualification of the SCADE code generator may save or eliminate low-level testing and structural coverage.

Appendix C provides details about the savings on verification activities, and appendix D give details about the qualification of the code generator.

### 4.4.5 Software and Hardware Integration Testing

ED-12/DO-178B mandates "*Requirements-Based Software Integration Testing*."

Using the SCADE simulator and Design Verifier for SCADE (from Prover Technologies) for the verification of the LLR produces large amounts of requirements-based test scenarios. These scenarios are first class material for requirements-based testing on the target, since:

- They are requirements based.
- They are consistent with the LLR (by construction).
- They save rewriting similar test scenarios.

*Note: the scenarios produced by the SCADE simulator and verifier are simple ASCII files which can be input to most test environments using simple scripts.*

### 4.4.6 The Combined Testing Process

In practice, we propose to organize the testing process in the following way (Figure 24):

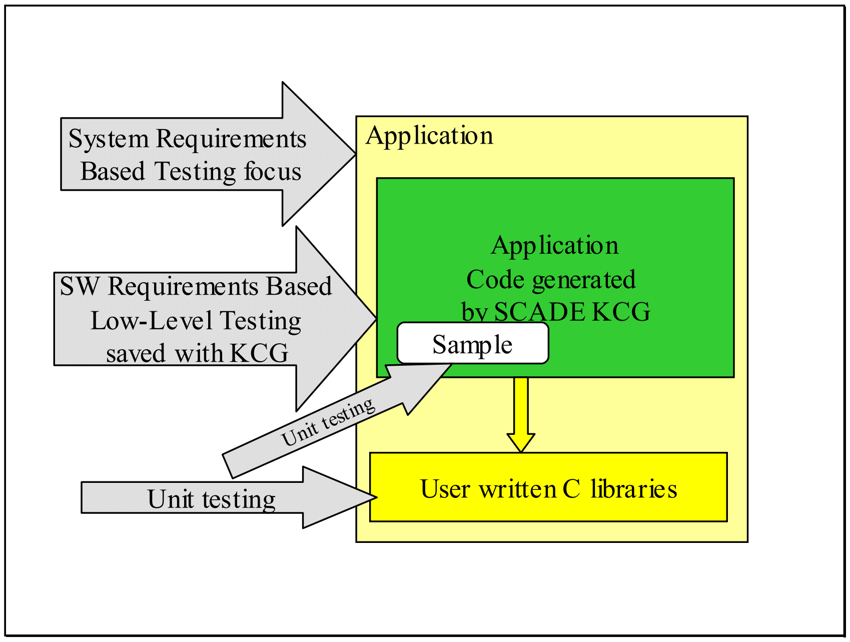

System Requirements Based Testing focus
SW Requirements Based Low-Level Testing saved with KCG
Unit testing
Unit testing
Application
Application Code generated by SCADE KCG
Sample
User written C libraries

**Figure 24 The combined testing process with KCG**

1) For the functions that are hand-coded in the Source Code language (e.g., library functions and/or executives):

- The user performs classical verification activities (including low-level testing and structural coverage analysis at the Source Code level).
- The Source to Object Code compiler is used in the same version and with the same options (no optimization) and in the same execution environment as when it is used to compile Source Code obtained from the KCG.
- Analysis of this Object Code is performed according to CAST Paper P-12 [CAST-12] to demonstrate that the object that is not directly traceable to source code is correct.

2) For the Source Code automatically generated by KCG:

- The user performs testing activities of a sample of the generated Source Code that comprises all used Source Code programming constructs in order to demonstrate that the Object Code generated from this Source Code is correct and does not introduce erroneous extra code that is not traceable at the Source Code level (as in CAST Paper P-12).

3) For the whole application:

- The user performs extensive systems requirements-based software and hardware/software integration testing. It is verified at this stage that all systems requirements allocated to software are covered by those tests.
- The stopping criterion is to achieve both coverage of the system requirements and the structural coverage of the model (see appendix C).

We acknowledge that, by specification, KCG uses only a small subset of the general purpose Source Code language, with a low-level of complexity (mostly expressions with comparisons, +; - etc) and generates very regular code structures. If the combination of all the above activities does not detect any error in the object code, then we can have sufficient confidence that the compiler does not introduce errors in the code generated by KCG for that application.

# 5 Examples of productivity enhancements

***This chapter shows results obtained by applying the process improvement on industrial projects.***

## 5.1 Eurocopter EC 135/155

Eurocopter (EC) is the world leader in civil helicopters and a large manufacturer of military helicopters. With four main sites and thirteen subsidiaries around the world, EC has delivered more than 11,000 aircraft to 1,700 clients in 132 countries.

Eurocopter introduced SCADE for the development of autopilots for the EC135 and EC155 civil helicopters, done in co-operation with the equipment manufacturer SFIM. These equipments must adhere to DO-178B guidelines for Level A. Eurocopter's primary challenge was to reduce development time, certification time, and costs.

To guarantee coherence in the development of the product range, this collaboration required precise formal specifications. By defining common rules for naming and structuring, SCADE made it possible to introduce, from the specification phase, detailed and complete information allowing unambiguous communication between the Eurocopter and its subcontractors. Eurocopter developed and integrated the EC155 autopilot operational functions while SFIM developed the equipment management functions. Both Eurocopter and SFIM sites used SCADE as a specification and code generation tool. A key benefit of this technique is that it allows simulation on a host machine before integration into the target computer, specifications that are better validated and more complete.

The result is

- 90% of the code could be generated with SCADE.
- The development time was reduced by 50%, compared to manual coding of an equivalent system.
- JAA certified the equipment at level A.

## 5.2 Airbus A340

Airbus introduced automatic code generation in the late 80's with in house tools, and there is a proven success using this approach with the A340.

Airbus participated in defining SCADE and introduced SCADE as a successor to their similar in house tools. Today, SCADE models are written and exchanged throughout the company and they are part of the technical annex of contracts with equipment manufacturers.

The SCADE KCG Code Generator has been used for the development of the software of the FCSC (Flight Control Secondary Computer) of the A340/500. This equipment must comply to DO-178B Level A guidelines. The result is the following:

- The ratio of automatically generated code reached 70%.
- No coding errors were found in the code generated by SCADE
- Specification changes were perfectly mastered and the modified code was quickly made available, therefore reducing time-to-market.
- The SCADE KCG Code Generator has passed qualification procedures in January 2002.
- Airbus measurements showed a reduction of 50% in development cost, and a reduction in modification cycle time by a factor 3 to 4, compared to manual coding, by using their in-house code generator [Pilarski]. Now, KCG has shown the same efficiency and will be used in other programs, such as the A380.
- SCADE is also expected to bring further benefits in the future, such as formal verification.

# 6 Business benefits

*This chapter analyzes the business benefits of the above described process improvement.*

### 6.1.1 Addressing the Bottlenecks of Traditional Development

SCADE Suite addresses the bottlenecks of traditional development:

- ❑ Multiple descriptions: they are replaced by the common model, which is refined and shared among project members.
- ❑ Ambiguity and lack of accuracy: being both formal and intuitive, a SCADE model prevents interpretation errors.
- ❑ Manual coding: is replaced by automatic coding.
- ❑ Late detection of requirements/design error: the SCADE model (or pieces of it) can be readily simulated.
- ❑ Complexity of updates: only the model is updated; documentation and code can be updated automatically.

### 6.1.2 Time to Market and Cost Reduction with the "Y" Cycle

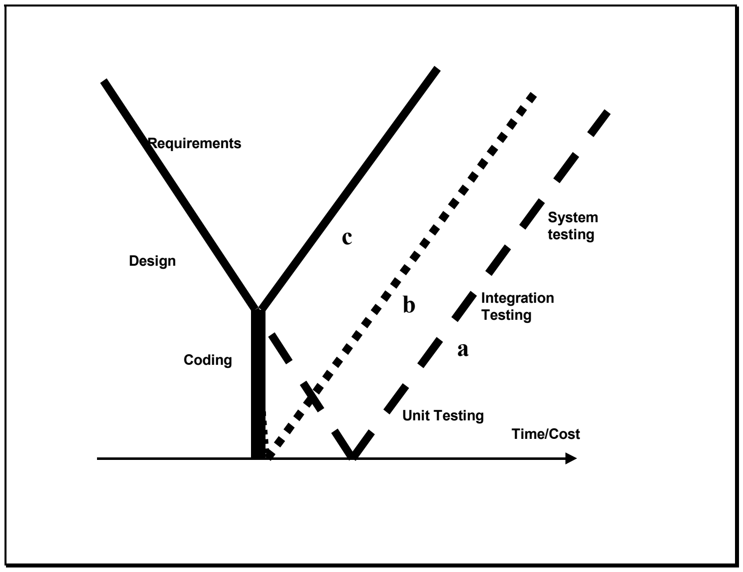


**Figure 25 From the V cycle to the Y cycle**

If we take the traditional "V cycle" as a reference, Figure 25 summarizes the time and cost savings, depending on the lifecycle:

- Lifecycle a: Traditional development.
- Lifecycle b: The Code Generator is just used as a productivity tool; it saves the tasks of writing the code, and the cost reduction is about 15% if there are no LLR changes at all, and 20% if some changes are made. All verification activities have to be performed.
- Lifecycle c: The Code Generator is qualified as a development tool; compliance of the code with the low-level requirements is guaranteed and the corresponding verification activities are saved. If the risk of undetected error introduction by the compiler is mastered, the compliance of the executable with the low-level requirements is guaranteed. We then shift from the V cycle to the "Y cycle", meaning that the cost of producing the object code and verifying its conformance to the requirements is near to zero.

Introduction of proof techniques might lead to 10% more savings, as estimated by Airbus projections [Pilarski] . This has to be further confirmed at a large scale.

As an example, the chart below shows potential costs reductions on a typical project. The figures concern the part of the application for which SCADE is applicable. Although the effort of writing the requirements and design can be reduced (for the same level of detail than in a traditional document), we assume that this is compensated by a more detailed description and more preparation of test cases at simulation stage. The effort for coding itself is almost completely saved. The integration effort is decreased thanks to the higher consistency of the SCADE model and of its generated code. When SCADE is used, the verification cost is decreased by 20%, due primarily to the verification of the design by the SCADE checking tools. When using KCG, there are major supplementary savings: one is the reduction of the "normal" verification, the other is due to the savings in the verification of code when correcting a requirement and reflecting this change in the implementation.

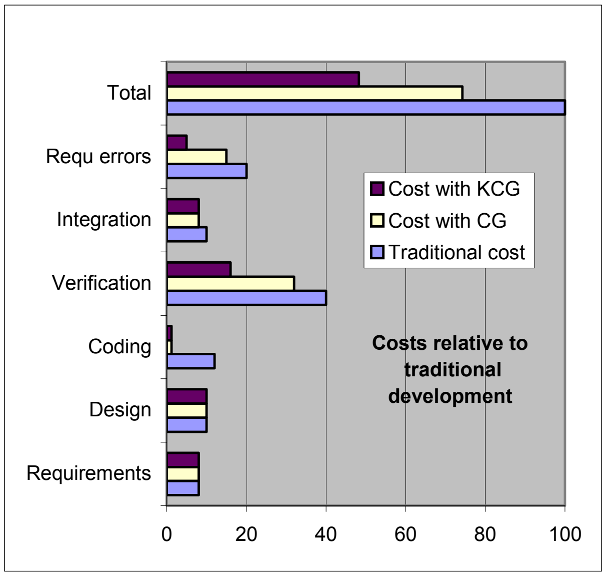


**Figure 26 Cost reduction with CG and KCG**

The effect of the Y cycle is not only to reduce direct costs. Reducing the development time leads to earlier availability of the product. This is clearly a competitive advantage for the company with the first version of an aircraft or equipment on the market.

This is even truer with new versions: a company has often to build new versions/variants of an equipment to adapt to various requirements defined by the customers. The Y cycle dramatically decreases the time to build the variants. This may favor the selection of the company offering such a shorter response time.

### 6.1.3 Higher Reuse Potential

In the traditional approach, the project produces on one hand a textual description and on the other software code. The code is low-level, hard to read and hard to maintain, and often target dependant, even if good coding rules have been applied. When new equipment has to be developed, it is hard to reuse the old software.

A SCADE software model is much more functional and target independent. Experience has shown that large amounts of SCADE blocks could be reused from one project to another. Libraries of functional blocks could be defined in one project and reused in another one. The software could be implemented on new targets by just adapting the bodies of the bottom-level library elements, without changing the SCADE models.

# 7 Appendix 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).

[CAST-12] *“Guidelines for Approving Source Code to Object Code Traceability*”, CAST-12 Position Paper, December 2002

[Chilenski] *“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, Oct 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).

[N81110.91] “*Guidelines for the Qualification of Software Tools using RTCA/DO-178B*”, FAA Notice, N81110.91, January 16th, 2001

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

[SCADE_Lang] “*SCADE Language Reference Manual,”* Esterel Technologies 2003

# 8 Appendix B Acronyms and Glossary

## 8.1 Acronyms

**A/C:** Aircraft.

**COTS**: commercial off-the-shelf.

**DER**: Designated Engineering Representative.

**EUROCAE**: European Organization for Civil Aviation Equipment .

**FAA**: Federal Aviation Administration.

**HLR**: High-level Requirement.

**LLR**: Low-level Requirement.

**JAA**: Joint Aviation Authorities.

**JAR**: Joint Aviation Requirements.

**MC/DC**: Modified Condition/Decision Coverage.

**RTCA**: RTCA, Inc.

**SCADE**: Safety Critical Application Development Environment.

**SQA**: software quality assurance.

**SW**: software.

## 8.2 DO-178B Glossary

**Certification** - Legal recognition by the certification authority that a product, service, organization or 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) which by design is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component, or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options.

**Dead code** - Executable object code (or data) which, as a result of a design error cannot be executed (code) or used (data) in a 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, which assists the software development, processes and other integral processes and, therefore, remains active throughout the software life cycle. The integral processes are the software verification process, the software quality assurance process, the software configuration management process, and the certification liaison process.

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

**Modified Condition/Decision Coverage** - Every point of entry and exit in the program 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.

# 9 Appendix C Impact of SCADE Suite on Verification Activities

## 9.1 Scope and Conventions of this Section

This section provides some additional details about the impact of SCADE Suite on verification. In particular, it positions the savings of verification activities in terms of DO-178B, and explains why/how SCADE Suite can make certain verification activities easier or even eliminate some of them.

Verification activities are described in section 6 of DO-178B, and their objectives are summarized in tables A-3 to A-7. These tables are used in this appendix as reminders for the explanations. We also added a column characterizing the potential **impact** of the use of SCADE Suite on the verification activities, with the following conventions:

- Blank: Means no significant impact.
- Efficiency: Means a significant efficiency improvement of the activity.
- Eliminated: Means that the verification activity can be eliminated.
- Automated: Means that the verification activity can be automated.

***Note: the statements of the next sections apply only to the parts modeled with SCADE and generated from the SCADE model.***

## 9.2 Verification of Outputs of the Software Requirements Process

All SCADE elements will be part of one global model, whether these elements have been written during the requirements process or the design process.

We are in a case where Source Code is generated directly from the model. So, the model elements that have been developed as part of the high-level requirements are also considered low-level requirements, and the guidelines for low-level requirements also apply to them (DO-178B; Section 5.0).

Therefore, we refer the reader to the next section, where the verification of low-level requirements is described.

## 9.3 Verification of Outputs of the Software Design Process

| | Objective | Impact |
|---|---|---|
| 1 | Low-level requirements comply with high-level requirements | Efficiency |
| 2 | Low-level requirements are accurate and consistent | Efficiency |
| 3 | Low-level requirements are compatible with target computer | Efficiency |
| 4 | Low-level requirements are verifiable | Eliminated |
| 5 | Low-level requirements conform to standards | Automated |
| 6 | Low-levels requirements are traceable to high-level requirements | Efficiency |
| 7 | Algorithms are accurate | Efficiency |
| 8 | Software architecture is compatible with high-level requirements | Efficiency |
| 9 | Software architecture is consistent | Efficiency |
| 10 | Software architecture is compatible with target computer | Efficiency |
| 11 | Software architecture is verifiable | Efficiency |
| 12 | Software architecture conforms to standards | Automated |
| 13 | Software partitioning integrity is confirmed | |

**DO-178B Table A-4**

**Low-level requirements comply with high-level requirements (resp. system requirements)**

The very functional nature of the SCADE notation with block diagrams and state machines makes it easy to write SCADE models that reflect the higher level functional requirements. This makes the verification of compliance accordingly simple. The same holds for the relationship between system requirements and SCADE elements written during the software requirements phase.

As explained in section 4.5, simulation is a very efficient technique to validate the requirements.

Formal verification, which could be compared to a kind of exhaustive simulation with respect to the properties to be verified, is another means of verifying the compliance of a SCADE model with higher-level requirements

**Low-level requirements are accurate and consistent**

SCADE is based on Lustre [Lustre] and inherits its formal semantics [SCADE_Lang]. A SCADE model is absolutely accurate and unambiguous. It has the same meaning for all the project participants ranging from the specification team to the validation team, thus avoiding interpretation errors. A SCADE model is also strictly deterministic, which means that a given input sequence from the initial state will always lead to the same output sequence.

The syntactic and semantic checker of SCADE Suite performs 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”, i.e. computation and exchange rates between blocks.

**Low-level requirements are compatible with target computer**

The structure of a SCADE model and of its implementation model (the cycle based model) is compatible with nearly any computer. The main possible incompatibility concerns the Worst Case Execution Time (WCET). The generated code has bounded execution time. By downloading code generated from early versions of the model, measurements can be performed early, compared to estimates, and refined all along the model development.

**Low-level requirements are verifiable**

The SCADE language is formal. That means that every requirement expressed in SCADE has a precise, unambiguous meaning. For every such requirement, it is possible to answer in an objective way "yes" or "no" to the question: "Does this system fulfill this requirement?"

**Low-level requirements conform to standards**

The SCADE notation itself is a standard with precise syntactic and semantic rules. These rules are verified by the qualified code generator (a quick check is also available as part of the editor).

It is also possible to define and verify additional custom rules, with the editor.

**Low-levels requirements are traceable to high-level requirements (resp. system requirements)**

There are several, non exclusive ways of managing traceability:

- From text to SCADE: by using the same functional structure, with similar names for functions, states and data.
- By refining a SCADE model, from one phase to the other (see for example the ACS modeling of the top level functions).
- Using the annotations features of SCADE to reference higher level requirements from SCADE elements.
- Using the DOORS Gateway. This allows analyzing the coverage of the high-level requirements by the low-level requirements, and identifying those high-level requirements that are not covered. DOORS may also be useful to manage the coverage of requirements by test cases.

**Algorithms are accurate**

Regarding logical and temporal aspects, any SCADE model is accurate by construction.

Concerning numerical aspects, specific design and verification activities are required, as explained in section 4.4.3.

**Software architecture is compatible with high-level requirements (resp system requirements)**

The architecture of the generated code is simple. It is a root function calling others sequentially. There is no operating system call. The generated code can be included in one operating system task, or can even run on a bare computer, with just a real time clock and hardware interface interrupt service routines.

**Software architecture is consistent**

The architecture of the software is a tree of nodes, translated to a tree of C function. The code generator semantically checks the consistency of the input model and ensures that it is reflected in the code. Verifications include checking rates and order of production and use of data.

**Software architecture is compatible with target computer**

The architecture of the software is a tree of nodes, translated to a tree of C function calls. Simple user written code such as a real time clock handler calls the root of the tree periodically.

**Software architecture is verifiable**

The model describes the architecture in a formal, verifiable way: which node calls which others, when, what data are exchanged between nodes.

**Software architecture conforms to standards**

The software architecture is described with the standard SCADE notation. Conformance to the standard is verified by SCADE tools.

**Software partitioning integrity is confirmed**

SCADE code/data cannot harm other code/data, but there is nothing that can protect it from other software. Partitioning has to be managed with specific hardware or operating system mechanisms.

## 9.4 Verification of Outputs of the Software Coding and Integration Processes

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

**DO-178B Table A-5**

**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 coding standards. The qualification of the code generator ensures that this 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 software integration process is complete and correct**

No impact of SCADE on verification.

## 9.5 Testing of Outputs of Integration Process

| | Objective | Impact |
|---|---|---|
| 1 | Executable Object Code complies with high-level requirements | Efficiency |
| 2 | Executable Object Code is robust with high-level requirements | Efficiency |
| 3 | Executable Object Code complies with low-level requirements | Efficiency |
| 4 | Executable Object Code is robust with low-level requirements | Efficiency |
| 5 | Executable Object Code is compatible with target computer | Efficiency |

**DO-178B Table A-6**

***Note: see the combined testing process in chapter 4***

**Executable Object Code complies with high-level requirements**

During the simulation phase of the SCADE requirements, it is possible to play requirements based test cases, and to record the scenarios containing them. These scenarios can be reused during integration testing. The combined testing process allows focusing requirements based testing at the application level.

**Executable Object Code is robust with high-level requirements**

There are 2 parallel verification streams for robustness:

- By construction: See section 4.4.3 the design rules and the library.
- By testing ranges of the input data at the application level.

**Executable Object Code complies with low-level requirements**

During the simulation phase of the SCADE requirements, it is possible to play requirements based test cases, and to record those scenarios. These scenarios can be reused during integration testing.

The combined testing process allows focusing at the application level.

**Executable Object Code is robust with low-level requirements**

There are 2 parallel verification streams:

- By construction: See section 4.4.3 the design rules and the library.
- By testing ranges of the input data.

**Executable Object Code is compatible with target computer**

The generated code uses bounded resources.

Memory usage is limited to static memory and bounded stack. Memory occupation can be predicted from compiler/linker maps, and/or measured.

The Worst Case Execution Time evaluation technique depends on the processor. The simple structure of the generated code eases this verification.

## 9.6 Verification of Verification of Process Results

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

**DO-178B Table A-7**

**Test procedures are correct**

In the case where the simulator has been used for requirements-based simulation scenarios, there is a lot of material reusable for building test procedures (the simulation input/output files can easily be transferred to a test environment). There is a shift of the verification of correctness of test procedures to the verification of correctness of simulation scenarios, and the total effort over the lifecycle is unchanged.

**Test results are correct and discrepancies explained**

In the case where the simulator has been used for requirements-based simulation scenarios, there is a lot of material reusable for building test procedures. If the test environment is open, it is easy to retrieve both the inputs and the expected results.

**Test coverage of high-level requirements is achieved**

If the high-level requirements are textual, then the test cases will be based on those requirements. Simulation scenarios, covering the high-level requirements should have been built and exercised during the design process. These scenarios may be reused during integration testing.

Similarly, if the high-level requirements are in SCADE, then the reference for test cases should be the system requirements.

**Test coverage of low-level requirements is achieved**

This verification by itself is no longer relevant. It has to be replaced by coverage of high-level requirements or system requirements.

**Test coverage of software structure is achieved**

Section 6.4.4.3 of DO-178B states that structural coverage may reveal code structures that were not analyzed during testing that may be the result of:

- ❑ Shortcomings in requirements-based test cases or procedures.
- ❑ Inadequacies in software requirements.
- ❑ Dead code.
- ❑ Deactivated code.

We therefore have to examine these four cases:

- ❑ **Shortcomings in requirements-based test cases or procedures:** KCG being a qualified development tool, there is no reason to test the Source Code against the software requirements. This point is therefore irrelevant.

- **Inadequacies in software requirements:** There can be problems with the software requirement. The testing activity relative to the software requirements that we described in this paper will exhibit those problems. We have complemented the usual analyses and reviews that are performed at this level by a testing process. We therefore have to measure the coverage of the software requirements by the corresponding test suite, as explained below.
- **Dead Code** and **Deactivated Code:** KCG cannot introduce code that does not correspond to the requirements contained in the input model. The traditional risk of introducing unintended code during the coding process no longer exists. The risk of dead/deactivated/unintended software requirements still exists, but structural coverage of code is not the most suited way to identify them. We propose in the section on Structural coverage analysis of the software requirements a systematic way of detecting unintended requirements.

This analysis demonstrates that, provided the fact that we perform the testing activity of the software requirements and the corresponding model coverage analysis, structural coverage (MC/DC) of the software structure is no longer required when KCG is used.

**Test coverage of software structure (data coupling and control coupling) is achieved**

DO-178B requires that test coverage of the software structure (data and control coupling) is achieved and it defines:

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

Control coupling – as "The manner or degree by which one software component influences the execution of another software component."

Data coupling

If we first consider the question of data coupling, it appears that it is a completion check of the integration effort that includes verifying:

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

These verification activities are managed at a higher level. As an example, we are in a situation where the interfaces between modules of the software structure are already defined at the model level and therefore the completion check may again be done at model level, using the SCADE semantic checker. As another example, we may say that project standards enforce rules at model level such that global data is not allowed, thus producing Source Code that has no global data.

Control coupling

If we now consider the question of control coupling, it appears as a completion check of the integration effort that includes verifying:

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

As an example, we are in a situation where the call sequences of the software structure maps the ones of the model representing the software requirements that are expressed in the SCADE graphical notation and therefore the completion check may again be done at model level, using the SCADE semantic checker.

## 9.7 Structural coverage analysis of the software requirements (model coverage)

Unintended code cannot be produced by KCG, from intended software requirements. The remaining point is the detection of dead/deactivated/unintended software requirements with respect to system requirements. Contrarily to dead code, a “dead requirement” is most often not a piece of text that has not been purged. It is more frequently a wanted feature, inhibited by a complex chain of dependencies that the requirements writers did not identify. Traditionally, this identification could only be achieved by human analysis of the software requirements, where it was difficult to analyze all possible dynamic situations. With SCADE Suite, it is possible to run system requirements based test cases of the software requirements by using the SCADE simulator. We therefore propose to evaluate the coverage of the software requirements in a measurable way.

The definition of requirements coverage criteria is an emerging topic. The objective is to answer the question: “did I exercise every piece of the software requirements during system requirements based testing?” (Simulation is considered here to be part of testing).

As a first example, with block diagrams, the criteria could be the activation of nodes, of selectors, and of characteristic inputs/outputs responses for library operators, the definition of these criteria being part of the library definition. For instance the confirmation of an input is one of the characteristic events for a confirmation node.

With state machines, one could typically use state and transition coverage criteria.

As another example, in the case we are dealing with a block diagram notation, it is possible to transpose the classical MC/DC criterion that normally applies to Source Code to these block diagrams, as is explained in the tutorial on MC/DC [Chilenski]. This is just one possible approach, and it has not been demonstrated that the same criteria should be used for requirements than for source code.

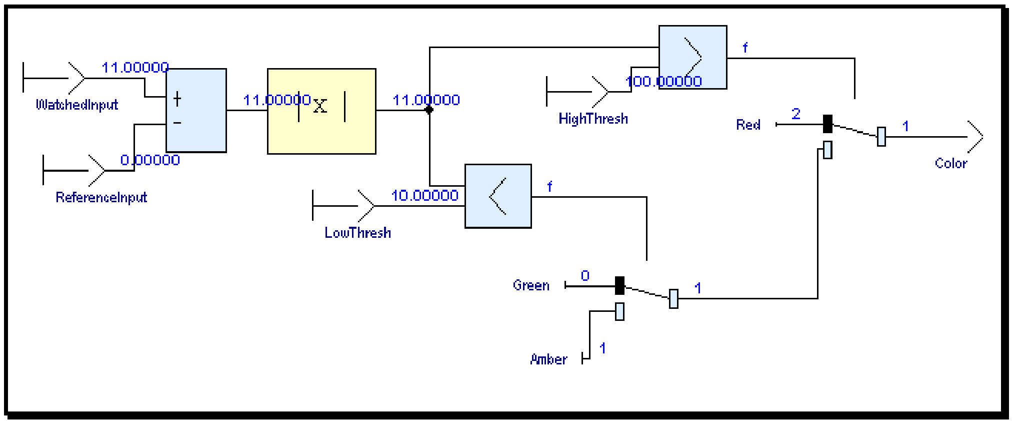


**Figure 27** **Simulation of the color computation function**

| | Step | Remark Column | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| Inputs | WatchedInput:real | | 1 | 10 | 11 | 100 | 101 | -12 | 56 |
| | ReferenceInput:real | | 0 | 0 | 0 | 0 | 0 | 0 | 56 |
| Hidden inputs | LowThresh:real | | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| | HighThresh:real | | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Expected Outputs | Color:LightColors | | 0 | 1 | 1 | 1 | 2 | 1 | 0 |
| Simulated Outputs | Color:LightColors | | 0 | 1 | 1 | 1 | 2 | 1 | 0 |
| OK | Status | | OK | OK | OK | OK | OK | OK | OK |

**Figure 28** **Validation of the color computation function**

For the example of Figure 27, we may characterize the test cases given by Figure 28 in the following manner:

- Cases 1, 3 and 5 cover the block diagrams from a strict MC/DC criterion.
- Cases 2 and 4 are added for checking the accuracy of the comparator.
- Cases 6 and 7 are added to show correctness of absolute value function and are beyond the coverage analysis of the current block diagram.

Structural coverage analysis of software requirements, although not required, is a plus of our methodology, which will help in identifying dead requirements. This was not possible in a more traditional setting because, when one is dealing with informal requirements.

## 9.8 Summary of verification of verification activities

Figure 29 summarizes the shift in verification and verification of verification activities, from traditional development to SCADE Suite based development:

- On the left, with the traditional process:
    - Verification of software requirements is performed by reviews and analyses.
    - There are no precise criteria to verify the verification of the requirements verification.
    - Code is verified by testing and analyses.
    - Testing is verified by MC/DC coverage.
- On the right, with SCADE Suite:
    - Verification of software requirements is enforced by simulation, since the requirements are executable.

Verification of requirements verification is based on structural coverage of requirements (model coverage).

- Verification of the code is suppressed, as a consequence of KCG qualification.
- Verification of the suppressed testing is no longer relevant.

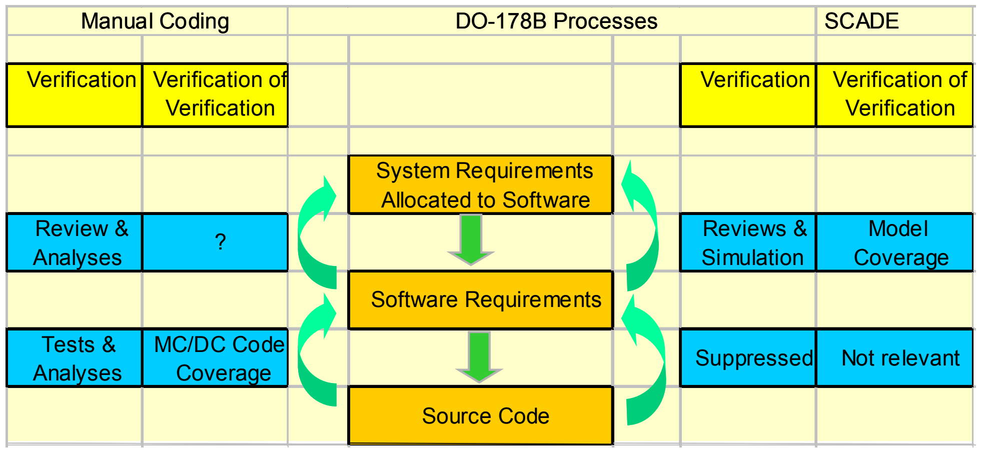


**Figure 29 Verification of verification**

# 10 Appendix D: Qualification of the Code Generator

## 10.1 What does Qualification mean and imply?

**Section 12.2 of ED-12/DO-178B** states that qualification of a tool is needed when ED-12/DO-178B processes "*are eliminated, reduced, or automated by the use of a software tool, without its output being verified as specified in section 6*."

The **FAA notice number 8110.91** provides further explanations regarding tool qualification:

- *ED-12/DO-178B defines verification tools as "*tools that cannot introduce errors, but may fail to detect them." *The following are examples of verification tools:*
  - *a) A tool that automates the comparison of various software products against some standard(s).*
  - *b) A tool that generates test procedures and cases from the requirements.*
  - *c) A tool that automatically runs the tests and determines pass/fail status.*
  - *d) A tool that tracks the test process and reports if the desired structural coverage has been achieved.*

- *ED-12/DO-178B defines development tools as "*tools whose output is part of the airborne software and thus can introduce errors." *If there is a possibility that a tool can generate an error in the airborne software <u>that would not be detected</u>, then the tool cannot be treated as a verification tool. An, example of this would be a tool that instrumented the code for testing and then removed the instrumentation code after the tests were completed. If there was no further verification of the tool's output, then this tool could have altered the original code in some unknown way.* Typically, the original code prior to instrumentation is what is used in the product. This example is included to demonstrate that tools used during verification are not necessarily verification tools. The effect on the final product must be assessed to determine the tool's classification

Section 12.2.1 states that:

*a. 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 airborne software.*

*b. The software level assigned to the tool should be the same as that for the airborne software it produces, unless the applicant can justify a reduction in software level of the tool to the certification authority.*

In summary, the user has to make sure that if he intends to use a tool, that tool has been developed in such a way that it can be qualified for its intended role (verification or development) and the safety level of the target software. Note that qualification is on a "per project" basis, although it is usually simpler and faster to re-qualify a tool in a context similar to the one it has already been qualified.

## 10.2 Prequalification of KCG as a Development Tool

The SCADE Qualifiable Code Generator (KCG) has been developed in such a way that it is "pre qualified" which means that it is ready for qualification on specific projects (remember that "qualification" is on a per project basis).

Safety objectives for Level A have been assigned to the tool, and its development has been conducted in accordance with these objectives, for instance:

- ❑ PSAC (Plan for Software Aspects of Certification) has been written before starting development.

- Analyses and reviews have been conducted in the same way as for embedded software.
- MC/DC coverage has been achieved.

## 10.3 Qualification Data

Qualification data are provided with KCG. Their purpose is to provide the data necessary to qualify the usage of SCADE KCG on specific projects.

The table below shows the documents that are required by the authorities and must be available upon request. The KCG qualification kit includes those that are listed as "delivered". Those listed as "accessible" are available to the authorities upon request at the Esterel Technologies premises.

| Data | FAA requirement N8810.91 | KCG package | DO-178B Reference |
|---|---|---|---|
| Tool Qualification Plan | Submit | Delivered | 12.2.3.a(1), 12.2.3.1, & 12.2.4 |
| Tool Operational Requirements (see detail below) | Available | Delivered | 12.2.3.c(2) & 12.2.3.2 |
| Tool Accomplishment Summary | Submit | Delivered | 12.2.3.c(3) & 12.2.4 |
| Tool Verification Results | Available | Accessible | 12.2.3.c |
| Tool Qualification Development data (e.g., design, code, test cases and procedures) | Available | Accessible | 12.2.3.c |
| | | | |

**Figure 30 Qualification data of the code generator**

The meaning of the data is the following:

**Tool Qualification Plan**: Describes the tool qualification process

**Tool Operational Requirements:** Describes the KCG specification. It is composed of the following documents:

•Reference Manual of the SCADE language.

•Software Specifications of SCADE/KCG.

**Tool Validation Plan:** Description of verification procedures

**Tool Verification Results:** Results of tests, reviews and analyses

**Tool Accomplishment Summary:** Summarizes and concludes what has been achieved, DO-178B Level A objectives, usage conditions, possible limitations

**Tool Configuration Index:** Identifies the configuration of the tool and its components