Efficient Development of Avionics Software with DO-178B Safety Objectives

© Esterel Technologies 2002

Abstract

This white paper addresses the issue of cost and productivity improvement in the development of safety critical software for avionics systems. Such developments follow guidelines defined by the ED-12/DO-178B document. This paper first reviews the activities traditionally performed in such developments. It then describes how individual tools can increase productivity in such a context. A global optimization of the development process with the SCADE Solution is then presented, which allows much larger savings than isolated tools. Industry examples demonstrate the efficiency of this approach, and business benefits are analyzed.

Author(s)

Jean-Louis CAMUS (Esterel Technologies)

ESTEREL
Technologies

Table of contents

1 Executive summary 5

2 The challenge of developing Software for Safety critical systems 6

3 ED-12/DO-178B survey 7

3.1 What is ED-12/DO-178B? 7

3.2 Life cycle processes structure 7

3.3 The development processes 8

3.4 The verification processes 9

4 Productivity enhancement with individual tools 13

4.1 Requirements and traceability management 13

4.2 Test support 13

4.3 Code analysis 14

4.4 Code generation 14

4.5 Simulation 14

4.6 Formal verification 14

4.7 Tool qualification 14

5 The SCADE solution for the development of safety critical software. 16

5.1 Savings by sharing accurate requirements 16

5.2 Savings in the coding phase 20

5.3 Savings in integration testing 22

5.4 Esterel Technologies offer for the development of safety critical software 22

6 Examples of productivity enhancements 23

6.1 Eurocopter EC 135/155 23

6.2 Airbus A340 23

7 Business benefits 24

8 Annex A: ED-12/DO-178B Glossary 26

8.1 Acronyms 26

8.2 Glossary 26

9 Annex B: References 29

10 Annex C Summary of savings on DO-178B required verification activities 30

Table of figures

Fig. 1 DO-178B Processes 8

Fig. 2 DO-178B Development processes 9

Fig. 3 SCADE Position in the development flow 16

Fig. 4 SCADE Suite diagrams 17

Fig. 5 Simulation allows to “play the requirements” 18

Fig. 6 From system analysis in Simulink to software with the SCADE Gateway 19

Fig. 7 From the V cycle to the Y cycle 24

Fig. 8 Cost reduction with CG and KCG 25

© Esterel Technologies 2002

© Esterel Technologies 2002

1 Executive summary

Companies developing avionics software are facing a challenge. On one hand, safety is not an option but a 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 10 Kilo Lines of Code (KLOC) level B software is 16 persons-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

On the other hand, the growing complexity in 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 guided by ED-12/DO-178B: where are the costs, how to reduce them by adopting efficient methods and tools.

First, an introduction to DO-178B activities is provided, to understand why safety objectives lead to high costs, if traditional development processes are used. In particular, verification activities are analyzed: they include not only extensive testing, but also analysis and review activities. It is shown that these verification activities are the cost driver and the bottleneck in project schedule.

A survey of the use of individual tools (such as simulation, code generation, test coverage tools) is made. It shows the benefits but also the limitations of automating individual pieces of the development process. While providing some gains, these techniques have a limited impact: they don't significantly affect the efforts needed to ensure and verify the consistency of software lifecycle data.

Then, a more global process optimization is described, with its supporting toolset, SCADE. It implements principles, which are based on consistency and accuracy:

- An accurate software description is shared among project participants
- Design errors are detected/fixed early in the lifecycle
- Qualified code generation saves not only writing the code, but also verifying it.

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

Finally, business benefits of this global improvement are analyzed. They include a cost reduction of 50%, a time to implement a change of 48 hours, and high-level reusable components, all leading to a high competitive advantage.

© Esterel Technologies 2002

2 The challenge of developing Software for Safety critical systems

Companies developing avionics software are facing a challenge. On one hand, safety is not an option but a 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 10 K Lines of Code (LOC) level B software is 16 persons-years
- The average avionics software is 46% over budget and 35% behind schedule
- The time to market was 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

On the other hand, the functionalities of software increase every year. As an example, the evolution of software size for the Airbus family is the following.

Aircraft	A310 (70')	A320 (80')	A340 (90')
Nb of digital units	77	102	115
Volume of onboard software in Mbytes	4	10	20

Similar figures apply to Boeing aircrafts.

We have also moved from an era where aircrafts were something prestigious, with limited competition, to an era with a high pressure on price and time to market.

Cost increase factors are directly conflicting with these economic constraints. If nothing changes, companies developing safety critical software will have more and more competitiveness problems.

In this paper we address the issue of productivity in the development of software for civil aircrafts, as guided by ED-12/DO-178B: where are the costs, how to reduce them by adopting efficient methods and tools.

3 ED-12/DO-178B survey

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 explicits some of the terminology that may be non intuitive to the reader and/or differ from usage in other domains.

3.1 What is ED-12/DO-178B?

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.

ED-12/DO-178B has been published by EUROCAE (a non profit organization addressing aeronautic technical problems) and RTCA (Requirements and Technical Concepts for Aviation). It provides guidelines for the production of software for airborne systems and equipment. The objective is that this software performs its intended function with a level of confidence in safety that complies with airworthiness requirements

These guidelines are in the form of :

- 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

Recommendations are provided in the areas of software planning, development, verification, configuration management, quality assurance, certification, and maintenance. A System Safety Assessment process is required to associate criticality levels with the various system elements.

ED-12/DO-178B defines 5 Development Assurance Levels for embedded software:

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).
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.

3.2 Life cycle processes structure

ED-12/DO-178B structures activities as a hierarchy of “processes”. This term “process” will appear several times in that document. It defines three top-level software life cycle groups of processes:

- The software planning processes that defines and coordinates the activities of the software development and integral processes for a project. It is beyond the scope of this paper.

- The software development processes that produces 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.

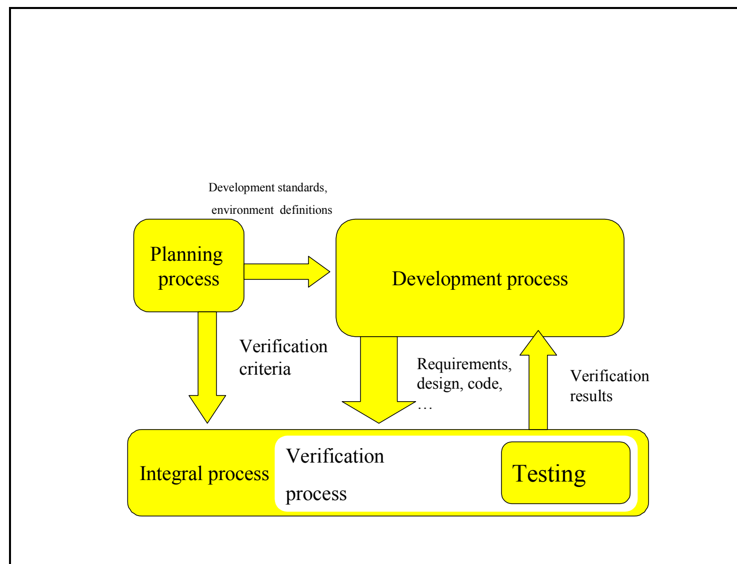


Fig. 1 DO-178B Processes

The most relevant process flows are the following:

- The planning process defines development standards (ex: requirements and coding standards) and environment (ex: tools)
- The development processes produce requirements, design, source code and integrated binaries
- The Integral processes ensure the correctness, control and confidence of the software life cycle processes and their outputs. The verification processes are part of the integral processes, and testing is part of the verification processes.

In the remainder of this document we focus on the development process and on the related verification activities.

3.3 The development processes

The software development processes are composed are:

- 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 produces object code and builds up to the integrated system or equipment.

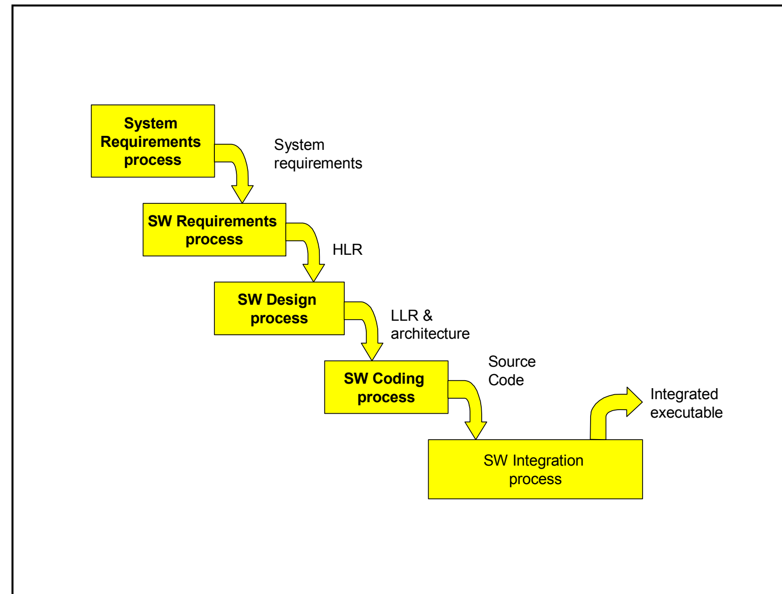


Fig. 2 DO-178B Development processes

The high level SW 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.

3.4 The verification processes

3.4.1 Objectives of software verification

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

Testing is part of the verification processes, but verification is not just testing. The verification process also relies 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.

© Esterel Technologies 2002

3.4.2 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 up to 80% of the budget [Amey, 2002]. It is also a bottleneck for the project completion. So, clearly any change in speed and/or cost of verification has a major impact on project time and budget.

Please note that the objective of this paper is by no means to relax the objectives of verification, but to improve the efficiency of the process, achieving at least the level of quality achieved by traditional means. This requires understanding and optimizing the whole development process.

In order to understand the verification activities, the remainder of this section describes the main verification activities related to the development process. Without any special tool, all these tasks have to be performed manually, and although the usage of tools is progressing, many activities are still performed manually.

3.4.3 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

3.4.4 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 that 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 that deviations from the standards are justified.
f. Traceability: the objective is to ensure that the high-level requirements were developed into the low-level requirements.
g. Algorithm aspects: ensure the accuracy and behaviour 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

© Esterel Technologies 2002

3.4.5 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.

3.4.6 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. There are 3 types of testing activities:

<u>Low-level testing</u>: To verify the implementation of software low-level requirements.

<u>Software integration testing</u>: 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.

<u>Hardware/software integration testing</u>: To verify correct operation of the software in the target computer environment.

DO-178B imposes that all test cases are requirements-based. Blind testing based on code, without clear relationship with the requirements is not valid.

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 have to 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 reverification
- 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 which 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.
- 100% MCD/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

4 Productivity enhancement with individual tools

This chapter, analyzes how individual tools can be used to automate or accelerate isolated activities.

4.1 Requirements and traceability management

Requirements management tools support the capture and management of requirements. These are generally defined as a hierarchy of labeled/numbered items containing text and/or figures. These tools support direct capture, and often retrieval of the requirements from external documents.

Other tools, or often the same tools support the management of traceability links between several levels and types of requirements, between requirements and test cases, or even other life cycle data. Various reports can be generated, listing requirements and/or links in both directions.

Since the number of links may be very high, such tools save significant amounts of time and effort, in particular when changes are required: it is easier to analyze the impact of a change, and it is faster and more reliable to update the requirements documents and the traceability reports.

4.2 Test support

4.2.1 Test harness and test case writing support

Writing test cases and developing their environment is a heavy, error prone task.

First, for a module to be tested, a test harness has to be developed: it is a calling context (a main program for instance) that can define input data, call the unit under test and gather the result. A test harness generation tool generates the calling environment by analyzing the interface of the unit under test.

A test case writing tool (often the same as the one providing the test harness) helps in writing test cases (inputs and expected outputs). It can provide a skeleton for input values, based on analysis of the input data domain. But this part of the job is not only incomplete; it is based on the code, regardless of the requirements, which is not accepted by DO-178B. So, it is required that humans write requirement-based test procedures.

4.2.2 Test execution and management support

Once the test harness and the cases are available, it is necessary to run them, to collect the results and to analyze them. Test execution tools support these tasks, analyze differences between two runs, and generate reports about test execution, their result (passed, failed, inconclusive), and various statistics.

They also allow to organize and store/retrieve test data. In practice, test execution, management and test harness support are often integrated in one test environment.

All these test support tools save considerable amounts of time and effort.

4.2.3 Structural coverage measurement support

There are many tools for the analysis of structural coverage. Most of them collect data by instrumentation of the source code. Some use debug or hardware emulation techniques. ED-12/DO-178B accepts that code coverage is measured at the source level, provided one can demonstrate traceability of the object code with the source code (this implies at least deactivation of optimization).

They may measure statement coverage, decision coverage or MC/DC coverage. Manual measurement is error prone and expensive, so structural coverage measurement tools tend to be adopted at least for large projects.

But this is just coverage measurement, not production of test cases or coverage resolution.

4.3 Code analysis

Several types of static analyses can be performed on the code, concerning:

- Code complexity analysis, for control structures (ex: Mc Cabe complexity metric), or for code vocabulary (example Halstead metrics); these are not required by DO-178B, but may be useful to analyze complexity and quality of software.
- Programming rules (standardized rules such as MISRA or SPARK Ada rules, or project specific rules); these rules have to be defined in the coding standards.
- Run-time error analysis: these tools are able to detect potential overflow of division by zero, unreachable code, non-terminating loops or recursion.

4.4 Code generation

Code generation is generally based on the requirements and on the architecture defined in the design phase. Code generation capabilities range from simple generation of skeleton, to generation of the complete code for parts of the application, such as: functional behavior, human machine interfaces, coding/decoding functions. The more precise the requirements are, the more complete and accurate the code generation can be.

Savings depend on the ratio of generated code for the application. In an avionics context, they depend even much more on the savings on verification: if the code generator is not qualified (see below), all verification activities have to be performed.

4.5 Simulation

Simulation is one of the most used techniques to understand and validate software, or a model of that software. Many simulation environments are customer specific, to fit their environment, such as control panels, prototype target systems. Their main drawback is the cost of maintenance and a limited reusability. On the other hand, general-purpose simulation tools may offer more advanced debugging features, faster availability at lower costs, but may lack specific features. The ideal solution allows integrating the general-purpose environment to the dedicated one.

4.6 Formal verification

Formal verification techniques are based on mathematical principles to formally demonstrate that a property holds or on the contrary if it does not, to provide a “counterexample”, which is a scenario violating the property.

Safety requirements are of particular interest. For instance “no stop command should be sent to the engine during takeoff”.

A prerequisite to formal verification is the formality of the model that has to be verified. Then, various proof techniques can be used, such as BDD (Binary Decision Diagrams), Induction, State-Space exploration. Tools automating proof range from expert-only to user friendly, easy to learn tools.

In practice, the current penetration of verification tools is limited by the lack of formal descriptions in most development projects. Availability of formal descriptions without supplementary cost is required for a real penetration of formal verification.

4.7 Tool qualification

While tools may save time and effort, they might introduce risks in the process. So DO-178B imposes qualification constraints on tools.

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 that would not be detected, 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 requalify a tool in a context similar to the one it has already been qualified.

5 The SCADE solution for the development of safety critical software.

This chapter shows how further savings can be achieved by optimizing the global design flow.

The SCADE solution for the development of safety critical software is a tool supported process that integrates the development flow in a seamless, consistent flow.

The principles of the solution are:

- **Share accurate requirements**
- **Do things once**: do not rewrite things 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.

As an example, we show how a tool such as SCADE™ can support this unified process. The position of SCADE in the lifecycle is depicted in the figure below.

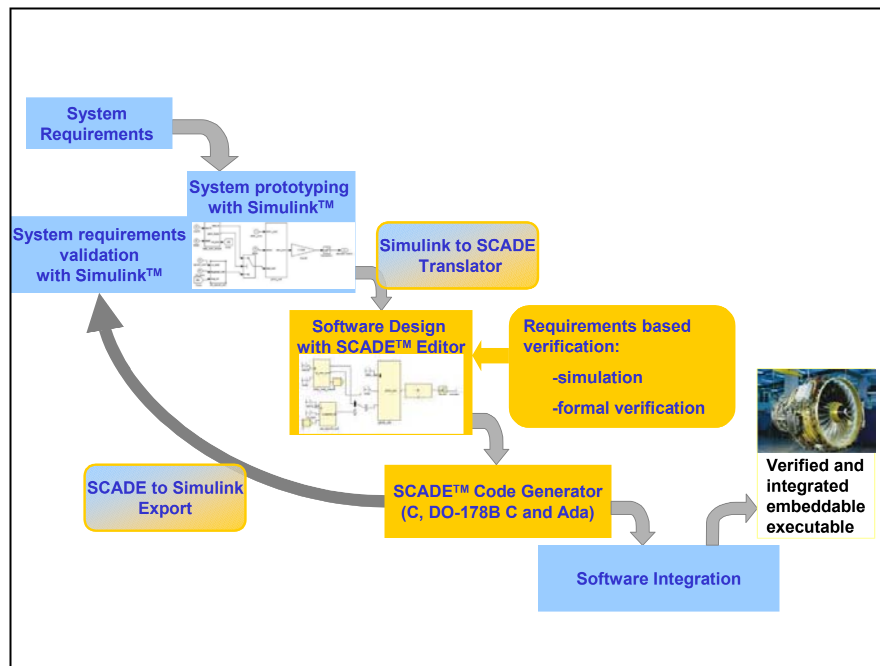


Fig. 3 SCADE Position in the development flow

5.1 Savings by sharing accurate requirements

5.1.1 Readability

An efficient process has to be based on good communication amongst people and activities.

First, people have to understand easily descriptions made by the others. The SCADE notation for writing the specification is easy to learn and use, since it is based on block diagrams and state machines, commonly used by system and software engineers. Block diagrams are used to capture data processing flow and state machines are used to capture the reactions to events.

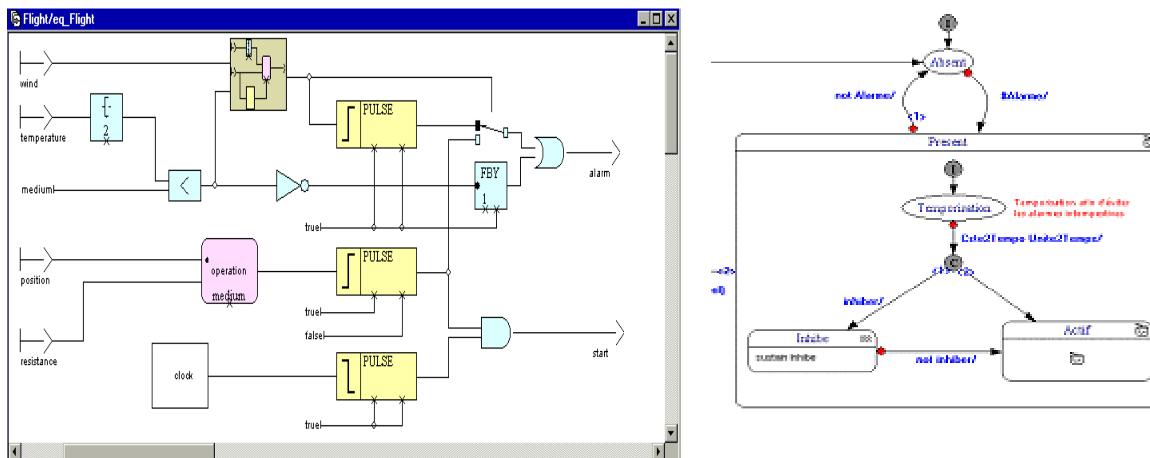


Fig. 4 SCADE Suite diagrams

5.1.2 Accuracy and determinism

DO-178B requires accurate requirements and deterministic behavior.

SCADE is based on Lustre [LUSTRE], which is a synchronous data flow language for the definition of reactive systems. A system is said to be "reactive" if it continuously interacts with its environment, as opposed to systems that only react to events (ex: a telecom switch), or compute and stop (ex a bill processing system). "Synchronous" means that it is based on logical "instants" (typically sample times), where the output is computed from the current input and state of the system, without interfering with the environment during computation in the "instant". This type of description addresses well the functional part of reactive software. This accounts for instance for 70% of the software for a flight control system. On the contrary, hardware drivers are beyond the scope of SCADE.

Since Lustre is a formal language, and since SCADE is semantically based on Lustre [SCADE LANG RM], 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 differences.

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.

5.1.3 Simulation of requirements

While having a clear specification of the software on paper or on screen is required, it is also helpful to exercise dynamically the behavior of that specification, to better understand how it behaves.

As soon as a SCADE model is available, it can be simulated with the SCADE simulator. Simulation can be interactive or batch. Scenarios (input/output sequences) can be recorded, saved, replayed later on the simulator or on the target.

© Esterel Technologies 2002

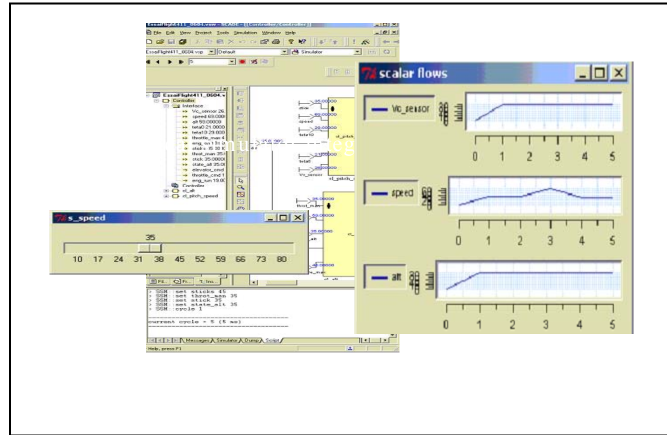


Fig. 5 Simulation allows to "play the requirements"

5.1.4 Structuring the high level requirements

The SCADE model can define the high level functions of the software. It can formalize the interfaces of the system and of its high level functions, and the data flow between those functions. State machines may be used to explicit the states of the system and how it reacts to events.

A requirements management tool such as DOORS helps in structuring requirements and managing traceability with other lifecycle 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.

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

Note: it is also possible to have the high level requirements and the low level requirements in just one set of documentation. In that case, the statements of the paragraph below also apply to the high level requirements.

5.1.5 Writing an accurate design

The design phase produces the Low level requirements (LLR) and the software architecture. SCADE is ideal for a precise description of the functional part of the software and of its software architecture.

The complete SCADE model details the low level functions of the software. It formalizes the interfaces of its low level functions, and the data flow between those functions. State machines may be used to structure the states of these functions and their reaction to events.

Architectural design choices are expressed by several non-exclusive means:

- Hierarchy of functions and data structures
- Data flow dependencies between functions
- Scheduling structure by expressing activation conditions

  SCADE automatically analyzes the consistency of data flow dependencies, taking into account the activation conditions, and checks that there is no causality loop in the network of data flow. These dependencies are taken into account also at code generation time.

5.1.6 Benefits of the "do things once" principle

The SCADE model formalizing the requirements and design 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.

© Esterel Technologies 2002

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

5.1.7 Teamwork and reuse

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

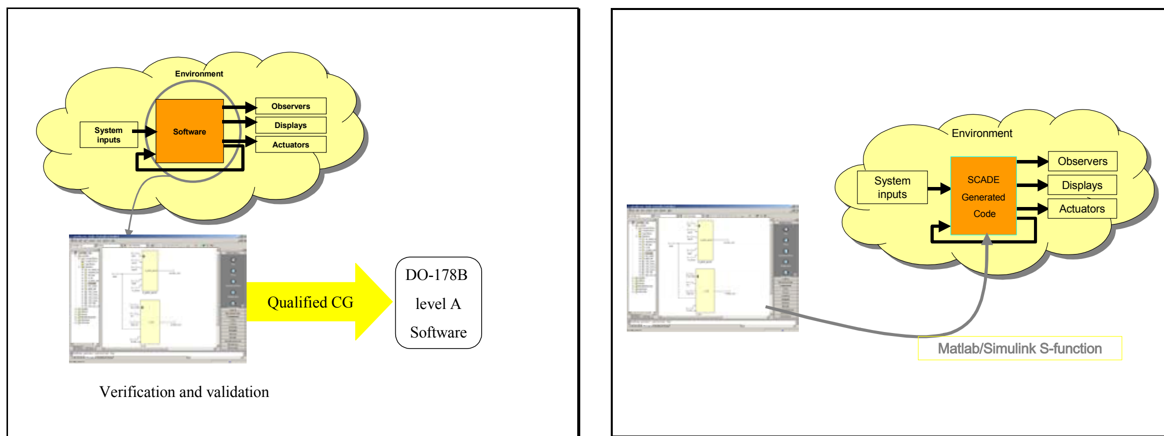
The SCADE language is modular: there is a clean distinction between interfaces and contents of modules (called “nodes” in SCADE) and there are no side effects from one node to another. So, a large software model can be split into several smaller SCADE models, when their interfaces are defined, and each team member can work on a specific part (called a SCADE “project”). Later, integration of those parts in a larger model can be achieved by linking these “projects” in a larger one. The SCADE semantic checker verifies the consistency of this integration.

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 standard SCCI interface (Microsoft Source Code Control Interface), supported by most configuration management systems.

Reuse is an important means of improving productivity and consistency in a project or a series of projects. SCADE libraries can store definitions of nodes and/or data types, which can be reused in several places. These range from basic nodes such as latches or integrators to complex, customer specific systems.

5.1.8 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.



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

Fig. 6 From system analysis in Simulink to software with the SCADE Gateway

When a tool such as MATLAB/Simulink™ 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 a level required by ED-12/DO-178B. In particular data types and clocks have to be more accurate. SCADE also gives access to formal verification.

Second, the user can take benefit of 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, thanks to the qualified code generator (see section on code generation).

5.1.9 Savings in verification activities of the requirements

ED-12/DO-178B imposes to verify that LLR are accurate and consistent, verifiable and traceable with the HLR.

Since SCADE has a formal basis, the LLR written are accurate: they have a precise semantic of the dynamic behavior. The SCADE checker verifies that all data are typed, that types are consistent, that variables are initialized and that there are no time/causality ambiguities.

The DOORS™ Gateway allows managing traceability with the high level requirements. DOORS Gateway allows analyzing the coverage of the high level requirements by the low level requirements, and identifying those high level requirements that are not covered. The flexible annotation feature of SCADE is a complementary means of managing such information.

With SCADE, the specification can be readily executed through interactive, random or batch simulation**.** Requirements-based simulation scenarios can be defined and run as soon as the SCADE specification is available, thus allowing to verify the compliance of the LLR to the HLR. The same code as the one that will be used on the final target may be generated and downloaded to a prototype target (such as an “iron bird”) to validate the behavior in an integrated environment. Thus a “hardware in the loop” validation can be performed very early.

Critical properties, such as safety properties can be extracted from the HLR and formalized. These properties can then be formally verified on the SCADE LLR, using the Prover Plug-In for SCADE. Thus, we have requirements-based formal verification of those properties, answering DO-178B requirements.

The proof engine is also a scenario generator. It can be asked for instance to generate a scenario such that the software will produce a given output (e: raise alarm X).

In summary, large amounts of verification efforts are minimized by construction, by traceability and by automated verification techniques. Last but not least, the scenarios produced by simulation and formal verification activities may be reused for requirements-based testing at a later stage.

5.2 Savings in the coding phase

5.2.1 Automation of code generation

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. Scheduling of the code complying with the data flow and activation conditions is generated.

5.2.2 Ways of using and presenting a SCADE Code Generator for certification of software

There are three ways of using a SCADE Code Generator for the development of software, and presenting it accordingly to the certification authority:

- *Productivity tool*: the tool is a way of writing the code more effectively. But no reduction of verification is claimed. Everything is verified as if the code were written manually.

- *Verification tool*: the tool is used as above, but is also considered to save manual verification of the coding rules and of traceability. A failure of these verifications cannot directly introduce an error in the final application. But all verifications of the code behavior (test, requirements and structural coverage) have to be performed as if the code were written manually.
- *Development tool*: if the tool can be qualified to the appropriate level, most verification activities required on its output may be partly or totally omitted.

All SCADE Code Generators can be qualified as productivity or verification tools.

SCADE KCG C Code Generator has already been qualified as a **development** tool for ED-12/DO-178B level A, i.e. the most demanding level, on several aircraft and helicopter equipment development, projects. It is currently **the only commercially available Code Generator qualifiable as a development tool for level A**.

5.2.3 Savings on verification of the source code

SCADE can reduce/eliminate the following analyzes and reviews of the coding phase:

- Compliance with low level requirements: this is by definition what the Code Generator ensures when its output is correct; qualification of the Code Generator as a development tool saves this verification.
- Compliance with the software architecture, to ensure that the source code matches the data flow and the control flow defined in the software architecture: the Code Generator ensures this compliance by computing a sequence of statements compliant with the data flow description, including complex delays and loops. Qualification of the Code Generator as a development tool saves this verification activity.
- Verifiability: the relationship between code and requirements defined by the specification of the SCADE code generator saves this verification
- Conformance to standards: the code generated by SCADE not only conforms to standard C or ADA, making the code portable. It also follows coding standards suited to the safety critical context, which have been defined with industrial partners from aerospace and energy. Qualification of the Code Generator as a verification or development tool saves this verification activity.
- Traceability: all constructs from the SCADE model are traceable in the generated code by names and/or comments. Qualification of the Code Generator as a verification or development tool saves this verification activity.
- Accuracy and consistency are guaranteed:
    - Data types and module integration consistency is verified on the SCADE model, and is reflected in the generated code
    - Accuracy and consistency of clocks and data flows is verified on the SCADE model and reflected in the generated code.
    - Memory usage is reliable since it is allocated statically, and statically bounded stack is used. Memory accesses are only made by value transfer or via static pointers.
    - No loops, no exceptions, no operating system calls are in the generated code.
    - All variables are initialized from their explicit definition in the SCADE model, where initialization is mandatory.

Other verifications are made easier by the ability to generate readily the code from the LLR. For instance, the code can be readily downloaded to the target to analyze execution time (Worst Case Execution Time) and other non-functional aspects.

5.2.4 Low level testing

ED-12/DO-178B uses the term “Low Level Testing” rather than “Unit Testing”. The objective of Low Level Testing is to verify the implementation of LLR.

The qualification of the Code Generator as a development tool replaces the verification of compliance of the code to the low level requirements, and the unit testing of the generated code is no longer required.

Unit testing of any other code (example: procedures/functions written by the user and called by the generated code or calling it) is still mandatory.

Note: special analyzes are required if there are differences in the representation of numbers between LLR and the target implementation, and if the algorithms are sensitive to these representations. If the algorithms are sensitive to floating point precision, this has to be identified in the design phase (rather than discovered by testing the code!!), and design standards have to be applied to master this. Similarly, if fixed-point arithmetic is used, design standards have to be applied at the design level, and should include range analysis and precise behavior of operators (such as saturation). Specific testing activities should then be used as a way of verifying those numeric analyses of the design.

5.3 Savings in integration testing

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

As mentioned above, using the SCADE simulator and Prover Plugin for SCADE for the verification of the LLR produces large amounts of requirements based test scenarios. These test scenarios are first class material for requirements-based testing on the target, since:

- They are requirements based, as mandated by ED-12/DO-178B
- 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 environment using simple scripts.

5.4 Esterel Technologies offer for the development of safety critical software

Esterel Technologies offers tools, services and support for the development of safety critical software. There are 2 levels of product: the "regular" toolset and the qualification package.

5.4.1 The regular toolset

The regular SCADE toolset is made of:

- The editor
- The simulator
- The SCADE™ Simulink Gateway
- The formal verifier (Prover Plug-In for SCADE)
- One of the "regular" Code Generators: C or ADA
- Gateways to configuration management tools, with the SCCI (Source Code Control Interface, supported by most commercial configuration management tools), and to the requirements management tools DOORS

5.4.2 The KCG qualification package

The KCG package gives access to code generation qualified as a development tool. It contains:

- The qualifiable Code Generator executable
- The baseline documentation for the certification authorities

6 Examples of productivity enhancements

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

6.1 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 are level A. The principal challenges were 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 partners. 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 in resulting in this type of 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

6.2 Airbus A340

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

Airbus participated in the SCADE definition and introduced SCADE as a successor to part of their in house tools. Today, SCADE models are written and exchanged in the whole company. 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 is level A. 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 already 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 [Airbus_Cost]. 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.

© Esterel Technologies 2002

7 Business benefits

This chapter analyzes the business benefits of the above described process improvement:

7.1.1 Time to market and cost reduction with the "Y" cycle

If we take the traditional "V cycle" as a reference (a), the figure below summarizes the time and cost savings, depending on the operation mode.

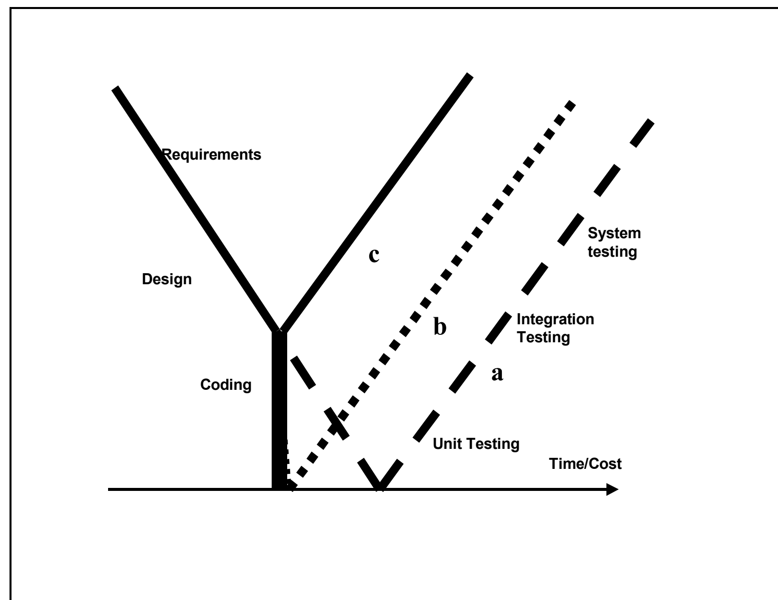


Fig. 7 From the V cycle to the Y cycle

The 2 currently used modes are the following:

- (b)When 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 change at all, and 20% if some changes are made. All verification activities have to be performed.
- (c) When 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

Alternative modes are the following:

- If the Code Generator is just qualified as a "verification tool", its saves the verification of coding rules and traceability, and saves 25% of budget.
- In the case where undetected error introduction by the compiler is not mastered, object code testing has to be done. But reusing the simulation scenarios will save significant amounts of effort. For this situation, we currently have no measurements available; we expect the savings to be in the range 30% to 35%.

© Esterel Technologies 2002

- Introduction of proof techniques could lead to 10% more savings, according to Airbus projections. 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.

For the regular CG, the figures assume that it is qualified as a verification tool. Although the effort of writing the requirements and design can be reduced, 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. In all cases, the verification cost is decreased by 20%, mainly thanks to the verification of the design by the SCADE checking tools. When using KCG, there are major supplementary savings: one is directly the savings of the "normal" verifications, the other is due to the savings in the verification of code when correcting a requirement and reflecting this change in the implementation.

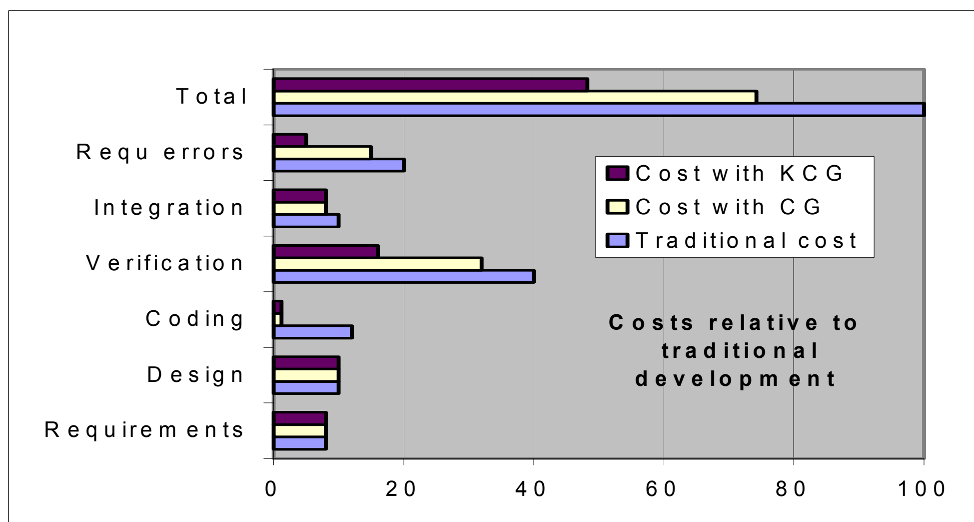


Fig. 8 Cost reduction with CG and KCG

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.

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 having such a shorter response time.

7.1.2 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 (even if good coding rules have been applied), and often target dependant. 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 without changing the SCADE models.

© Esterel Technologies 2002

8 Annex A: ED-12/DO-178B Glossary

8.1 Acronyms

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

8.2 Glossary

Baseline - The approved, recorded configuration of one or more configuration items, that thereafter serves as the basis for further development, and that is changed only through change control procedures.

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 Authority - The organization or person responsible within the state or country concerned with the certification of compliance with the requirements.

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

© Esterel Technologies 2002

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

Emulator - A device, computer program, or system that accepts the same inputs and produces the same output as a given system using the same object code.

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.

© Esterel Technologies 2002

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.

© Esterel Technologies 2002

9 Annex B: References

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

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

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

[HAL] Halstead, M.H.: "Elements of Software Science". *Prentice-Hall, Inc., New York, 1977*

[Mc Cabe] T.J. Mac Cabe, "A complexity measure", *IEEE Transactions on Software Engineering, S.E.,2(4),76.*

[Lustre] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. "The synchronous dataflow programming language Lustre*". Proceedings of the IEEE, 79(9):1305--1320, September 1991*

[SCADE LANG RM] "SCADE Language Reference Manual", *Esterel Technologies 2002*

10 Annex C Summary of savings on DO-178B required verification activities

Potential Effort Reduction in the verification				
*	**	***	****	*****
0%	From 0% to 20%	From 20% to 50%	From 50% to 80%	From 80% to100%

Verification of Outputs of Software Design Process (A-4)				
	Objective	**Potential Effort Reduction**	**Comments**	**From Level**
1	Low-level requirements comply with high-level requirements	****	Precise notation and the capability of early simulation of SCADE model are very powerful for checking high-level requirements compliance	C
2	Low-level requirements are accurate and consistent	*****	Consistency and completeness are automatically checked	C
3	Low-level requirements are compatible with target computer	**	Early code generation is a good way to check the overall compatibility.	B
4	Low-level requirements are verifiable	*****	Anything described in SCADE is verifiable	B
5	Low-level requirements conform to standards	*****	A SCADE model checked by the SCADE checker conforms to the SCADE standard notation. Project specific rules can also be defined and automatically checked	C
6	Low-levels requirements are traceable to high-level requirements	**	Annotations can be given to any SCADE construct The DOORS link also supports traceability	C
7	Algorithms are accurate	****	A correct SCADE model is accurate. This does not cover numerical aspects.	C
8	Software architecture is compatible with high-level requirements	****	The architecture is deduced from the high-level requirements	C
9	Software architecture is consistent	*****	Produced by the code generator, based on the model and the code generation options.	C
10	Software architecture is compatible with target computer	**	No specific OS or hardware dependencies in the generated code	B
11	Software architecture is verifiable	*****	Thanks to the structure of the generated code characteristics	B
12	Software architecture conforms to standards	*****	By construction	C
13	Software partitioning integrity is confirmed	**	Generated code has no side effect (bounded time and memory)	D

© Esterel Technologies 2002

Verification of Outputs of Software Coding & Integration Processes (A-5)				
	Objective	**Potential Effort Reduction**	**Comments**	**From Level**
1	Source Code complies with low-level requirements	*****	Every automatically generated piece of code complies with its design.	C
2	Source Code complies with software architecture	*****	The generated code conforms to the model and the code generation architectural options.	C
3	Source Code is verifiable	*****	Ensured by the code generator specification	B
4	Source Code conforms to standards	*****	Conformance to code generator standard is automatic	C
5	Source Code is traceable to low-level requirements	*****	The source code is fully traceable to the SCADE model thanks to automatically generated comments	C
6	Source Code is accurate and consistent	****	Given by the characteristics of the generated code Does not cover numeric and fixed point aspects	C
7	Output of software integration process is complete and correct	*		C

© Esterel Technologies 2002