Ansys

Efficient Development of Safe Automotive Software with ISO 26262 and ASPICE using SCADE

Methodology Handbook / First Edition

/ /

ABSTRACT

This methodology handbook provides detailed explanations on how to fully satisfy ISO 26262:2018 safety-related requirements and recommendations with a SCADE model-based software development approach while promoting an efficient development and verification strategy. The proposed approach also aims at reducing costs and improving software quality in supporting ASPICE guidelines.

The handbook introduces the ISO 26262:2018 standard before presenting the optimization of the software development and verification processes that can be achieved with the SCADE toolchain and methodology. SCADE tools support the automated production and verification of a large part of the development lifecycle elements. The effect of using the SCADE toolchain is presented in terms of savings in the development and verification activities, following a step-by-step approach and considering the objectives that must be met at each step. The inclusion of a SCADE-based workflow in a broader AUTOSAR software development workflow is also considered.

The handbook does not intend to impose formal conditions of use. Formal guidelines can be found in the SCADE Suite KCG Safety Case and in the TÜV SÜD Reports on the SCADE Suite KCG, SCADE ACG, SCADE Test, and SCADE LifeCycle certificates.

TABLE OF CONTENTS

ABSTRACT 2
TABLE OF CONTENTS 3
LIST OF FIGURES 7
LIST OF TABLES 10
1 DOCUMENT BACKGROUND, OBJECTIVES, AND SCOPE 12
1.1 Background 13
1.2 Objectives and Scope 13
2 DEVELOPMENT OF SAFETY-RELATED AUTOMOTIVE SOFTWARE 15
2.1 Introduction to the ISO 26262:2018 Series of Standards 16
2.2 Hazard Analysis and Risk Assessment (HARA) in ISO 26262-3:2018 18
2.2.1 Situation analysis and hazard identification 18
2.2.2 Hazardous event classification 19
2.2.3 Estimation of potential severity 19
2.2.4 Estimation of the probability of exposure regarding operational situations 19
2.2.5 Estimation of controllability 20
2.2.6 ASIL assignment 20
2.3 From Hazardous Events in ISO 26262-3:2018 to Technical Safety Requirements in ISO 26262-4:2018 21
2.4 Software Development Process Overview in ISO 26262-6:2018 22
2.4.1 Reference phase model for the product development at the software level 22
2.4.2 Objectives, prerequisites, and work products of each sub-phase 23
2.4.3 Model-based development (MBD) approaches in ISO 26262-6:2018 27
2.5 Confidence in the Use of Software Tools in ISO 26262-8:2018 29
2.5.1 Required level of confidence in a software tool 29
2.5.2 Possible methods to qualify a tool 29
3 MODEL-BASED DEVELOPMENT WITH SCADE 31
3.1 What is SCADE? 32
3.1.1 SCADE origin and application domain 32
3.1.2 SCADE as a bridge between control and software engineering 33
3.2 Scade Modeling Techniques 34
3.2.1 Modeling behavior with Scade 34
3.2.2 The SCADE Suite cycle-based intuitive computation model 39
3.2.3 SCADE modeling and safety benefits 40
3.3 The SCADE Toolchain 41

3.3.1 SCADE toolchain overview 41
3.3.2 SCADE Architect 41
3.3.3 SCADE Suite 43
3.3.4 SCADE Test 45
3.3.5 SCADE LifeCycle 45
3.3.6 SCADE for AUTOSAR 46
3.3.7 SCADE-based workflow summary 49
3.4 Takeaway from Using SCADE as a Model-Based Development Environment 50

4 GENERAL TOPICS FOR THE PRODUCT DEVELOPMENT AT THE SOFTWARE LEVEL 51
4.1 Objectives and Work Products 52
4.2 Requirements and Recommendations 52
4.3 Using SCADE for the Product Development at the Software Level 53
4.3.1 Traceability throughout the development process 53
4.3.2 Collaborative software development with SCADE 54
4.3.3 Agile software development with SCADE 55
4.4 Takeaway from Using SCADE for the Product Development at the Software Level 56

5 SPECIFICATION OF SOFTWARE REQUIREMENTS 57
5.1 Objectives and Work Products 58
5.2 Requirements and Recommendations 58
5.3 Specification of Software Requirements with Ansys SCADE, Medini and VRXPERIENCE 59
5.4 Takeaway from Using the Ansys Toolchain to Specify the Software Requirements 61

6 SOFTWARE ARCHITECTURAL DESIGN 62
6.1 Objectives and Work Products 63
6.2 Requirements and Recommendations 63
6.3 Software Architectural Design with SCADE Architect, SCADE Suite, and SCADE LifeCycle 65
6.3.1 Global architectural design 65
6.3.2 Software architectural design with SCADE Architect, SCADE Suite, and SCADE LifeCycle 66
6.4 Takeaway from Using SCADE Architect, SCADE Suite, and SCADE LifeCycle for Software Architectural Design 69

7 SOFTWARE UNIT DESIGN AND IMPLEMENTATION 70
7.1 Objectives and Work Products 71
7.2 Requirements and Recommendations 71
7.3 Software Unit Design with SCADE Suite 72
7.3.1 Filtering and control 73
7.3.2 Decision logic 74
7.3.3 Re-usable components and library management 75
7.3.4 Scade language concepts for re-usability 76
7.3.5 Robustness management 77
7.4 Software Unit Implementation with SCADE Suite KCG 80

7.4.1 Properties of the generated code ... 80
7.4.2 Tuning code to target and project constraints ... 82
7.4.3 Code generation from multiple software units ... 82
7.5 Takeaway from Using SCADE Suite for Software Unit Design and Implementation ... 84

8 SOFTWARE UNIT VERIFICATION ... 85
8.1 Objectives and Work Products ... 86
8.2 Requirements and Recommendations ... 86
8.3 Software Unit Verification with SCADE Suite, SCADE Test Environment for Host, and SCADE LifeCycle ... 88
8.3.1 Model accuracy and consistency ... 89
8.3.2 Compliance of the model with the software requirements ... 89
8.3.3 Compatibility with target computer ... 95
8.3.4 Impact of SCADE Suite KCG code generator qualification ... 98
8.4 Coverage Analysis with SCADE Test Model Coverage ... 101
8.4.1 Using SCADE Test Model Coverage ... 101
8.4.2 Model coverage criteria ... 103
8.5 Takeaway from Using SCADE Suite, SCADE Test, and SCADE LifeCycle for Software Unit Verification ... 108

9 SOFTWARE INTEGRATION AND VERIFICATION ... 109
9.1 Objectives and Work Products ... 110
9.2 Requirements and Recommendations ... 110
9.3 Software Integration with SCADE Suite ... 112
9.3.1 Integration aspects of a SCADE application ... 112
9.3.2 Interface with the external environment ... 113
9.3.3 SCADE Suite module integration ... 113
9.3.4 Integration of external code ... 113
9.3.5 Scheduling and tasking ... 115
9.3.6 Integration of AUTOSAR software components ... 119
9.4 Software Verification with SCADE Test Target Execution and SCADE Test Model Coverage ... 129
9.4.1 Compliance and robustness of the Executable Object Code (EOC) with the software requirements ... 130
9.4.2 Compliance and robustness of the Executable Object Code (EOC) when using library operators ... 131
9.5 Takeaway from Using SCADE Suite and SCADE Test for Software Integration and Verification ... 131

10 TESTING OF THE EMBEDDED SOFTWARE ... 132
10.1 Objectives and Work Products ... 133
10.2 Requirements and recommendations ... 134
10.3 Testing the Embedded Software with SCADE Suite and SCADE test ... 135
10.4 Takeaway from Using SCADE Suite and SCADE Test Target Execution for TESTING the Embedded Software ... 137

11 SUMMARY ... 138

12 APPENDICES 141

APPENDIX A
ACRONYMS AND GLOSSARY 142
A.1 Acronyms 142
A.2 Glossary 144

APPENDIX B
REFERENCES 148

APPENDIX C
COMPLIANCE MATRIX OF SCADE WITH ISO 26262-6:2018 151
C.1 General topics for the product development at the software level (Clause 5) 151
C.2 Specification of software safety requirements (Clause 6) 154
C.3 Software architectural design (Clause 7) 155
C.4 Software unit design and implementation (Clause 8) 161
C.5 Software unit verification (Clause 9) 164
C.6 Software integration and verification (Clause 10) 167
C.7 Testing of the embedded software (Clause 11) 170

APPENDIX D
SCADE SUPPORT OF ASPICE 171
D.1 ASPICE overview 171
D.2 The ASPICE process reference model 171
D.3 Traceability and consistency in ASPICE 172
D.4 The ASPICE capability assessment model 173
D.5 SCADE support of ASPICE 174

APPENDIX E
QUALIFICATION OF SCADE CODE GENERATION AND VERIFICATION TOOLS FOR ISO 26262:2018 184
E.1 Qualification of SCADE Suite KCG 184
E.2 Qualification of SCADE Automotive Code Generator for AUTOSAR (ACG) 185
E.3 Qualification of SCADE LifeCycle Reporter and SCADE LifeCycle Model Change 186
E.4 Qualification of SCADE Test Environment for Host and SCADE Test Target Execution 186
E.5 Qualification of SCADE Test Model Coverage 187

APPENDIX F
SCADE SUITE COMPILER VERIFICATION KIT (CVK) 188
F.1 SCADE Suite CVK overview 188
F.2 SCADE Suite CVK representativity 189

APPENDIX G
TÜV SÜD SCADE CERTIFICATES 193
G.1 SCADE Suite KCG Certificate 193
G.2 SCADE Automotive Code Generator for AUTOSAR (ACG) Certificate 194
G.3 SCADE Test Model Coverage Certificate 195
G.4 SCADE LifeCycle Reporter Certificate 196
G.5 SCADE Test Environment Certificate 197

LIST OF FIGURES

Figure 1: Overview of the ISO 26262:2018 series of standards 17
Figure 2: From item definition to software and hardware requirements 22
Figure 3: Reference phase model for the product development at the software level 23
Figure 4: The scope of ISO 26262-6 24
Figure 5: Example workflow with model-based design and automatic code generation 28
Figure 6: How to read ISO 26262:2018 Tables 30
Figure 7: The application part of the embedded software 32
Figure 8: Control engineering view of a controller 33
Figure 9: A software engineering view 33
Figure 10: Graphical and textual representation of operators 34
Figure 11: Sample of model data flows from an Adaptive Cruise Control (ACC) system 35
Figure 12: Detection of a causality problem 35
Figure 13: Functional expression of concurrency and dependency 36
Figure 14: Detection of a flow initialization problem 36
Figure 15: Initialization of flows 37
Figure 16: A hierarchical state machine 37
Figure 17: Mixed data and control flows in an adaptive cruise control (ACC) 38
Figure 18: The cycle-based execution model of SCADE 39
Figure 19: The SCADE product family 41
Figure 20: SCADE Architect product capabilities 42
Figure 21: medini Analyze product capabilities 42
Figure 22: SCADE in the AUTOSAR flow 43
Figure 23: SCADE Suite product capabilities 44
Figure 24: SCADE Test product capabilities 45
Figure 25: The AUTOSAR three-layer architecture 46
Figure 26: AUTOSAR architecture example – VFB 47
Figure 27: AUTOSAR architecture example – ECU mapping and RTE 48
Figure 28: The SCADE AUTOSAR workflow 49
Figure 29: SCADE-based integrated software workflow 49
Figure 30: Traceability between software requirements and SCADE designs 54
Figure 31: Typical teamwork organization 55
Figure 32: From requirements to deployment with SCADE 56
Figure 33: A multi-disciplinary approach to the creation of an AEB system and its software requirements 59
Figure 34: Software requirements specification of the AEB function 60

Figure 35: The Software Architectural Design process with SCADE 65
Figure 36: Top-level AEB software architecture in SCADE Architect and allocation of software requirements 66
Figure 37: Refinement of AEB function software requirements and allocation to Radar_Tracker component 67
Figure 38: Refinement of the Radar Tracker software requirements and allocation to the leaf components 67
Figure 39: SCADE Architect and SCADE Suite synchronization 68
Figure 40: The AEB software architectural design in SCADE Suite 68
Figure 41: A first order filter 73
Figure 42: Algorithm to iterate each detected cluster of radar points through existing track database 74
Figure 43: A Scade state machine describing the automatic emergency braking (AEB) decision logic 75
Figure 44: Concept of SCADE Suite library 76
Figure 45: Example of a generic operator instantiated with int and bool types 77
Figure 46: Example of an operator parameterized by size 77
Figure 47: Inserting a Confirmator in a Boolean input flow 78
Figure 48: Inserting a Limiter in an output flow 78
Figure 49: Example assumptions for an ACC operator 79
Figure 50: Example of robust architecture 79
Figure 51: The software coding and integration process with SCADE Suite 80
Figure 52: SCADE Suite data flow to generated C source code traceability 81
Figure 53: SCADE Suite state machine to generated C source code traceability 81
Figure 54: Non-expanded and Expanded modes 82
Figure 55: Code generation from multiple components 83
Figure 56: Incremental reviews with SCADE LifeCycle Model Change 90
Figure 57: Positioning of SCADE Test Environment for Host within the verification flow 90
Figure 58: Test cases creation and management in SCADE Test Environment for Host 91
Figure 59: Model-in-the-Loop testing results on host 92
Figure 60: Observer operator containing the safety property 94
Figure 61: Connecting the observer operator to the controller 94
Figure 62: Example a Design Verifier report when a sequence of inputs invalidates the property 94
Figure 63: Example of sequence provided to falsify the property 95
Figure 64: Timing and Stack analysis global visualization 97
Figure 65: Timing verifier analysis reports 97
Figure 66: Positioning of SCADE Test Model Coverage within the verification flow 101
Figure 67: Model coverage analysis and resolution with SCADE Model Test Coverage 102
Figure 68: A Confirmator 102
Figure 69: An Integrator 103
Figure 70: Tag propagation and output observation for SCADE Suite model coverage 104
Figure 71: Tags and observation for Influence 104
Figure 72: Tags and observation for ODC 105
Figure 73: Limiter operator used to limit CruiseSpeed 106
Figure 74: Equivalence classes for Limiter 107

Figure 75: Limiter observer defining equivalence classes criteria 107
Figure 76: Coverage report including equivalence classes 107
Figure 77: Execution semantics of SCADE Suite 115
Figure 78: SCADE Suite code integration 116
Figure 79: Modeling a bi-rate system 117
Figure 80: Timing diagram of a bi-rate system 117
Figure 81: Modeling slow system over fours cycles 118
Figure 82: Timing diagram of distributed computations 118
Figure 83: Explicit read of a VariableDataPrototype in PortPrototype 120
Figure 84: Explicit write of a VariableDataPrototype in PortPrototype 120
Figure 85: List of Server call points for a Runnable 120
Figure 86: Modeling and implementation of the ReadPRAMBlock service 121
Figure 87: PIM and Runnable association 122
Figure 88: PIM synchronization 123
Figure 89: Code generation for an AUTOSAR Software Component 123
Figure 90: Runnable development flow in SCADE Architect and SCADE Suite 124
Figure 91: Multiple read operations 125
Figure 92: Handling multiple reads 125
Figure 93: Parallel outputs 126
Figure 94: Factoring Model-in-the-Loop and target testing with SCADE Test 129
Figure 95: Overview of the SCADE testing process 129
Figure 96: Re-running test cases and procedures with SCADE Test Target Execution 130
Figure 97: Positioning of SCADE Test Target Execution within the verification flow 130
Figure 98: Final model-based integration testing of the AEB application software 135
Figure 99: Setting up a breakpoint in the AEB function model 135
Figure 100: Detailed analysis using the SCADE Suite simulation 136
Figure 101: Rapid Prototyping for AEB radar tracking 136
Figure 102: MiL testing of NCAP AEB CCRm scenario 137
Figure 103: Optimization of the generic model-based development workflow 139
Figure 104: The optimized SCADE model-based workflow 140
Figure 105: Overview of the Automotive SPICE process reference model 171
Figure 106: The ASPICE V model for engineering processes 172
Figure 107: Bidirectional traceability and consistency in ASPICE 172
Figure 108: Role of KCG and CVK in the verification of user development environment 188
Figure 109: Strategy for developing and verifying CVK 190
Figure 110: SCADE Suite CVK in user processes 191
Figure 111: Position of SCADE Suite CVK in the compiler verification process 192

LIST OF TABLES

Table 1: Classes of severity ... 19
Table 2: Classes of probability of exposure ... 20
Table 3: Classes of controllability ... 20
Table 4: ASIL determination ... 21
Table 5: Overview of product development at the software level ... 24
Table 6: Determination of Tool Confidence Level (TCL) ... 29
Table 7: Qualification of software tools classified TCL3 ... 30
Table 8: Topics to be covered by modeling and coding guidelines ... 53
Table 9: Notations for software architectural design ... 64
Table 10: Principles for software architectural design ... 64
Table 11: Methods for verification of the software architectural design ... 65
Table 12: Notations for software unit design ... 72
Table 13: Principles for software unit design and implementation ... 72
Table 14: Methods for software unit verification ... 87
Table 15: Methods for deriving test cases for software unit testing ... 88
Table 16: Structural coverage metrics at the software unit level ... 88
Table 17: Arithmetic error detection performed with SCADE Suite Design Verifier ... 93
Table 18: SCADE Suite KCG application conditions (Installation, use, and Scade modeling) ... 100
Table 19: Coverage criteria in SCADE Test Model Coverage for Scade models ... 105
Table 20: Model to code level coverage implication ... 106
Table 21: Methods for verification of software integration ... 111
Table 22: Methods for deriving test cases for software integration testing ... 112
Table 23: Structural coverage at the software architecture level ... 112
Table 24: SCADE Suite KCG application conditions (Integration) ... 114
Table 26: SCADE Automotive Code Generator for AUTOSAR (ACG) additional application conditions ... 126
Table 27: Test environments for conducting the software testing ... 134
Table 28: Methods for tests of the embedded software ... 134
Table 29: Methods for deriving test cases for the test of the embedded software ... 134
Table 30: Compliance with requirements regarding the software development environment and processes ... 151
Table 31: Compliance with requirements regarding the software development environment and processes ... 152
Table 32: Compliance with topics to be covered by modeling and coding guidelines ... 153
Table 33: Compliance with requirements regarding the software safety requirements ... 154
Table 34: Compliance with requirement regarding the notation for software architectural design ... 155

Table 36: Compliance with requirements regarding the principles for software architectural design 156
Table 37: Compliance with principles for software architectural design 157
Table 38: Compliance with requirements regarding the scope of the software architectural design 158
Table 39: Compliance with requirement for the verification of the software architectural design 160
Table 40: Compliance with methods for the verification of the software architectural design 160
Table 41: Compliance with generic requirements for software unit design and implementation 161
Table 42: Compliance with notation for software unit design 161
Table 43: Compliance with properties of software unit designs 162
Table 44: Compliance with principles for software unit design and implementation 163
Table 45: Compliance with generic requirements for software unit verification 164
Table 46: Compliance with methods for software unit verification 165
Table 47: Compliance with methods for deriving test cases for software unit testing 166
Table 48: Compliance with structural coverage metrics at the software unit level 166
Table 49: Compliance with requirements for the test environment for software unit testing 166
Table 50: Compliance with generic requirements for software integration and verification 167
Table 51: Compliance with methods for verification of software integration 168
Table 52: Compliance with methods for deriving test cases for software integration testing 168
Table 53: Compliance with structural coverage at the software architecture level 169
Table 54: Compliance with requirement regarding unspecified functions as part of the embedded software 169
Table 55: Compliance with requirement regarding the test environment for software integration testing 169
Table 56: Compliance with test environments for conducting the software testing 170
Table 57: Compliance with methods for tests of the embedded software 170
Table 58: Compliance with methods for deriving test cases for the test of the embedded software 170
Table 59: Process capability levels according to ISO/IEC 33020 173
Table 60: Process attributes according to ISO/IEC 33020 173
Table 61: SCADE support of ASPICE 174

1

DOCUMENT BACKGROUND, OBJECTIVES, AND SCOPE

1.1 Background

Currently, numerous people play a role in defining and creating safety-related systems for the automotive industry. The functions and architecture of a system are defined by system engineers using some informal notation. The functional safety of the system is analyzed by safety engineers. The embedded production software is often specified textually and hand-coded by software engineers in the coding language.

In this context, the support of a model-based qualified toolchain, including but not limited to qualified code generation, carries strong Return on Investment (ROI), while preserving the safety of the application.

Basically, the idea is to describe the application through a software model and to automatically generate the code from this model using a code generator that has been qualified with respect to [ISO 26262:2018].

This method has several advantages for the development life cycle when a proper modeling approach is defined:

- It fulfills the needs of software engineers by supporting an accurate specification of the software and by providing efficient automatic code generation of software having the qualities expected for such applications (i.e., efficiency, determinism, static memory allocation, etc.).
- It allows for establishing efficient processes to ensure that ISO 26262:2018 requirements are met.
- It saves coding time, as this is automatic.
- It saves a significant part of the verification time, as the use of such tools guarantees that the generated source code conforms to the software model and agrees with necessary coding standards, thus removing the need to perform back-to-back comparison test between model and code, and/or code reviews.
- It allows for identifying problems earlier in the development cycle, since most of the verification activities, incl. reviews, analyses, and testing, can be carried out at model level.
- It reduces the change cycle time since modifications can be done at model level and code can automatically be regenerated.

1.2 Objectives and Scope

This handbook provides a careful explanation of an [ISO 26262:2018] compliant software life cycle, as described in Part 6 of the ISO 26262:2018 series of standards (noted ISO 26262-6:2018). The rest of the document explains how the use of proper modeling techniques and qualified code generation from models can drastically improve productivity in the development and verification of safety-related software.

It is organized as follows:

Chapter 2 introduces the ISO 26262:2018 series of standards used when developing embedded automotive systems and software. It also addresses the ways to get “Confidence in the use of software tools” as described in Part 8 of ISO 26262:2018 (noted ISO 26262-8:2018).

Chapter 3 presents an overview of the SCADE Suite methodology and tools, including how this solution achieves the highest quality standards while reducing costs thanks to model-based development and verification, with a strong emphasis on the following points:

- A unique and accurate software description, which enables the prevention of many specification and design errors, and can be shared among all project participants
- Early identification of design errors, making it possible to fix them in requirements/design phase rather than in the testing and integration phases
- Qualified code generation that not only saves the cost of writing the code by hand, but also the cost of verifying it
- Automation of verification activities relying on a set of qualified SCADE testing and lifecycle management tools

Chapters 4, 5, 6, 7, 8, 9, and 10 present the development and verification activities to be performed when using SCADE tools while complying to Clauses 5 to 11 of ISO 26262-6:2018:

- General topics for the product development at the software level
- Specification of software requirements
- Software architectural design
- Software unit design and implementation
- Software unit verification
- Software integration and verification
- Testing of the embedded software

Chapter 11 provides a summary.

Finally, **Appendixes A–G** detail the following topics:

A) acronyms used in this handbook and glossary for key terminology

B) list of references

C) compliance of SCADE with ISO 26262-6:2018

D) description of ASPICE support from SCADE

E) qualification process of the SCADE code generators and verification tools

F) SCADE Suite Compiler Verification Kit

G) TÜV SÜD certificates for SCADE tools qualification

The concepts and methodology described in this handbook are applicable starting from the following product configuration (and onwards): Ansys SCADE 2021 R2, with SCADE Suite KCG 6.6.2 and SCADE ACG 2.1.

2

DEVELOPMENT OF SAFETY-RELATED AUTOMOTIVE SOFTWARE

2.1 Introduction to the ISO 26262:2018 Series of Standards

The ISO 26262:2018 series of standards is the adaptation of the [IEC 61508] standard for the automotive industry. It sets out the automotive approach for all safety lifecycle activities for safety relevant systems comprised of electrical and/or electronic (E/E) components.

It addresses possible hazards caused by functional behavior of E/E safety-related systems due to malfunctions. It does not address non-functional hazards due to technical realization (for example, electric shock, fire, smoke, heat, etc.) and it does not address nominal performance of E/E systems.

Safety is one of the key issues of automobile development. Functionality in driver assistance and autonomy, electrification, but also in vehicle dynamics control and active and passive safety systems increasingly touches the domain of safety engineering. Development and integration of these functions requires a safe system development process.

In most situations, safety is achieved by several protective systems, which rely on many technologies (for example, mechanical, hydraulic, pneumatic, electrical, electronic, programmable electronic…). Any safety strategy must therefore consider not only all the elements within an individual system (for example sensors, controlling devices, and actuators) but also the combination of all safety-related systems. Therefore, while ISO 26262 is concerned with E/E safety-related systems, it may also provide a framework which participates to the achievement of safety at the vehicle level.

This International Standard:

- adopts a customer risk-based approach for the determination of the risks
- provides an automotive specific method to identify the safety integrity level of each hazardous event (potential source of harm caused by malfunctioning behavior of the system in a particular operational situation)
- uses safety integrity levels to specify the additional activities to be performed during the development of the E/E system to ensure its safety
- provides requirements for the whole lifecycle of E/E (engineering, production, operation, maintenance, decommissioning) necessary to achieve the required functional safety for E/E which are linked to the safety integrity levels

Functional safety is an attribute of any vehicle system or functionality and is defined and affected during all phases of the safety lifecycle. It can be influenced and measured by safety-related activities that include design and development activities like testing, validation, evaluation, and also, conformity of production and configuration, as well as management activities and personal responsibilities.

Figure 1 shows the overall framework of this International Standard

Source: ISO 26262-1:2018

1. VOCABULARY

2. MANAGEMENT OF FUNCTIONAL SAFETY

- **2–5** Overall safety management
- **2–6** Project dependent safety management
- **2–7** Safety management regarding production, operation, service and decommissioning

3. CONCEPT PHASE

- **3–5** Item definition
- **3–6** Hazard analysis and risk assesment
- **3–7** Functional safety concept

4. PRODUCT DEVELOPMENT AT THE SYSTEM LEVEL

- **4–5** General topics for the product development at the system level
- **4–6** Technical safety concept
- **4–7** System and item itegration and testing
- **4–8** Safety validation

7. PRODUCTION, OPERATION, SERVICE AND DECOMMISSIONING

- **7–5** Planning for production, operation, service and decommissioning
- **7–6** Production
- **7–7** Operation, service, and decommissioning

12. ADAPTATION OF ISO 26262 FOR MOTORCYCLES

- **12–5** General topics for adaptation for motorcycles
- **12–6** Safety culture
- **12–7** Confirmation measures
- **12–8** Hazard analysis and risk assessment
- **12–9** Vechicle integration and testing
- **12–10** Safety validation

5. PRODUCT DEVELOPMENT AT THE HARDWARE LEVEL

- **5–5** General topics for the product development at the hardware level
- **5–6** Specification of hardware safety requirements
- **5–7** Hardware design
- **5–8** Evaluation of the hardware architectural metrics
- **5–9** Evaluation of safety goal violations due to random hardware failures
- **5–10** Hardware integration and verification

6. PRODUCT DEVELOPMENT AT THE SOFTWARE LEVEL

- **6–5** General topics for the product development at the software level
- **6–6** Specification of software safety requirements
- **6–7** Software architectural design
- **6–8** Software unit design and implementation
- **6–10** Software intergration and verification
- **6–11** Testing of the embedded software

8. SUPPORTING PROCESSES

- **8–5** Interfaces within distributed developments
- **8–6** Specification and management of safety requirements
- **8–7** Configuration management
- **8–8** Change management
- **8–9** Verification
- **8–10** Document management
- **8–11** Confidence in the use of sofware tools
- **8–12** Qualification of software components
- **8–13** Evaluation of hardware elements
- **8–14** Proven in use argument
- **8–15** Interfacing an application that is out of scope of ISO 26262
- **8–16** Integration of safety-related systems not developed according to ISO 26262

9. AUTOMOTIVE SAFETY INTEGRITY LEVEL (ASIL)-ORIENTED AND SAFETY-ORIENTED ANALYSES

- **9–5** Requirements decomposition with respect to ASIL tailoring
- **9–6** Criteria for coexistence of elements
- **9–7** Analysis of dependent failures
- **9–8** Safety analyses

10. GUIDELINES ON ISO 26262

11. GUIDELINES ON APPLICATION OF ISO 26262 TO SEMICONDUCTORS

FIGURE 1: OVERVIEW OF THE ISO 26262:2018 SERIES OF STANDARDS

2.2 Hazard Analysis and Risk Assessment (HARA) in ISO 26262-3:2018

In this Section, Part 3 of ISO 26262:2018 (Concept Phase) is referenced as ISO 26262-3:2018, and we concentrate on the first two topics of Part 3:

- item definition
- hazard analysis and risk assessment

An **item** is *a system or a combination of systems, to which ISO 26262:2018 is applied, that implements a function or part of a function at the vehicle level*. As stated in Clause 6 of the ISO 26262-3:2018 standard, the objective of this phase is:

1. to systematically identify all hazards caused by malfunctioning behavior of the item
2. to classify them by their criticality (as explained below)
3. to formulate the safety goals (top-level safety requirements) with corresponding ASILs suitable for their prevention or mitigation

For this, the item is evaluated regarding its safety implications.

The **Automotive Safety Integrity Level (ASIL)** is determined by a systematic evaluation of potentially hazardous driving or operating situations. The rationale for the ASIL evaluation is documented and considers the estimation of the impact factors: severity, probability of exposure, and controllability.

The hazard analysis and risk assessment method comprises three steps:

Situation analysis and hazard identification: The goal of situation analysis and hazard identification is to identify the operational situations and operating modes in which an item's malfunctioning behavior will result in a hazardous event.

1. **Hazard classification:** The goal of hazard classification is to determine for each hazardous event and operational situation considered the classes of:
    - **probability of exposure (E):** exposure to a certain situation in which a malfunction could lead to harm (not probability of the malfunction)
    - **controllability (C):** controllability by humans or external mechanisms (not by the E/E-based safety mechanisms to be built into the item during the safety activities)
    - **severity (S):** the severity of the resulting harm to humans if the hazard actually leads to an accident
2. **ASIL Assignment:** The goal of ASIL assignment is to determine the automotive safety integrity level (ASIL) for each hazardous event.

2.2.1 Situation analysis and hazard identification

According to the ISO 26262:2018 hazard model, a "hazardous event" is defined as combination of a driving or operation situation with a vehicle-level malfunctioning behavior which, in this situation, can potentially lead to harm. So, the principal work of performing a HARA is, according to Clause 6.4.2.1 of ISO 26262-3:2018, "The operational situations and operating modes in which an item's malfunctioning behavior will result in a hazardous event shall be described; both when the vehicle is correctly used and when it is incorrectly used in a reasonably foreseeable way." To this end, as a preparation, a systematic catalog of driving and operating situations must be established, considering factors like road type, usage or maneuver, weather, road conditions, visibility and presence and behavior of other traffic participants.

Using a systematic approach like FMEA or HAZOP, the potential malfunctioning behaviors that can lead to hazards are identified.

Combining the operational situations and operating modes with the malfunctioning behaviors previously identified (the hazardous events are the combinations), the relevant hazardous events shall be determined as well as their consequences.

In situation analysis and hazard identification only the item to be developed shall be evaluated, *i.e.*, risk-reducing measures within the item that are intended to be implemented or have already been implemented in predecessor systems shall not be considered.

If hazards that are not related to malfunctions of the E/E part of a system (*e.g.*, release of toxic material due to constructive weaknesses) are identified which are outside the scope of this International Standard then they shall be documented, and these must be addressed based on organization specific procedures.

The persons undertaking situation analysis and hazard identification shall include those with a good knowledge and domain experience of the behavior of the possible components, and of the way a vehicle and its driver can behave.

2.2.2 Hazardous event classification

Potential severity S, probability of exposure in the driving situations E, and controllability C shall be estimated using a qualitative approach.

In hazardous event classification, as with hazard identification, only the item to be developed shall be evaluated.

The scheme shall be applied to all hazardous events identified during the previous step (situation analysis and hazard identification).

2.2.3 Estimation of potential severity

The severity of potential harm shall be estimated in accordance with Table 1.

The severity class S0 shall be used if the hazard analysis determines that the consequences of a failure mode are clearly limited to material damage and do not involve harm to persons. If a hazard is assigned to hazard class S0, no ASIL assignment is required.

TABLE 1: CLASSES OF SEVERITY

Source: Table 1 in ISO 26262-3:2018

	Class			
	S0	**S1**	**S2**	**S3**
Description	No injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-threatening injuries (survival uncertain), fatal injuries

2.2.4 Estimation of the probability of exposure regarding operational situations

To classify the probabilities of exposure in the driving and operational situations, the estimation parameter E shall be used.

The proportion of vehicles equipped with the item shall not be considered for the estimation of the probability of exposure.

The probability of exposure of the driving and operational situations shall be classified in accordance with Table 2. If a hazardous event is assigned exposure class E0, no ASIL assignment is required.

TABLE 2: CLASSES OF PROBABILITY OF EXPOSURE

Source: Table 2 in ISO 26262-3:2018

	Class				
	E0	**E1**	**E2**	**E3**	**E4**
Description	Incredible	Very low probability	Low probability	Medium probability	High Probability

2.2.5 Estimation of controllability

The controllability by the driver or other traffic participants shall be classified in accordance with Table 3.

For situations which are regarded as simply distracting or disturbing but as controllable in general, the class C0 may be used. No ASIL assignment is required for situations that are assigned to class C0.

TABLE 3: CLASSES OF CONTROLLABILITY

Source: Table 3 in ISO 26262-3:2018

	Class			
	C0	**C1**	**C2**	**C3**
Description	Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable

2.2.6 ASIL assignment

The Automotive Safety Integrity Level (ASIL) shall be determined for each hazardous event using the estimation parameters severity (S), probability of exposure (E) and controllability (C) in accordance with Table 4:

- Four ASILs are defined: ASIL A, ASIL B, ASIL C and ASIL D, whereas ASIL A implies low safety requirements and ASIL D implies high safety requirements.
- In addition to these safety-related levels, there is also the class QM, which stands for Quality Management, i.e., quality processes are sufficient to manage the identified risk.

The result of the ASIL assignment shall be documented and shall include at least the following information:

- driving situations with severity
- probability of exposure
- controllability
- and the resulting ASIL

TABLE 4: ASIL DETERMINATION

Source: Table 4 in ISO 26262-3:2018

Severity Class	Exposure Class	Controllability Class		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

2.3 From Hazardous Events in ISO 26262-3:2018 to Technical Safety Requirements in ISO 26262-4:2018

After identifying the hazardous events and assessing their ASIL within the HARA, it is time to start treating them, still in ISO 26262-3:2018 (Concept phase).

Safety Goals (SG) will be derived from those hazardous events as top-level safety requirements at vehicle level. Once the safety goals identified, some safety analysis such as Fault Tree Analysis (FTA) could be used at system level to identify the hazardous event root causes which could lead to the violation of a safety goal; this analysis helps to establish a Functional Safety Concept (FSC).

The FSC aims to specify the safety measures required to address the effects of relevant faults and allocate functional safety requirements to the system architectural design or external measures.

Continuing with ISO 26262-4:2018 (Product development at the system level), these Functional Safety Requirements (FSR) will be the input to establish the Technical Safety Concept (TSC) where the FSR will be refined and allocated to the elements of the system architecture or external measures, either with the ASIL inherited from FSR or decomposed ASIL, if appropriate according to ISO 26262-9:2018 (Automotive safety integrity level-oriented and safety-oriented analyses).

The result of the Technical Safety Concept is a set of Technical Safety Requirements (TSR) allocated to the system's elements. Each element inherits TSR from one or several systems to which it contributes. These element's TSR will be then refined down to hardware leading to the specification of hardware safety requirements, as defined in ISO 26262-5:2018 (Product development at the hardware level), and to software leading to the specification of software safety requirements, as defined in ISO 26262-6:2018 (Product development at the software level).

A summary of the steps from Item definition down to Allocation of the Technical Safety Requirements to Hardware and Software is given in Figure 2.

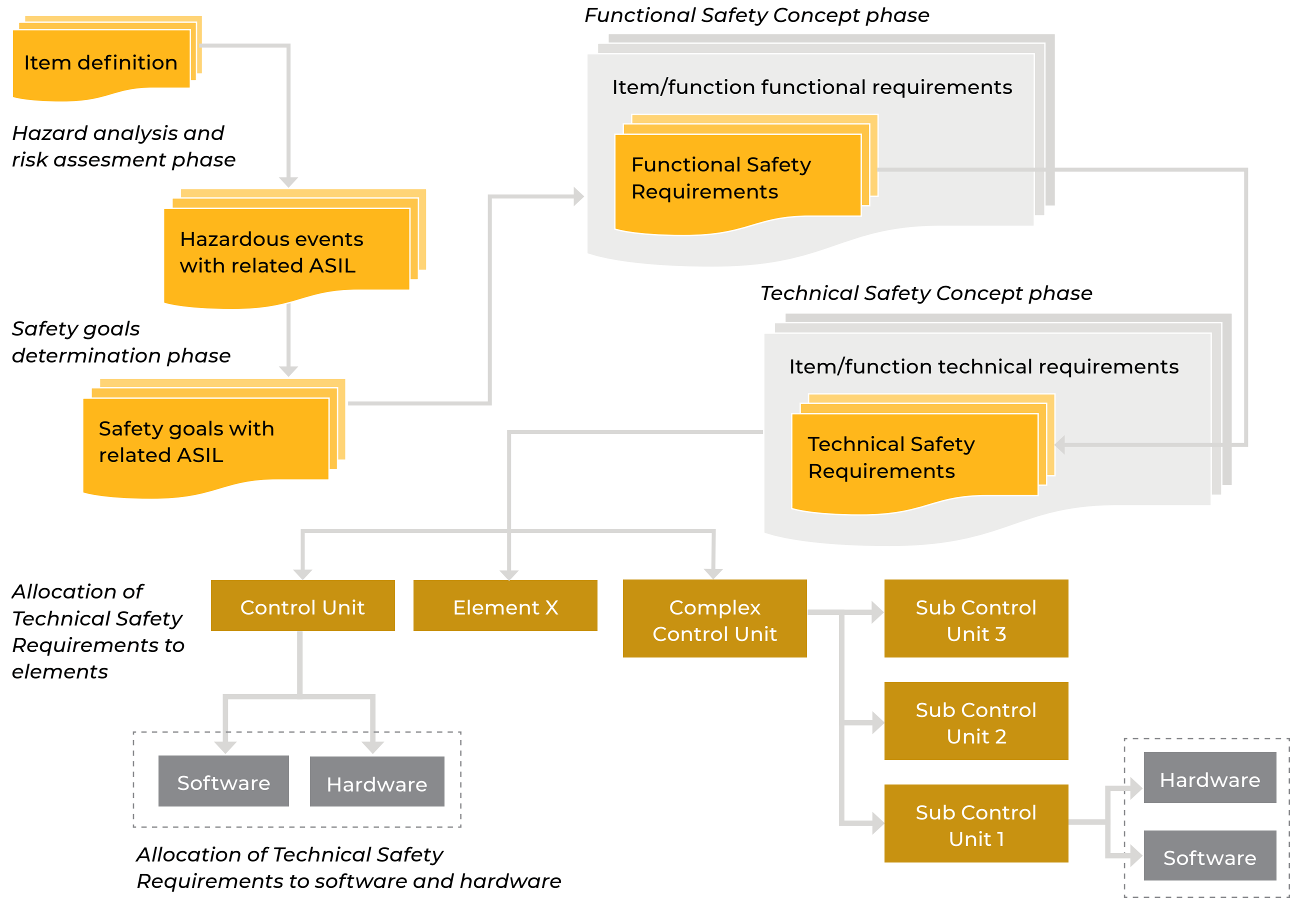


FIGURE 2: FROM ITEM DEFINITION TO SOFTWARE AND HARDWARE REQUIREMENTS

2.4 Software Development Process Overview in ISO 26262-6:2018

As stated above, the Technical Safety Concept (TSC) provides inputs into the software development process of ISO 26262-6:2018, which we now consider.

After a general introduction defining the terms and organization, the [ISO 26262-6:2018] document is structured in Clauses (5 to 11). For each clause there is a definition of the objectives, inputs, requirements and recommendations, and work products. When appropriate, these requirements and recommendations are further described in Tables (1 to 15).

2.4.1 Reference phase model for the product development at the software level

Clause 5 describes how to initiate product development at the software level. The following Figure 3 provides a reference phase model for software development.

Source: Figure 2 in ISO 26262-6:2018

DESIGN PHASE VERIFICATION
UNIT VERIFICATION
DESIGN PHASES
TEST PHASES
SCOPE OF PART 4
SYSTEM AND ITEM VERIFICATION
4-6 Technical safety concept
4-7 System and item integration and testing
SCOPE OF PART 4
SOFTWARE TESTING
6-6 Specification of software safety requirements
6-11 Testing of the embedded software
SCOPE OF PART 6
SCOPE OF PART 6
SOFTWARE VERIFICATION
6-7 Software architectural design
6-10 Software integration and verification
6-8 Software unit design and implementation
6-9 Software unit verification
UNIT VERIFICATION

FIGURE 3: REFERENCE PHASE MODEL FOR THE PRODUCT DEVELOPMENT AT THE SOFTWARE LEVEL

In the Figure above, the specific clauses of each part of the ISO 26262:2018 series of standards are indicated as m-n. For example, 4-7 represents Clause 7 of ISO 26262-4:2018.

2.4.2 Objectives, prerequisites, and work products of each sub-phase

As stated above, for each of the sub-phases in [ISO 26262-6:2018], the standard specifies:

- inputs
- requirements and recommendations, including tables related to notations, principles and methods
- work products

Typically, the standard applies to an existing development process, which is supported by tools, and which is described in internal guidelines, and it is adding constraints to these guidelines to ensure safety requirements are met. Overall, the methods, tools and guidelines contribute to the confidence level in achieving each ISO 26262-6 requirements and recommendations, as depicted in Figure 4 below.

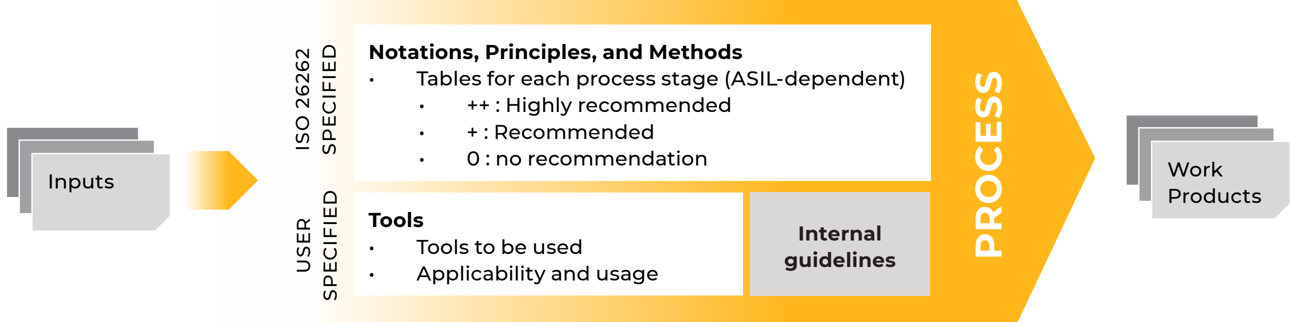


FIGURE 4: THE SCOPE OF ISO 26262-6

The following Table 5 provides a summary of the objectives, inputs, and work products of the phases of product development at the software level.

TABLE 5: OVERVIEW OF PRODUCT DEVELOPMENT AT THE SOFTWARE LEVEL[1]

Source: Table A.1 in ISO 26262-6:2018

Clause	Objectives	Prerequisites	Work products
5 **General topics for the product development at the software level**	The objectives of this Clause are: a) to ensure a suitable and consistent software development process; and b) to ensure a suitable software development environment.	(none)	5.5.1 Documentation of the software development environment
6 **Specification of software safety requirements**	The objectives of this sub-phase are: a) to specify or refine the software safety requirements which are derived from the technical safety concept and the system architectural design specification; b) to define the safety-related functionalities and properties of the software required for the implementation; c) to refine the requirements of the hardware-software interface initiated in ISO 26262-4:2018, Clause 6; and d) to verify that the software safety requirements and the hardware-software interface requirements are suitable for software development and are consistent with the technical safety concept and the system architectural design specification	Technical safety requirements specification (see ISO 26262-4:2018, 6.5.1) Technical safety concept (see ISO 26262-4:2018, 6.5.2) System architectural design specification (see ISO 26262-4:2018, 6.5.3) Hardware-software interface (HSI) specification (see ISO 26262-4:2018, 6.5.4) Documentation of the software development environment (see 5.5.1)	6.5.1 Software safety requirements specification 6.5.2 Hardware-software interface (HSI) specification (refined) 6.5.3 Software verification report

[1] *Note that references in this Table and the following other ISO 26262-6 Tables are referencing sections in this standard.*

Clause	Objectives	Prerequisites	Work products
7 **Software Architectural design**	The objectives of this sub-phase are: a) to develop a software architectural design that satisfies the software safety requirements and the other software requirements; b) to verify that the software architectural design is suitable to satisfy the software safety requirements with the required ASIL; and c) to support the implementation and verification of the software.	Documentation of the software development environment (see 5.5.1) Hardware-software interface (HSI) specification (refined) (see 6.5.2) Software safety requirements specification (see 6.5.1)	7.5.1 Software architectural design specification 7.5.2 Safety analysis report 7.5.3 Dependent failures analysis report 7.5.4 Software verification report
8 **Software unit design and implementation**	The objectives of this sub-phase are: a) to develop a software unit design in accordance with the software architectural design, the design criteria and the associated software requirements which supports the implementation and verification of the software unit; and b) to implement the software units as specified.	Documentation of the software development environment (see 5.5.1) Hardware-software interface (HSI) specification (refined) (see 6.5.2) Software architectural design specification (see 7.5.1) Software safety requirements specification (see 6.5.1) Configuration data (see C.5.3), if applicable Calibration data (see C.5.4), if applicable	8.5.1 Software unit design specification 8.5.2 Software unit implementation

Clause	Objectives	Prerequisites	Work products
9 **Software unit verification**	The objectives of this sub-phase are: a) to provide evidence that the software unit design satisfies the allocated software requirements and is suitable for the implementation; b) to verify that the defined safety measures resulting from safety analyses in accordance with 7.4.10 and 7.4.11 are properly implemented; c) to provide evidence that the implemented software unit complies with the unit design and fulfils the allocated software requirements with the required ASIL; and d) to provide sufficient evidence that the software unit contains neither undesired functionalities nor undesired properties regarding functional safety.	Hardware-software interface (HSI) specification (refined) (see 6.5.2) Software architectural design specification (see 7.5.1) Software unit design specification (see 8.5.1) Software unit implementation (see 8.5.2) Configuration data (see C.5.3), if applicable Calibration data (see C .5.4), if applicable Safety analysis report (see 7.5.2) Documentation of the software development environment (see 5.5.1)	9.5.1 Software verification specification 9.5.2 Software verification report (refined)
10 **Software integration and verification**	The objectives of this sub-phase are: a) to define the integration steps and integrate the software elements until the embedded software is fully integrated; b) to verify that the defined safety measures resulting from safety analyses at the software architectural level are properly implemented; c) to provide evidence that the integrated software units and software components fulfil their requirements according to the software architectural design; and d) to provide sufficient evidence that the integrated software contains neither undesired functionalities nor undesired properties regarding functional safety.	Hardware-software interface (HSI) specification (refined) (6.5.2) Software architectural design specification (see 7.5.1) Safety analysis report (see 7.5.2) Dependent failures analysis report (see 7.5.3), if applicable Software unit implementation (see 8.5.2) Configuration data (see C.5.3), if applicable Calibration data (see C.5.4), if applicable Documentation of the development environment (see 5.5.1) Software verification specification (see 9.5.1)	10.5.1 Software verification specification (refined) 10.5.2 Embedded software 10.5.3 Software verification report (refined)

Clause	Objectives	Prerequisites	Work products
11 **Testing of the embedded software**	The objectives of this sub-phase are to provide evidence that the embedded software: a) fulfils the software safety requirements when executed in the target environment; and b) contains neither undesired functionalities nor undesired properties regarding functional safety.	Software architectural design specification (see 7.5.1) Software safety requirements specification (see 6.5.1) Embedded software (see 10.5.2) Calibration data (see C.5.4), if applicable Documentation of the software development environment (see 5.5.1) Software verification specification (refined) (see 10.5.1)	11.5.1 Software verification specification (refined) 11.5.2 Software verification report (refined)
Annex C **Software configuration**	The objectives of software configuration are: a) to enable controlled changes in the behaviour of the software for different applications; b) to provide evidence that the configuration data and the calibration data fulfil the requirements with the required ASIL; and c) to provide evidence that the application-specific embedded software and its calibration data are suitable for release for production.	See applicable prerequisites of the relevant phases of the safety lifecycle in which software configuration is applied.	C.5.1 Configuration data specification C.5.2 Calibration data specification C.5.3 Configuration data C.5.4 Calibration data C.5.6 Verification specification (refined) C.5.7 Verification report (refined) C.5.8 Software architectural design specification (refined) C.5.9 Documentation of the software development environment (refined)

2.4.3 Model-based development (MBD) approaches in ISO 26262-6:2018

ISO 26262-6 considers the possible usage benefits and potential issues of model-based development approaches (MBD) for software development (see Annex B of [ISO 26262-6:2018]).

In this Annex B, the following uses cases of MBD are considered:

- Specification of software safety requirements
- Development of the software architectural design
- Design and implementation the software units (with or without automated code generation) and their verification and integration

In the Figure below, we consider the use of a model-based approach to represent software design (including software architectural design and software units design), coming together with automatic code generation from the design models.

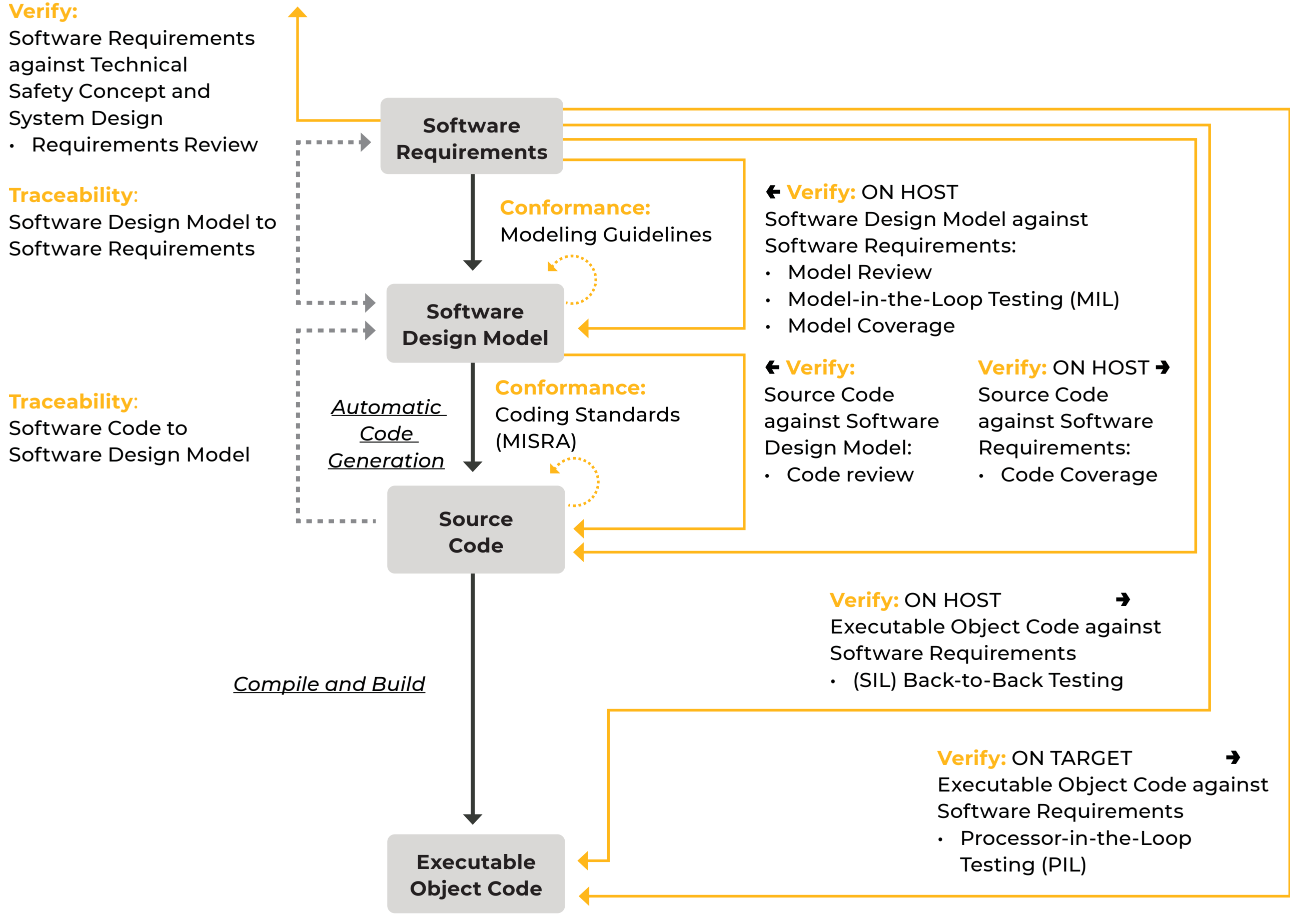


FIGURE 5: EXAMPLE WORKFLOW WITH MODEL-BASED DESIGN AND AUTOMATIC CODE GENERATION[2]

In this workflow, we consider the following four levels of representation of the software:

- Software requirements
- Software design model (incl. software architectural design and detailed design of the software units)
- Source code that is automatically generated from the design models
- Executable object code (for host and target)

Associated to each of these levels there are several verification activities that have been defined according to the activities described in Table 5 above:

- requirements review (Clause 6d)
- design models review (Clause 7b, 9a)
- Model-in-the-Loop testing (Clause 9a, 9b)
- code reviews (Clause 9c)
- etc.

In the following chapters of this handbook (Chapters 3 to 11), we will consider a significant improvement of the above workflow, with **automatic and qualified code generation** based on using a **formally defined language** for software architectural and detailed design.

[2] *We assume here that there are two versions of the "Executable Object Code", one that can run on the Host computer and one that will run on the Target computer.*

2.5 Confidence in the Use of Software Tools in ISO 26262-8:2018

Part 8 of [ISO-26262:2018] describes the requirements for supporting processes including safety requirements management, configuration and change management, verification, documentation, confidence in the use software tools, etc.

Let us now present the topic of "confidence in the use of software tools", as tool qualification is key to reducing the cost of developing and verifying safety-related embedded software.

2.5.1 Required level of confidence in a software tool

Clause 11 of [ISO-26262-8] provides criteria to determine the required level of confidence in a software tool and the means of qualification for a software tool so that tool users can rely on the correct functioning of the tool as it is used for achieving activities required by ISO 26262:2018.

The intended usage of the software tool must be analyzed to determine the Tool Impact (TI):

- TI1 shall be selected when there is an argument that there is no possibility that a malfunction of the software tool can introduce or fail to detect an error in the safety-related software being developed.
- TI2 shall be selected otherwise.

The confidence in measures that prevent the software tool from malfunctioning or that detect the tool has malfunctioned is expressed by the Tool error Detection (TD) class:

- TD1 shall be selected if there is a high degree of confidence that a malfunction can be prevented or detected.
- TD2 shall be selected if the degree is medium.
- TD3 shall be selected otherwise.

Based on TI and TD, the tool confidence level (TCL) is determined on Table 6 below.

TABLE 6: DETERMINATION OF TOOL CONFIDENCE LEVEL (TCL)

Source: Table 3 in ISO 26262-8:2018

		Tool error Detection		
		TD1	**TD2**	**TD3**
Tool Impact	**TI1**	TCL1	TCL1	TCL1
	TI2	TCL1	TCL2	TCL3

For example, for a code generator producing source code from a software model and in the case the output of the code generator (the source code) is not verified, classes TI2 (tool may introduce an error) and TD3 (malfunction may not be detected) must be selected and therefore the tool confidence level is TCL3. For TCL1 tools, no qualification is required.

2.5.2 Possible methods to qualify a tool

If we consider this example, Clause 11 of ISO 26262-8 defines in Table 7 below the possible methods to qualify the tool as a function of the ASIL of the safety-related item that is being developed.

TABLE 7: QUALIFICATION OF SOFTWARE TOOLS CLASSIFIED TCL3

Source: Table 4 in ISO 26262-8:2018

Methods		ASIL			
		A	B	C	D
1a	Increased confidence from use in accordance with 11.4.7	++	++	+	+
1b	Evaluation of the tool development process in accordance with 11.4.8	++	++	+	+
1c	Validation of the software tool in accordance with 11.4.9	+	+	++	++
1d	Development in accordance with a safety standard[a]	+	+	++	++

[a] *No safety standard is fully applicable to the development of software tools. Instead, a relevant subset of requirements of the safety standard can be selected.*

EXAMPLE: Development of the software tool in accordance with ISO 26262, IEC 61508 or RTCA DO-178

Interpretation of this Table, and all similar Tables in the rest of this handbook, is given in Figure 6 below.

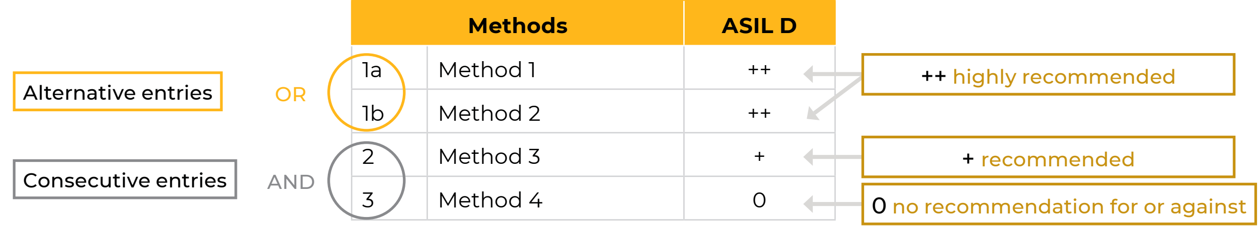


FIGURE 6: HOW TO READ ISO 26262:2018 TABLES

Therefore, if we assume the embedded software is ASIL D, there are two highly recommended (++) methods in Table 7 from which one must be chosen for qualifying the code generation tool:

- validation of the code generator in accordance with 11.4.9 which demonstrates that the tool complies with its requirements, typically by running a test suite that evaluates the functional and non-functional aspects of the tool
- development of the code generator in accordance with a safety standard (*e. g.*, ISO 26262, IEC 61508, DO-178C).

As required by Section 11.5 of [ISO 26262-8], work products of the qualification of a software tool include:

1. **Software tool criteria evaluation report,** based on TI and TD
2. **Software tool qualification report,** based on the method that is chosen to qualify the tool

3

MODEL-BASED DEVELOPMENT WITH SCADE

3.1 What is SCADE?

3.1.1 SCADE origin and application domain

SCADE is a product family that includes the following product lines:

- SCADE Architect for the analysis and design of system and software architectures
- SCADE Suite for the design of embedded control applications
- SCADE Display for the design of embedded displays
- SCADE Test for the dynamic verification of the design models
- SCADE LifeCycle for the application life cycle management of these applications

The name SCADE stands for “Safety-Critical Application Development Environment”. When spelled Scade it refers to the language on which SCADE Suite is based.

In its early academic inception, the Scade language was designed for the development of safety-related software. It relies on the theory of languages for real-time applications and, more particularly, on the Lustre and Esterel languages as described in [Lustre] and [Esterel]. The Scade language has evolved from this base and currently is a formal notation spanning a full set of features needed to model complex, hard real-time, critical applications [Scade 6].

SCADE Suite addresses the application part of the embedded software, as illustrated in Figure 7. This is usually the most complex and changeable aspect of software. It typically represents 60 to 90 percent of the embedded software.

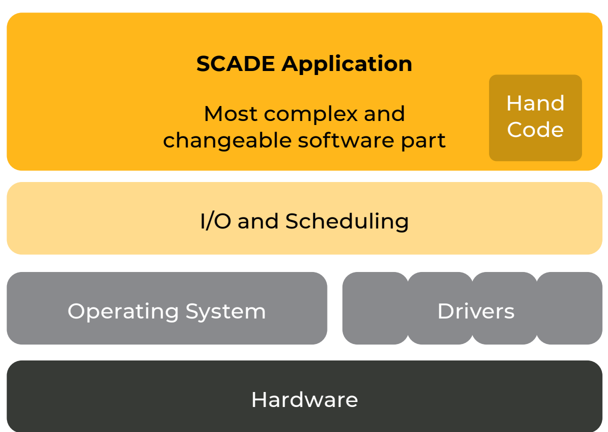


FIGURE 7: THE APPLICATION PART OF THE EMBEDDED SOFTWARE

3.1.2 SCADE as a bridge between control and software engineering

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

- Control engineers describe systems and their controllers using block diagrams and transfer functions (s form for continuous time, z form for discrete time), as shown in Figure 8.

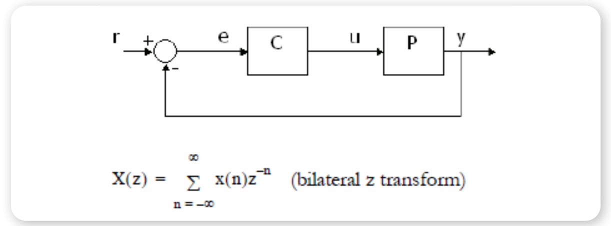


FIGURE 8: CONTROL ENGINEERING VIEW OF A CONTROLLER

- Software engineers describe their programs in terms of tasks, flow charts, and algorithms, as shown below in Figure 9.

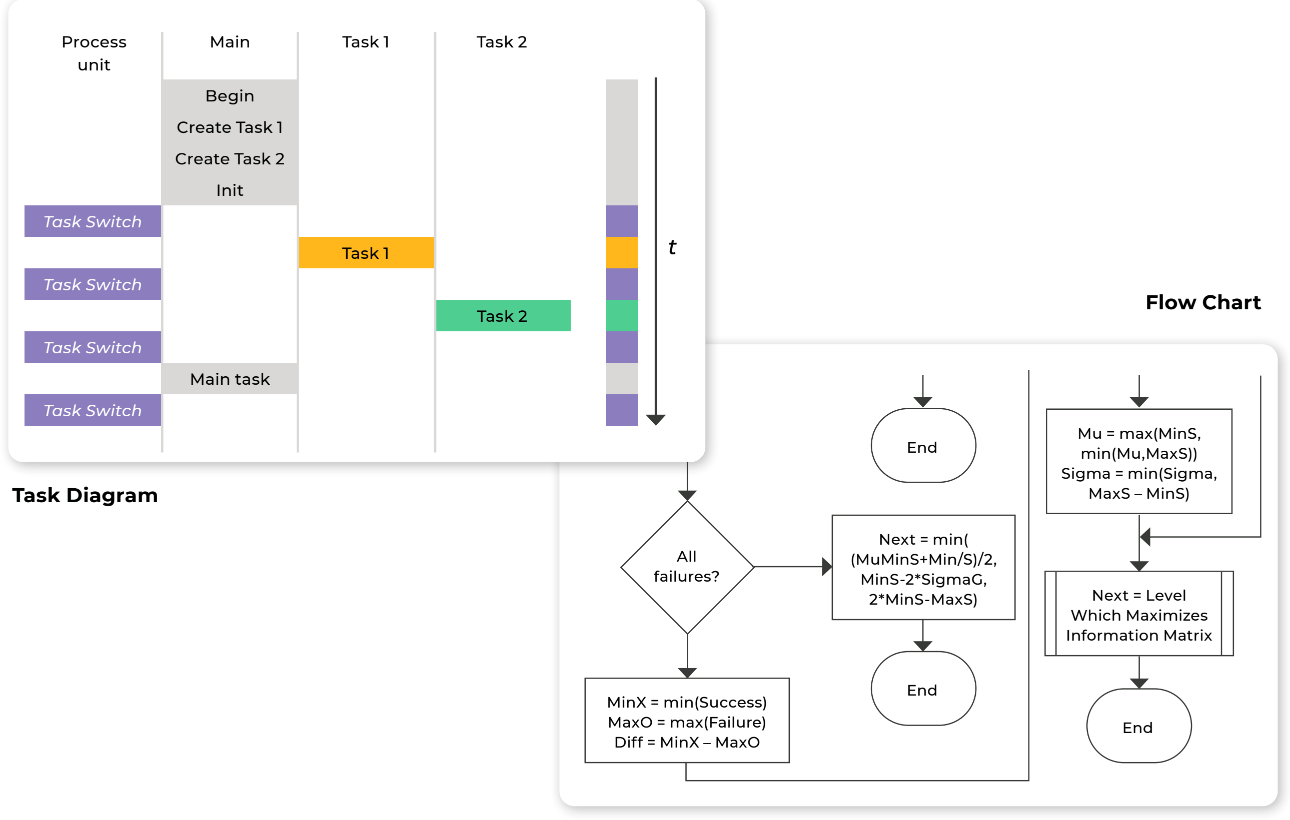


FIGURE 9: A SOFTWARE ENGINEERING VIEW

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

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

- Its data flow structure fits the block diagram approach.
- Its clocks support formal expression of sampling rates.
- Its time operators fit the z operator of control engineering. For instance, z-1, the operator of control engineering (meaning a unit delay), has an equivalent operator called “pre” in Scade.

3.2 Scade Modeling Techniques

3.2.1 Modeling behavior with Scade

FAMILIARITY AND ACCURACY RECONCILED

Scade unifies in a single language two specification formalisms that are familiar to control engineers:

- data flow diagrams to specify control algorithms (*e.g.*, control laws, filters, etc.)
- state machines to specify modes and transitions in an application (*e.g.*, transition from "city driving" to "highway driving", etc.)

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

SCADE OPERATORS

The basic Scade building block is called an operator. It is either a pre-defined operator (*e.g.*, +, pre) or a user-defined operator that decomposes itself using other operators. This allows to build a complex application in a structured way.

An essential concept for operators is that they contain:

1. An interface of strongly typed inputs and outputs
2. A set of equations to compute the outputs from the inputs and any internal context.

An operator can be represented graphically or textually as shown below.

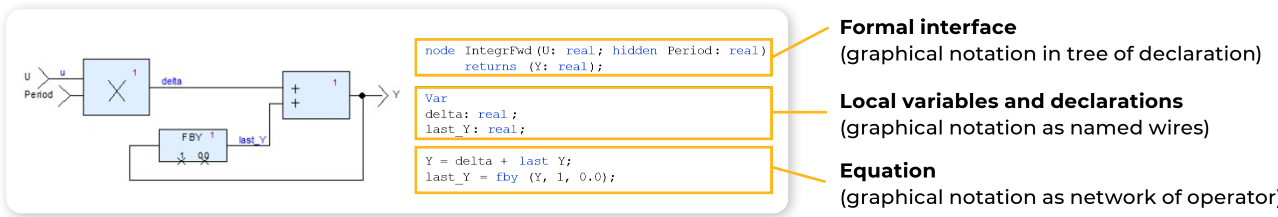


FIGURE 10: GRAPHICAL AND TEXTUAL REPRESENTATION OF OPERATORS

There are two formats for storing Scade models:

- .scade files that use the BNF of the Scade language
- .xscade files that are used for everything created within the SCADE Suite IDE

The textual notation is a projection of the graphical one since it does not contain the graphical layout information. In the SCADE Suite IDE, a user-friendly editing mode supports both graphical and textual operator descriptions.

An operator is fully modular:

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

DATA FLOW DIAGRAMS FOR CONTROL

By “control”, we mean regular periodic computation such as sampling sensors at regular time intervals, performing signal-processing computations on their values, computing control laws and outputting the results. The same sequential function applies to each computation cycle.

In the Scade language, control is graphically specified using data flow diagrams, such as the one illustrated in Figure 11 below.

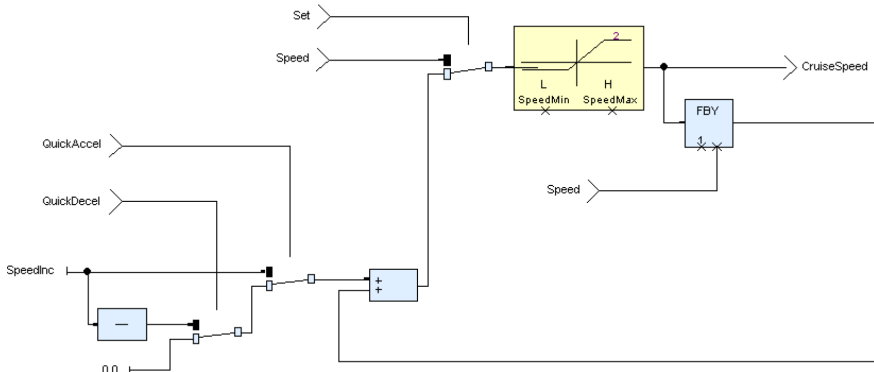


FIGURE 11: SAMPLE OF MODEL DATA FLOWS FROM AN ADAPTIVE CRUISE CONTROL (ACC) SYSTEM

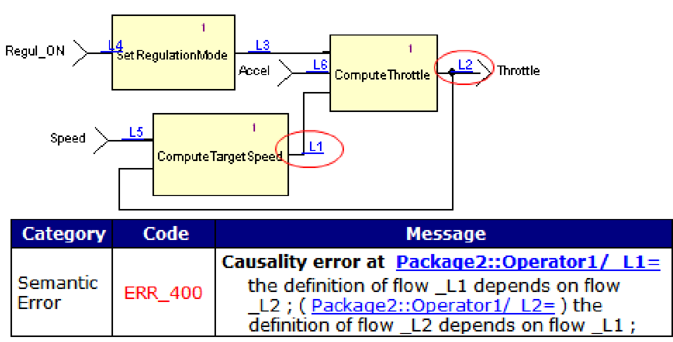
Operators compute mathematical functions, filters, and delays, while wires denote data flowing between operator instances. Operator instances that have no functional dependency are computed concurrently. Flows may carry numeric, Boolean, enumeration, or structured values used or produced by operators.

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

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

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

Scade makes it possible to deal properly with issues of sequence in time and causality. Causality means that if data x depends on data y, then y must be available before the computation of x starts. A recursive data circuit poses a causality problem, as shown in Figure 12 below, where the “Throttle” output depends on itself via the ComputeTargetSpeed and ComputeThrottle operators. With SCADE Suite Semantics Checker, semantic checks[3] detect this error and signal that this output has a recursive definition.



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

FIGURE 12: DETECTION OF A CAUSALITY PROBLEM

[3] *SCADE Suite Semantics Checker is provided with SCADE Suite for running semantic checks during software modeling.*

As shown in Figure 13, inserting an FBY (delay with initial value) operator in the feedback loop solves the causality problem, since the input of the ComputeTargetSpeed operator is now the value of “Throttle” from the previous cycle.

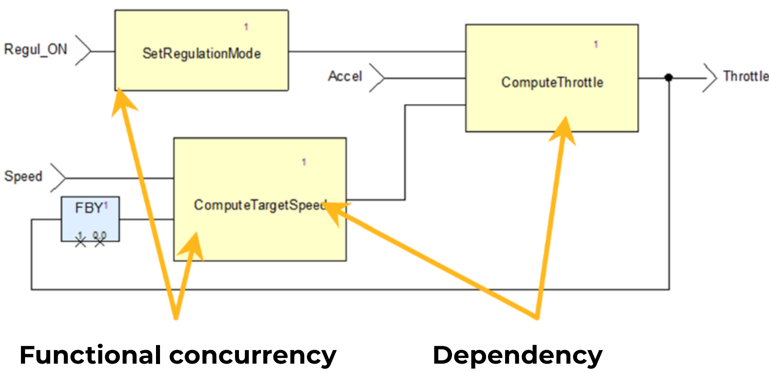
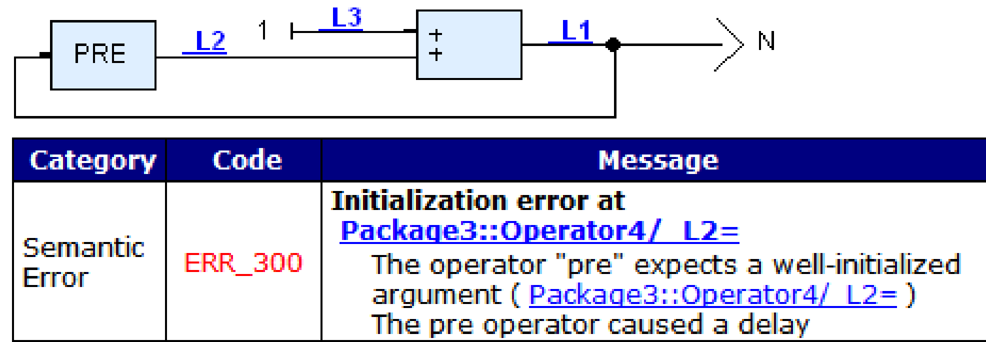


FIGURE 13: FUNCTIONAL EXPRESSION OF CONCURRENCY AND DEPENDENCY

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

- Operators SetRegulationMode and ComputeTargetSpeed are functionally parallel; since they are independent, the relative computation order of these operators does not matter (because, in the Scade language, there are no side effects).
- ComputeThrottle functionally depends on an output of ComputeTargetSpeed.
- Once it has been established that data flow dependencies are correct (*i.e.*, there is no causality cycle), the SCADE Suite KCG code generator[4] takes this into account: it generates code that executes ComputeTargetSpeed before ComputeThrottle. The computation order is always up-to-date and correct, even when dependencies are indirect and when the model is updated. The users do not need to spend time performing tedious and error-prone dependency analyses to determine sequencing manually. They can focus on functions rather than on coding.

Another important feature of the Scade language is related to the initialization of flows. In the absence of explicit initialization, for instance by using the -> (Init) operator, SCADE Suite semantic check emits errors, as illustrated in Figure 14 for a counter model.



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

FIGURE 14: DETECTION OF A FLOW INITIALIZATION PROBLEM

As shown in Figure 15, inserting an Init operator in the feedback loop solves the initialization problem. The second argument of the + operator is 0 in step 1 (initial value), and the previous value of flow N in

[4] *The role of the SCADE Suite KCG code generator is described in detail in Section 7.4.*

steps 2, 3, etc. Mastering initial values is indeed a critical topic for safety-related embedded software.

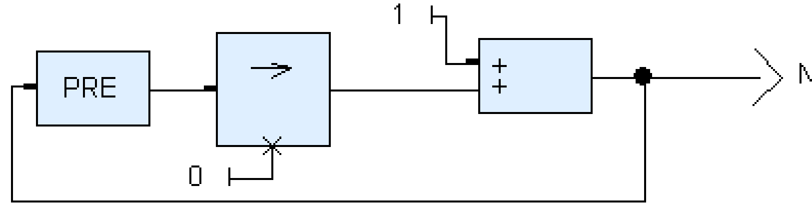


FIGURE 15: INITIALIZATION OF FLOWS

STATE MACHINES FOR DECISION LOGIC

By “decision logic” we mean changing behavior according to external events originating either from sensors and user inputs or from internal program events, for example, value threshold detection. Such decision logic is needed when behavior varies qualitatively as a response to events. This is characteristic of modal human-machine interfaces, alarm handling, complex mode handling, or communication protocols.

As a topic of very extensive studies over the last fifty years, state machines and their theory are well-known and accepted. However, in practice, they have not been adequate even for medium-size applications since their size and complexity tend to explode very rapidly. For this reason, and as shown in Figure 16, a richer concept of hierarchical state machines was introduced in Scade to handle the “decision logic” part of an application.

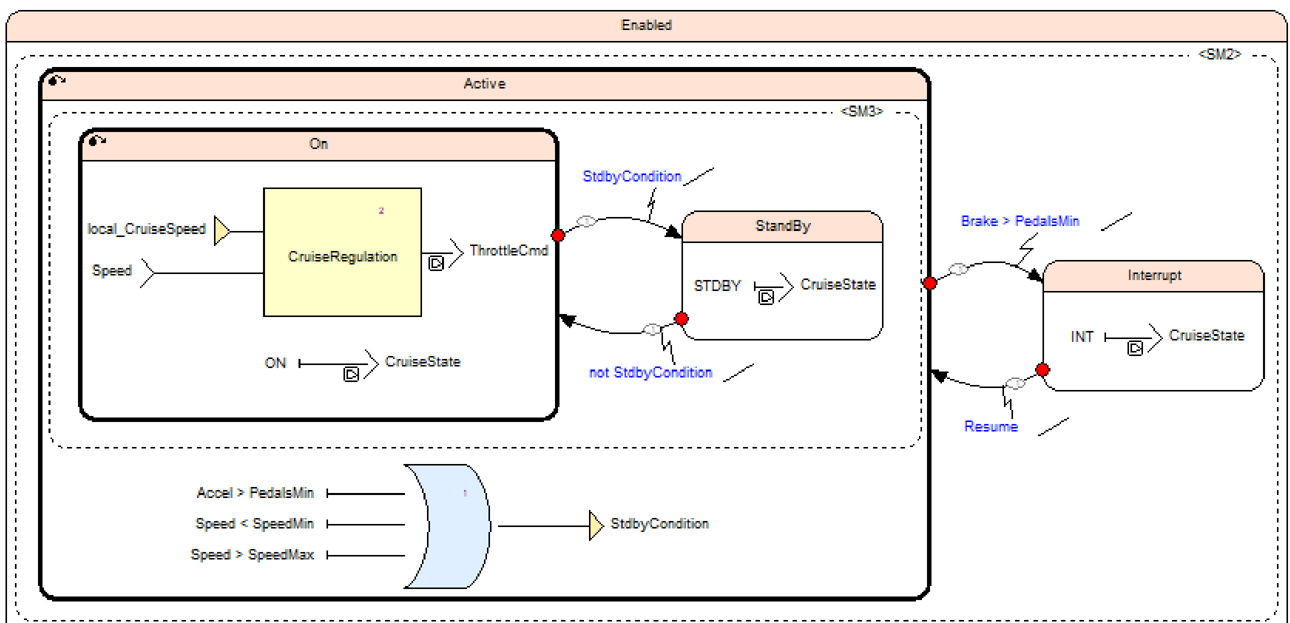


FIGURE 16: A HIERARCHICAL STATE MACHINE

States can be either simple states or macro states, themselves containing a full state machine. When a macro state is active, so is its content that may be composed of other state machines and block diagrams running in parallel. When a macro state is exited by taking a transition out of its boundary, the macro state is exited and all the active state machines it contains are preempted, whichever state they were in. State machines communicate by exchanging flows and signals that may be scoped to the macro state that contains them.

The definition of state machines specifically forbids dubious constructs found in other hierarchical state machine formalisms: transitions crossing macro state boundaries, transitions that can be taken halfway and then backtracked, non-deterministic choice of the transition that can be fired, and so

on. These are non-modular, semantically ill-defined, and very hard to figure out, hence inappropriate for safety-related designs. Most of them are based on a "run to completion semantics" without guarantees that this run terminates. They are usually not recommended by methodology guidelines (see [Statecharts] for a detailed analysis).

COMBINING DATA FLOWS AND STATE MACHINES

Large applications contain cooperating data flows and state machines. SCADE Suite gives developers the ability to freely and rigorously combine and nest these data and control flows, as shown in Figure 17.

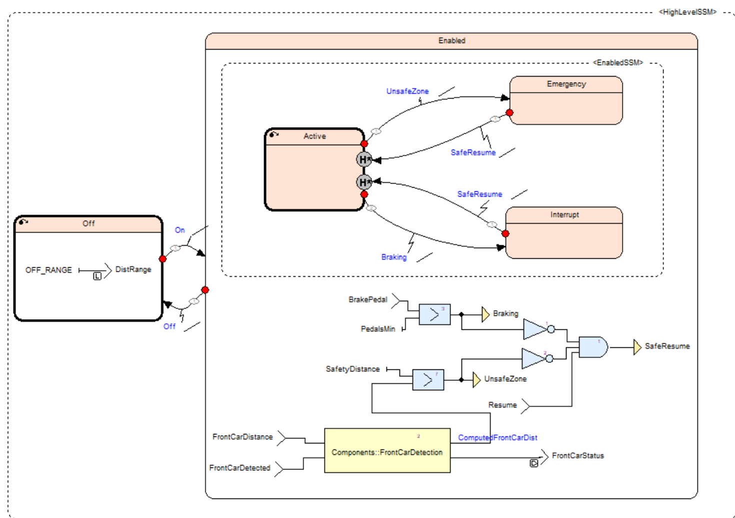


FIGURE 17: MIXED DATA AND CONTROL FLOWS IN AN ADAPTIVE CRUISE CONTROL (ACC)

DATA TYPING

The Scade language is strongly typed, and the following data types are supported:

- Predefined types:
  - Boolean
  - Integer (int8, uint8, int16 uint16, int32, uint32, int64, uint64)
  - Floating point (float32, float64)
  - Enumeration
  - Character
- Structured types:
  - Structures make it possible to group data of different types. For example:

```
Ts = {x. int, y. real};
```

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

```
tab = real^3;
```

- Imported types that are defined in C (to interface with legacy software)

All variables are explicitly typed, and type consistency is verified by SCADE Suite semantic checks.

3.2.2 The SCADE Suite cycle-based intuitive computation model

The cycle-based execution of a SCADE Suite model is a direct computer implementation of the ubiquitous sampling-actuating model of control engineering. It consists in performing a continuous loop of the form illustrated in Figure 18 below.

The body of the loop does in sequence: acquisition of inputs, computation of the reaction, and then emission of the outputs. Once the input sensors are read, the cyclic function starts computing the cycle outputs. During that time, the cyclic functions are unaffected by environment changes[5]. 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.

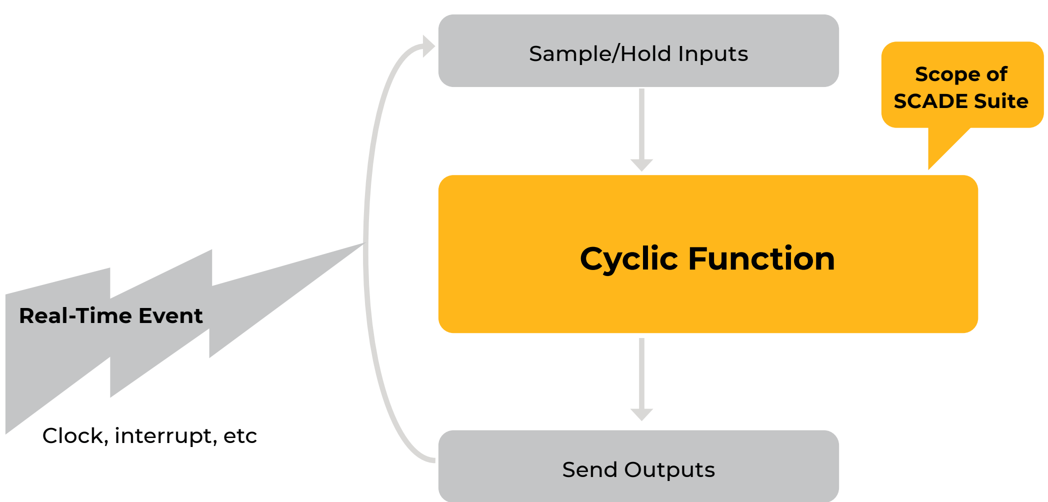


FIGURE 18: THE CYCLE-BASED EXECUTION MODEL OF SCADE

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

THE CONCEPT OF CYCLE IN SCADE SUITE

In a Scade model, each operator and flow have a so-called clock (the event triggering its cycles) and all operators that do not exhibit data flow dependencies act concurrently (see Figure 13). Operators can all have the same clock, or they can have different clocks, which subdivide a master clock. At each of its clock cycle, an operator reads its inputs and generates its outputs. If an output of operator A is connected to an input of operator B, and A and B have the same clock cycle, the outputs of A are used by B in the same cycle, unless an explicit delay is added between A and B. This is the essence of the semantics of the Scade language.

State machines share the same notion of cycle. For a simple state machine, a cycle consists in performing the adequate transition from the current state to this cycle's active state and compute actions in the active state. Concurrent state machines communicate with each other, receiving signals sent by other state machines and possibly sending signals back. Finally, data flow diagrams and state machines in the same design also communicate at each cycle.

BENEFIT OF THE CYCLE-BASED COMPUTATION MODEL

This cycle-based computation model of SCADE carefully distinguishes between logical concurrency and physical concurrency. The application is described in terms of logically concurrent activities, data flow diagrams or state machines. Concurrency is resolved at code generation time, and the

[5] *It is still possible for interrupt service routines or other tasks to run if they do not interfere with the cyclic function.*

generated code remains standard, sequential, and deterministic code, all represented within a simple subset of C. What matters is that the final sequential code behaves exactly as the original concurrent specification, which can be formally guaranteed. There is no overhead for communication, which is internally implemented using well controlled shared variables without any context switching.

3.2.3 SCADE modeling and safety benefits

In conclusion to this Section 3.2, we have shown Scade models formalize a significant part of the software units design. The models are written and maintained once in the project and shared among team members. Expensive and error-prone rewriting is thus avoided; interpretation errors are minimized. All members of the project team, from the specification team to the review and testing teams, can share models as a reference.

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

Finally, we have shown that the Scade language and the SCADE Suite tool strongly supports safety at model level for the following reasons:

- The Scade language has been rigorously defined. Its interpretation does not depend on readers or any tool. It relies on more than 30 years of academic research ([Esterel], [Lustre], [Scade 6]). The semantic kernel of Scade is very stable: it has not changed over all these years.
- The Scade language is simple. It relies on very few basic concepts and simple combination rules of these concepts.
- Control structures remain at a high-level of abstraction. For example, array operations in Scade are expressed as such and do not require low-level loops and indexes. There is no need for goto's, no need for the creation of memory at runtime, no way to incorrectly access memory through pointers or an index out of bounds in an array. Moreover, these principles are reflected in the generated code out of SCADE Suite KCG.
- The Scade language contains specific features oriented towards safety: strong typing, mandatory initialization of flows, etc.
- Scade models are deterministic. A system is deterministic if it always reacts in the same way to the same inputs occurring with the same timing. In contrast, a non-deterministic system can react in different ways to the same inputs, the actual reaction depending on internal choices or computation timings.
- The Scade language provides a simple and clean expression of concurrency at functional level (data flows express dependencies between operators). Within a model, this avoids the traditional problems of deadlocks and race conditions.
- SCADE Suite performs the complete verification of language syntactic and semantic rules, such as type and clock consistency, initialization of data flows, or causality in models.

Note 1: To assess determinism of an application developed in Scade, it is necessary to consider the Scade model and the boundaries of the Scade model with its environment. The inputs and outputs of the Scade model are at the boundary of the model. The imported operators (i.e., the operators that are called by the Scade model, but are developed in another language, such as C) have their inputs and outputs that are also at the boundary of the Scade model. Imported code inputs are model output and imported code outputs are model inputs. As mentioned above, the Scade model itself is deterministic and thus does not introduce any source of non-determinism; only imported operators may do so by doing side effects.

Note 2: An in-depth analysis regarding determinism is proposed in this handbook for the specific case of integration of the application in an AUTOSAR platform (see Section 9.3.6 and [SCS-ACG-Safety Analysis] for further details). However, this is a generic topic that must be handled, whatever is the host platform for the embedded application software.

The remainder of this handbook presents the full SCADE Suite toolchain and explains how full benefits can be obtained using SCADE Suite and its companion verification tools in an ISO 26262:2018 project.

3.3 The SCADE Toolchain

3.3.1 SCADE toolchain overview

SCADE is a product family for embedded systems and software development that comprises several products that can be used together or independently (SCADE Architect, SCADE Suite, SCADE Display, SCADE Test, and SCADE LifeCycle), as well as dedicated solutions for Automotive (SCADE Automotive Package).

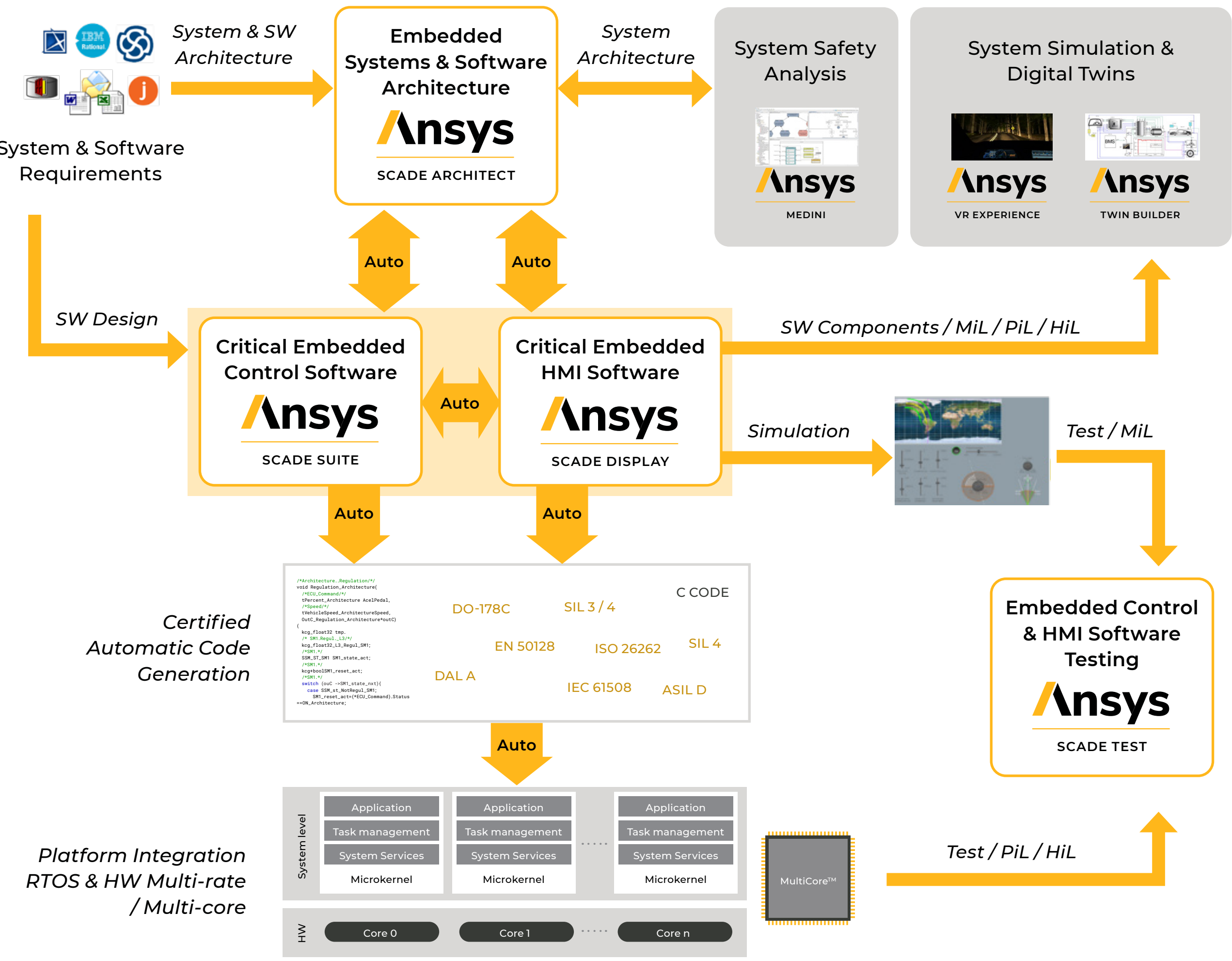


FIGURE 19: THE SCADE PRODUCT FAMILY

3.3.2 SCADE Architect

SCADE ARCHITECT OVERVIEW

SCADE Architect is a system and software architecture design product line for complex embedded systems modeling, based on the SysML standard notation. SCADE Architect augments this underlying capability by providing a user-friendly and intuitive model-based environment for system and software architects.

SCADE Architect product capabilities are shown in Figure 20. SCADE Architect allows to refine the system and software requirements, design the architecture of the application, verify design rules, and automatically produce Interface Control Documents (ICDs). Moreover, blocks that are

implemented in software can be automatically synchronized with the corresponding SCADE Suite blocks in the software design model (see next Section), thus supporting collaborative work between system, safety, and software engineers.

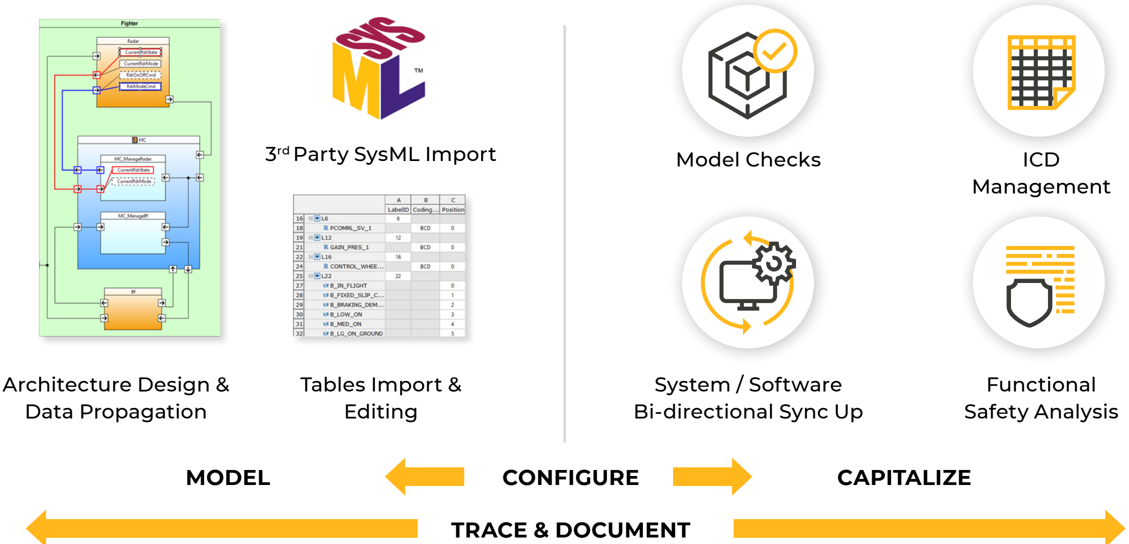


FIGURE 20: SCADE ARCHITECT PRODUCT CAPABILITIES

INTEGRATION OF SCADE ARCHITECT AND ANSYS MEDINI

SCADE Architect can be integrated with Ansys medini Analyze for architecture-driven safety and cybersecurity analyses.

Ansys medini (see Figure 21) is a model-based solution that supports standard safety analysis methods such as FHA, FMEA, FTA at system level in a consistent and efficient way, as well as the creation of a Functional and Technical Safety Concept, ending up in new safety requirements. This solution also supports cybersecurity analysis methods such as threat analysis, and safety management with the Digital Safety Manager (DSM) new module which supports the creation and management of safety plans and safety cases.

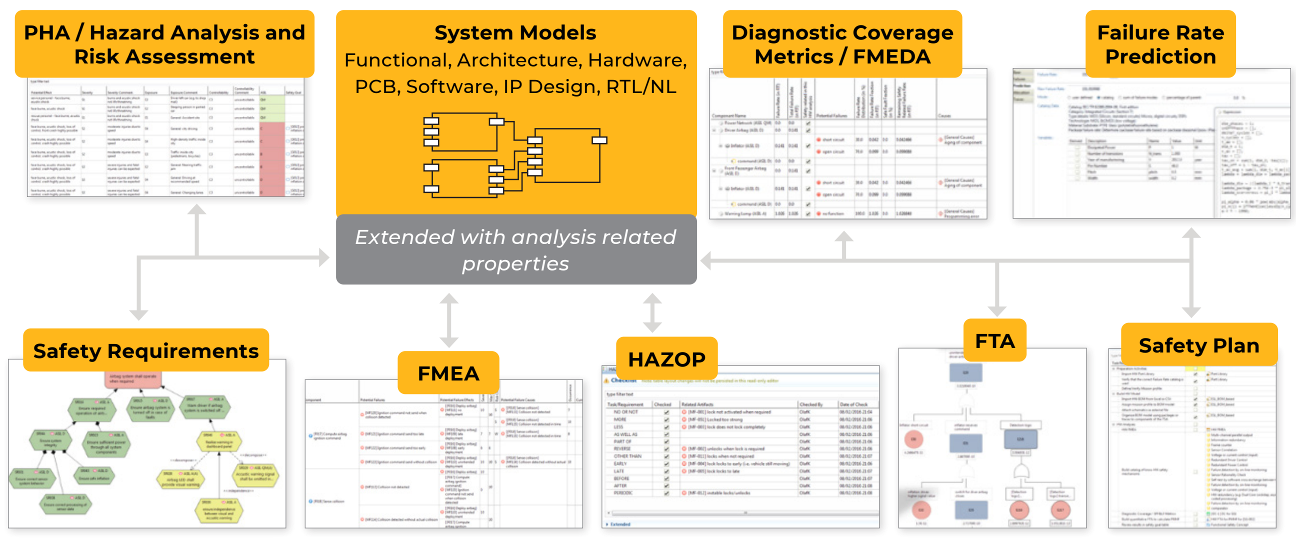


FIGURE 21: MEDINI ANALYZE PRODUCT CAPABILITIES

SCADE ARCHITECT AUTOSAR CONFIGURATION AND SUPPORT

The Ansys SCADE Automotive Package extends SCADE Architect and SCADE Suite design capabilities for the automotive industry with an **AUTOSAR configuration of SCADE Architect**, compliant with the AUTOSAR standard [AUTOSAR].

The SCADE Architect AUTOSAR configuration:

- provides AUTOSAR XML (ARXML) import/export capabilities and synchronization with SCADE Suite for Software Components (SWC) design
- comes with dedicated SCADE Automotive Code Generator for AUTOSAR (ACG) which relies on SCADE Suite KCG and generates the integration code between SCADE Suite KCG generated code and AUTOSAR RTE functions

Figure 22 provides an overview of the SCADE positioning in an AUTOSAR workflow. This is further detailed in Section 3.3.6.

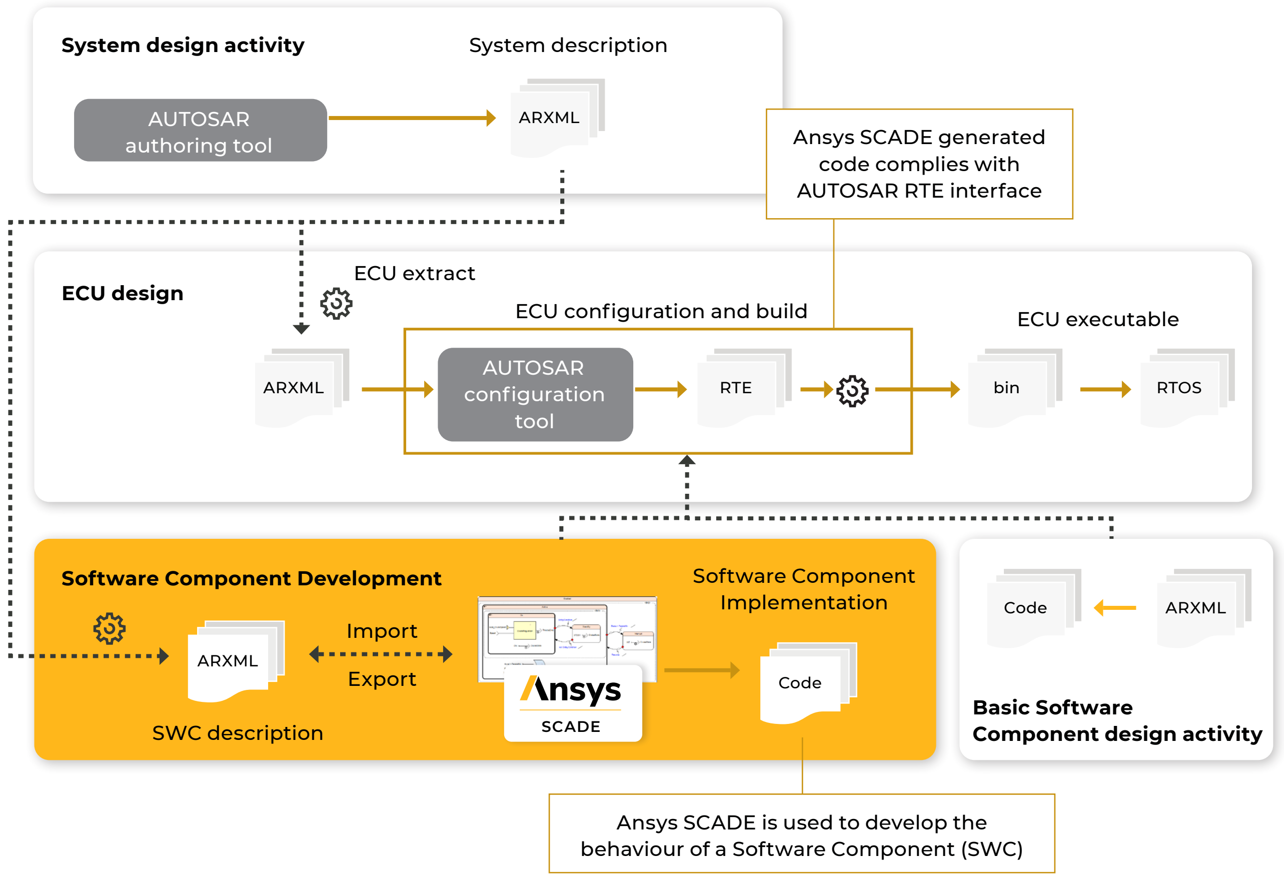


FIGURE 22: SCADE IN THE AUTOSAR FLOW

3.3.3 SCADE Suite

SCADE Suite is a software design product line for embedded control software modeling, verification, and code generation. SCADE Suite provides a user-friendly and intuitive model-based environment for software engineers.

SCADE Suite product capabilities are depicted in Figure 23. SCADE Suite allows to create software design models, to check consistency of the designs, to perform model simulation, and to automatically generate source code from the models through a qualified code generator, SCADE Suite KCG that produces MISRA C:2012 compliant C code (see [MISRA C:2012] and [MISRA C:2012/AMD1]).

As sketched above, the Scade modeling language and the SCADE Suite tool provide many ways to ensure robustness and safety. For example:

- The Scade modeling language is fully deterministic.
- There is no way to express an array access that could be out of bounds.
- All possible cases are addressed in selection constructs.
- All possible variables values fire a deterministic transition in a state machine.
- Concurrency is expressed at logical level, with no risk of deadlock or race condition at model level (and thus, at the level of the generated code).
- etc.

The SCADE Suite Semantics Checker performs a wide number of safety checks on Scade models:

- All variables have been correctly initialized.
- All variables are assigned a value no more than once in a given execution cycle.
- The Scade model does not have instantaneous loops within an execution cycle (a value is needed before it has been computed).
- etc.

In addition, SCADE Suite comes with several verification tools to accomplish all other needed verification activities:

- SCADE Suite Simulator for model debugging
- SCADE Suite Design Verifier (DV) for formal verification of functional model properties
- SCADE Suite Timing & Stack Optimizer (TSO) and Timing & Stack Verifier (TSV) for estimating the relative Worst-Case Execution Time (WCET) or stack usage of tasks of a SCADE application
- SCADE Suite Rule Checker for verification of user specific design rules
- etc.

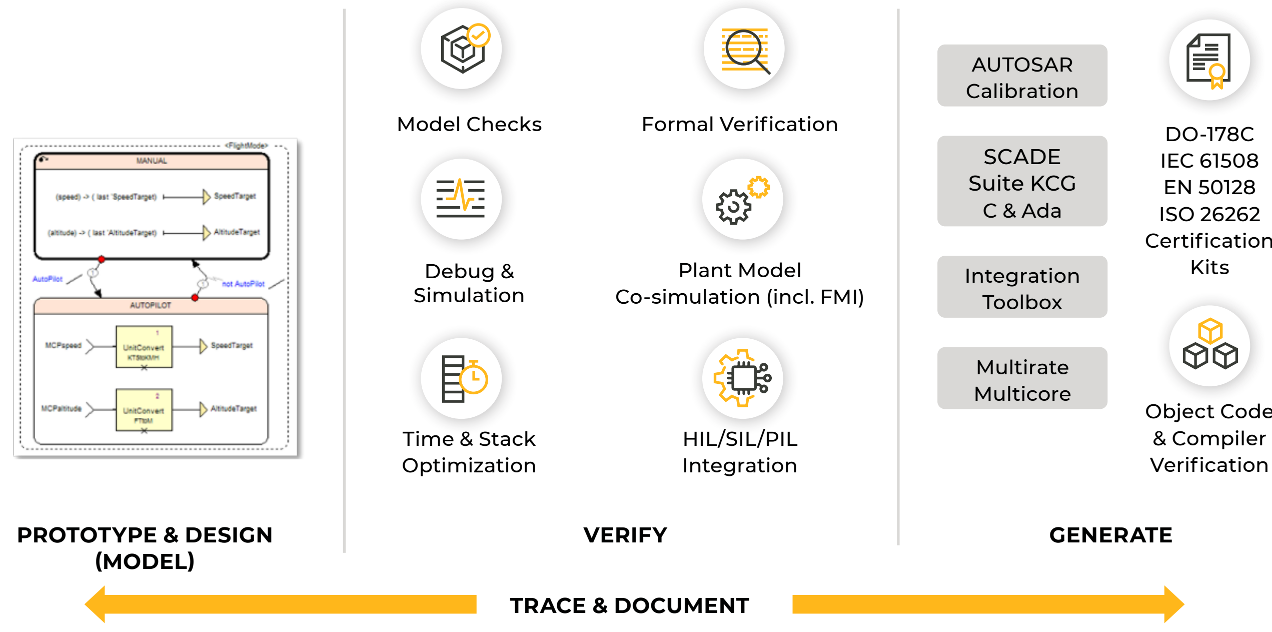


FIGURE 23: SCADE SUITE PRODUCT CAPABILITIES

SCADE Suite comes with the following additional capabilities:

- export to FMI (Functional Mock-up Interface) compliant simulation tools. SCADE Suite models can be exported as FMU (Functional Mock-up Unit) to support efficient model exchange and Co-simulation in system level simulators
- SCADE Integration Toolbox, a Python framework providing access to all project data: model, test cases, project, generated code, generated reports, etc. These APIs facilitate SCADE toolchain and generated code integrations. Thanks to this Toolbox the integration process can be automated.

3.3.4 SCADE Test

SCADE Test (see Figure 24) provides engineers with a complete testing environment for Scade models, enabling prototyping and validation using graphical widgets to build simulation control cockpits, test case authoring and management, test case execution on host, and automatic translation of host test cases to target test cases, as well as model coverage assessment. SCADE Test provides a user-friendly and intuitive environment for verification engineers.

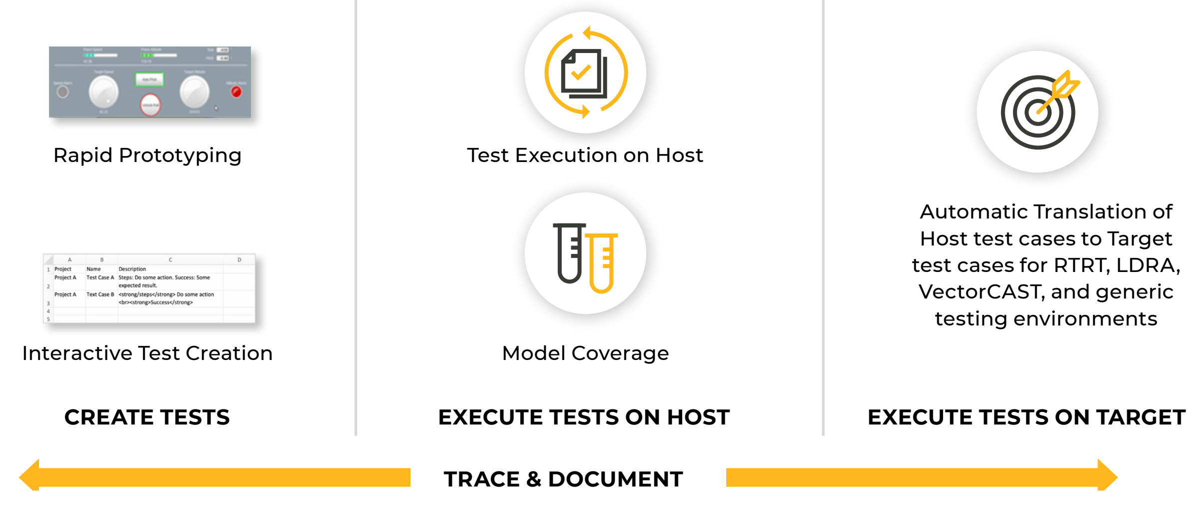


FIGURE 24: SCADE TEST PRODUCT CAPABILITIES

3.3.5 SCADE LifeCycle

SCADE LifeCycle is an Application LifeCycle Management (ALM) product line that provides software engineers using the SCADE product family necessary tools to manage their projects efficiently.

SCADE LifeCycle is composed of the following modules:

- SCADE LifeCycle Reporter to automate the time-consuming task of creating detailed and complete reports from SCADE Architect, SCADE Suite, and SCADE Test models
- SCADE LifeCycle Model Change to enable incremental reviews of SCADE Suite models
- SCADE LifeCycle ALM Gateway to establish direct traceability between SCADE Architect, SCADE Suite and SCADE Test models and test suites, and requirements managed in various third-party tools (*e.g*, IBM DOORS, IBM DOORS NG, Siemens Polarion, Jama Connect, Microsoft Word, Excel, etc.)

3.3.6 SCADE for AUTOSAR

In this Section, we briefly introduce AUTOSAR, and we then present how SCADE supports the development of AUTOSAR Runnables.

INTRODUCTION TO AUTOSAR

AUTomotive Open System ARchitecture (AUTOSAR) [AUTOSAR] is a global development partnership of automotive interested parties founded in 2003. It pursues the objective to create and establish an open and standardized software architecture for automotive electronic control units (ECUs).

As shown in Figure 25, AUTOSAR describes standard interfaces for a three-layer architecture of application software components communicating using a Virtual Functional Bus (VFB) and accessing basic services provided by the platform.

It is structured as follows:

- **Basic Software (BSW) components:** standardized software modules that offers services needed to run the functional part of the upper software layers
- **Runtime Environment (RTE):** middleware which abstracts from the network topology for the inter- and intra-ECU information exchange between the application software components and between the Basic Software (BSW) and the applications
- **Software Components (SWC):** application software that interact with the Runtime Environment. Each software component is composed of one or more Runnables (or tasks) that access to the SWC's ports. Additionally, the SWC contains the description of local memories and all the activation conditions of the Runnables (event-based activation, schedule based on time, etc.)

Source: [AUTOSAR]

Application Layer
Runtime Environment
System Services
Memory Services
Crypto Services
Off Board Communcation Services
Communication Services
I/O Hardware Abstraction
Complex Drivers
Onboard Device Abstraction
Memory Hardware Abstraction
Crypto Hardware Abstraction
Wireless Communication HW Abstraction
Communication Hardware Abstraction
Microcontroller Drivers
Memory Drivers
Crypto Drivers
Wireless Communication Drivers
Communication Drivers
I/O Drivers
Microcontroller

FIGURE 25: THE AUTOSAR THREE-LAYER ARCHITECTURE

During the RTE **Contract Phase**, the interfaces of the Runnables are generated as C header files. The implementation of the C function (done manually or using a model-based notation, such as Scade) corresponding to each Runnable is based on the defined interfaces and a software requirements specification.

As all the interfaces, internal variables, activation conditions and services are standardized, the Runtime Execution Environment (RTE) can be automatically generated. The RTE provides the implementation of the various service calls (communications or low-level services) that will be exposed to the application, as well as the implementation of SWC internal variables and Runnables scheduling. The RTE exposes its API as C functions which prototypes are fully determined by the AUTOSAR standard.

Note: These functions usually return an error code, and not directly a value. The value is given as a parameter (in or out, depending on the service).

An example of an architecture in AUTOSAR with the Virtual Functional Bus (VFB) concept is provided in Figure 26 . All the information is stored in standard ARXML file(s). The software architecture, depicted as a SCADE Architect model, is made of three software components: SWC1, SWC2, SWC3 with ports for communication. Inside a SWC the behavior is given by the interface of the Runnables (R<i>) that will be implemented in C as tasks in the operating system.

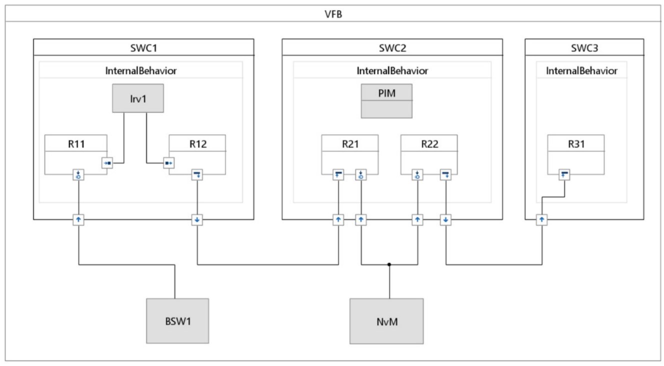


FIGURE 26: AUTOSAR ARCHITECTURE EXAMPLE – VFB

Let us have a deeper look at the architecture:

- SWC1 contains two Runnables, R11 and R12 and an internal runnable variable Irv1:
  - R11 and R12 can communicate with each other using Irv1 with a dedicated **VariableAccess.**
  - R11 performs a **Server call** to the basic software component BSW1.
  - R12 has a dedicated **VariableAccess** to a port of SWC1.
- SWC2 contains two Runnables R21 and R22 and a Per-Instance Memory (PIM):
  - R21 and R22 perform **Server calls** to the Non-Volatile Memory NvM BSW component. At that level, there is no information that the NvM uses the PIM. This is described in a dedicated part of the ARXML file. There is also no information that R21 or R22 use the PIM. This is described in the Runnable requirements.
  - R21 and R22 each have a dedicated **VariableAccess** to SWC2 ports.
- Similarly, SWC3 has a Runnable R31, which has a dedicated **VariableAccess** to its port.
- There are also communications through their ports between SWC1 and SWC2, SCW2 and SWC3.

The VFB view describes the software components, the communications, and the data access without detailing the mapping on ECUs, and it does not consider how the communications are performed (shared memory, CAN, Flexray, etc.).

At that level, we can also specify:

- If a communication is implicit or explicit:
  - An **explicit** communication if fired as soon as requested.
  - An **implicit** write is done once the Runnable ends its execution. The readers are guaranteed to read a stable value.
- What conditions activate a Runnable.

From this description in ARXML, it is possible to generate the C code headers with the declarations for the RTE API functions to read or write a **VariableAccess**, the Runnable functions, the memory information and the functions associated to **Server calls**.

Once the software architecture is available, implementation of the Runnable can start. The configuration of the ECUs can also start with additional information, such as the mapping of the SWCs on the ECUs and the communication means.

The system architecture, depicted as a SCADE Architect model, is provided in Figure 27:

- SWC 1 and SWC2 are mapped onto ECU1.
- SWC2 is mapped onto ECU2.
- Communication between ECU1 and ECU2 is done using the CAN protocol.

The role of the integration engineer is to configure the ECU with only the services requested by the application (SWC1 and SWC2 or SWC3, depending on the considered ECU), to prepare the RTOS, to allocate memories and to combine with the results of the Runnable development (C source code or object code). The RTE is automatically generated from the ARXML Architecture description (implementation of the API code and scheduling of the Runnables).

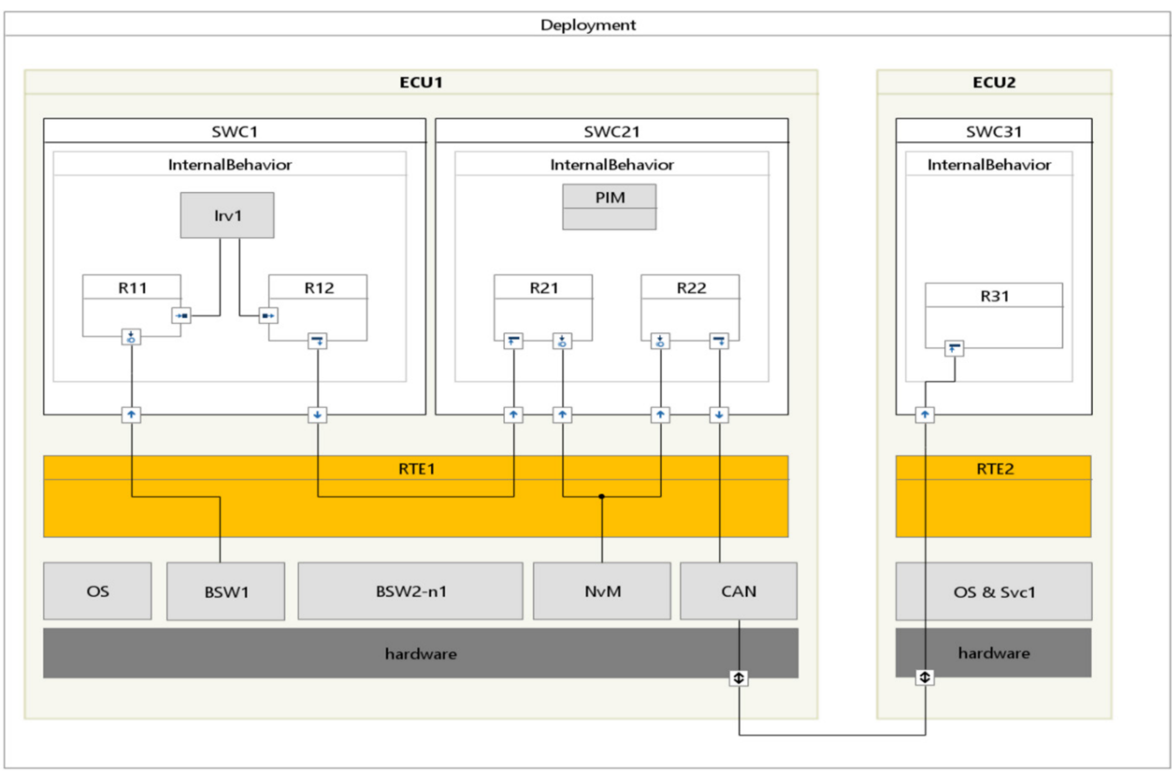


FIGURE 27: AUTOSAR ARCHITECTURE EXAMPLE – ECU MAPPING AND RTE

SCADE AUTOSAR SUPPORT FOR THE DEVELOPMENT OF RUNNABLES

The SCADE AUTOSAR workflow that is used to develop and verify the software components (SWC) Runnables is shown in Figure 28 below.

It is made of a combination of the following SCADE tools:

- SCADE Architect, configured for AUTOSAR Runnables and SWC architectural design
- SCADE Suite for modeling the SWC Runnables
- SCADE LifeCycle Reporter for documenting the Runnables
- SCADE Test for verifying the Runnables
- SCADE Automotive Code Generator for AUTOSAR (ACG) for generating the AUTOSAR-compliant C source code for each Runnable

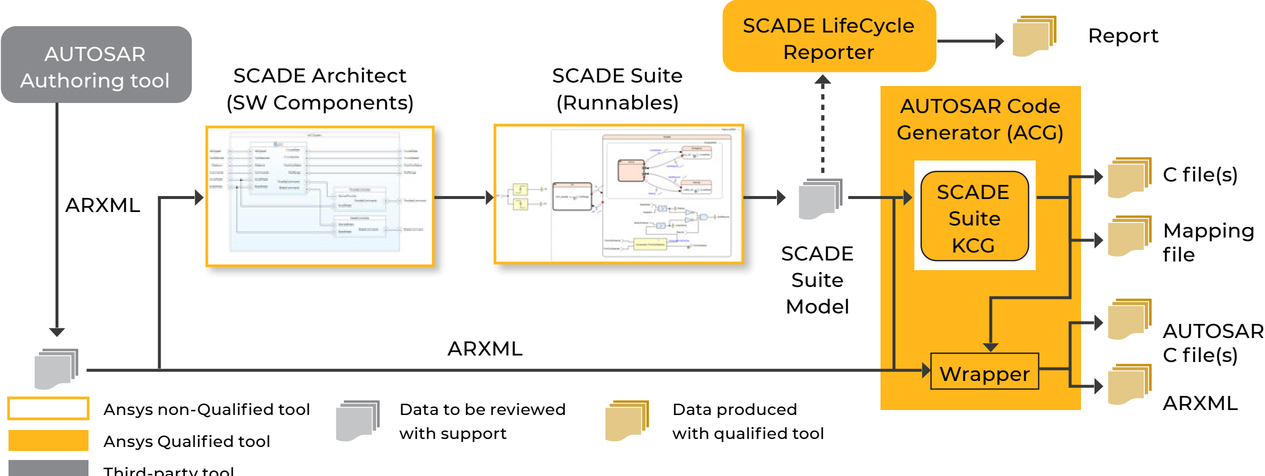


FIGURE 28: THE SCADE AUTOSAR WORKFLOW

3.3.7 SCADE-based workflow summary

As a summary, we have enabled a SCADE-based workflow that is illustrated by Figure 29 below and offers the following benefits:

- Passing information from system to software is based on models that can be understood by both parties and shared by the tools used in the lifecycle. SCADE Architect and SCADE Suite can synchronize the software architecture description.
- SCADE Suite semantic checks detect many modeling inconsistencies such as inconsistent data typing, data that are not computed before use, or that are computed in several places.
- SCADE Test enables validation of Scade models *wrt.* software requirements, in close cooperation between system architects and software engineers.
- SCADE Suite KCG/SCADE ACG automatically generate MISRA C:2012 and AUTOSAR compliant code from the design models.
- Since SCADE Suite provides qualified code generation and SCADE Test qualified automated translation of host test cases to target test cases, experience has shown that re-running these test cases on target is much less expensive than in a traditional process, since almost all errors are detected before going to target and there is no need to re-write or review the test cases of the SCADE part of the application software.

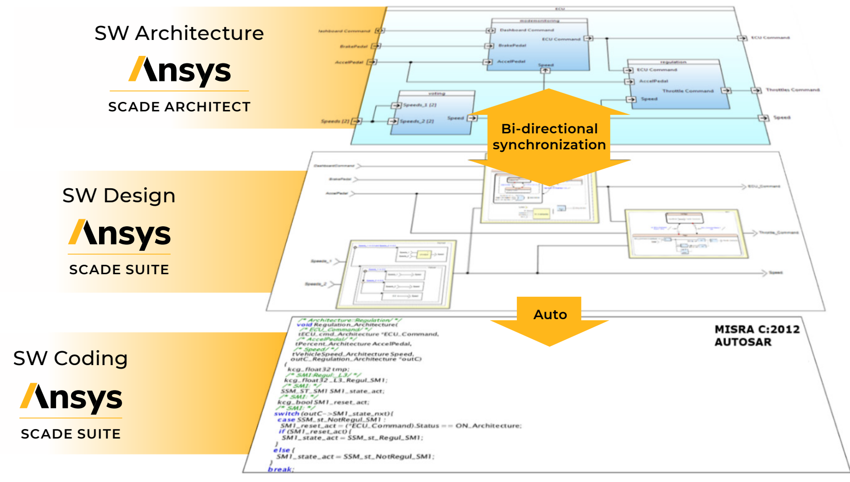


FIGURE 29: SCADE-BASED INTEGRATED SOFTWARE WORKFLOW

3.4 Takeaway from Using SCADE as a Model-Based Development Environment

In this Chapter, we have presented the SCADE model-based approach, which relies on Scade, a domain specific language for the development of safety-related embedded software applications.

The Scade language

- is based on well-trusted principles such as modularity, hierarchy and concurrency
- is a strongly typed language
- specifies behaviors unambiguously and is deterministic
- comes with both a graphical and a textual notation

The SCADE toolchain provides

- syntactic and semantic checks to verify that Scade models are correct
- automated production of design documentation, ensuring that it is correct and up to date by construction
- simulation of Scade models to verify their dynamic behavior according to the software requirements
- formal verification techniques that can be directly applied to prove functional properties of models and detect corner cases defects
- model coverage analysis to assess how thoroughly a Scade model was tested and to detect unintended functions in the model
- time and stack analysis for early verification of compatibility in terms of execution time and memory size between the Scade model and the target platform.
- qualified code generation that automatically produces MISRA C:2012 and AUTOSAR compliant source code and guarantees that the source code complies with the semantics of the input Scade model
- connection to requirements and configuration management tools

4

GENERAL TOPICS FOR THE PRODUCT DEVELOPMENT AT THE SOFTWARE LEVEL

4.1 Objectives and Work Products

The objective of this initial sub-phase (Clause 5 of [ISO 26262-6:2018]) are:

- to ensure a suitable and consistent software development process (see Figure 3)
- to ensure a suitable software development environment

Supporting information for this sub-phase includes:

- available qualified software tools
- modeling and coding guidelines
- methodological guidelines

4.2 Requirements and Recommendations

Section 5.2 of [ISO 26262-6:2018] defines the reference model for the development of software that is reproduced in Figure 3 of this handbook. NOTE 1 of this Figure is adding: “Development approaches or methods from agile software development can also be suitable for the development of safety-related software, …. However, agile approaches and methods cannot be used to omit safety measures or ignore the fundamental documentation, process or safety integrity of product rigour required for the achievement of functional safety”.

Section 5.4 of [ISO 26262-6:2018] provides the following requirements and recommendations for setting up the development environment:

- ensure suitability of methods, guidelines and tools to develop safety-related embedded software
- ensure consistent support of the development sub-phases and their work products throughout the development lifecycle

Criteria for selecting a design, modeling or programming language include:

- unambiguous definition
- suitability for specifying safety requirements
- modularity, abstraction and encapsulation
- support of structured constructs

If the chosen languages are not sufficiently addressing these criteria, they must be covered by additional guidelines, as listed in Table 9 below.

TABLE 8: TOPICS TO BE COVERED BY MODELING AND CODING GUIDELINES

Source: Table 1 in ISO 26262-6:2018

Topics		ASIL			
		A	B	C	D
1a	Enforcement of low complexity[a]	++	++	++	++
1b	Use of language subsets[b]	++	++	++	++
1c	Enforcement of strong typing[c]	++	++	++	++
1d	Use of defensive implementation techniques[d]	+	+	++	++
1e	Use of well-trusted design principles[e]	+	+	++	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++
1i	Concurrency aspects[f]	+	+	+	+

[a] *An appropriate compromise of this topic with other requirements of this document may be required.*

[b] *The objectives of topic 1b include:*
- *Exclusion of ambiguously-defined language constructs which may be interpreted differently by different modellers, programmers, code generators or compilers.*
- *Exclusion of language constructs which from experience easily lead to mistakes, for example assignments in conditions or identical naming of local and global variables.*
- *Exclusion of language constructs which could result in unhandled run-time errors.*

[c] *The objective of topic 1c is to impose principles of strong typing where these are not inherent in the language.*

[d] *Examples of defensive implementation techniques:*
- *Verify the divisor before a division operation (different from zero or in a specific range).*
- *Check an identifier passed by parameter to verify that the calling function is the intended caller.*
- *Use the "default" in switch cases to detect an error.*

[e] *Verification of the validity of the underlying assumptions, boundaries and conditions of application may be required.*

[f] *Concurrency of processes or tasks is not limited to executing software in a multi-core or multi-processor runtime environment.*

4.3 Using SCADE for the Product Development at the Software Level

Our proposal is to use the SCADE toolchain as the basis for the Software Development Environment, for the parts that relate to the application software.

The overall toolchain, the tool features, their benefits, and their applicability in the context of safety-related embedded software development are described in Chapter 3 and Table 29 of Appendix C.1 that clearly states how SCADE meets the modeling and coding guidelines of Table 1 of ISO 26262-6:2018.

Let us now address three additional topics:

- Traceability throughout the development process
- Collaborative software development with SCADE
- Agile software development with SCADE

4.3.1 Traceability throughout the development process

As pictured in Figure 3, the software development process is composed of:

- the software requirements specification process, including specifications of functional and operational requirements, timing and memory constraints, hardware and software interfaces,

failure detection and safety monitoring requirements, as well as requirements related to freedom from interference between software elements

- the software architectural design process, which is based on the software requirements and includes description of components and their interfaces, resource limitations, scheduling, and communication mechanisms
- the software unit design and implementation process which produces the software units detailed design, and the source code and object code
- the software unit verification process which verifies the above
- the software integration and verification process which produces executable object code
- the testing process of the fully integrated embedded software on the target platform

At all stages of the development process, traceability is required: between software requirements and architectural design, between software requirements, architectural design and software units detailed design, between software units detailed design and source code; and also, between software requirements and tests.

Traceability between software requirements, software architectural design and software units detailed design, as well as traceability between software requirements and test cases and procedures, are supported by the SCADE LifeCycle ALM Gateway. Software requirements are created in an external Requirements Management (RM) tool; they are imported into SCADE; links between requirements, test cases and procedure, and SCADE design elements can be established; models and links can be re-exported to the RM tool.

Figure 30 below illustrates the links between requirements and design elements.

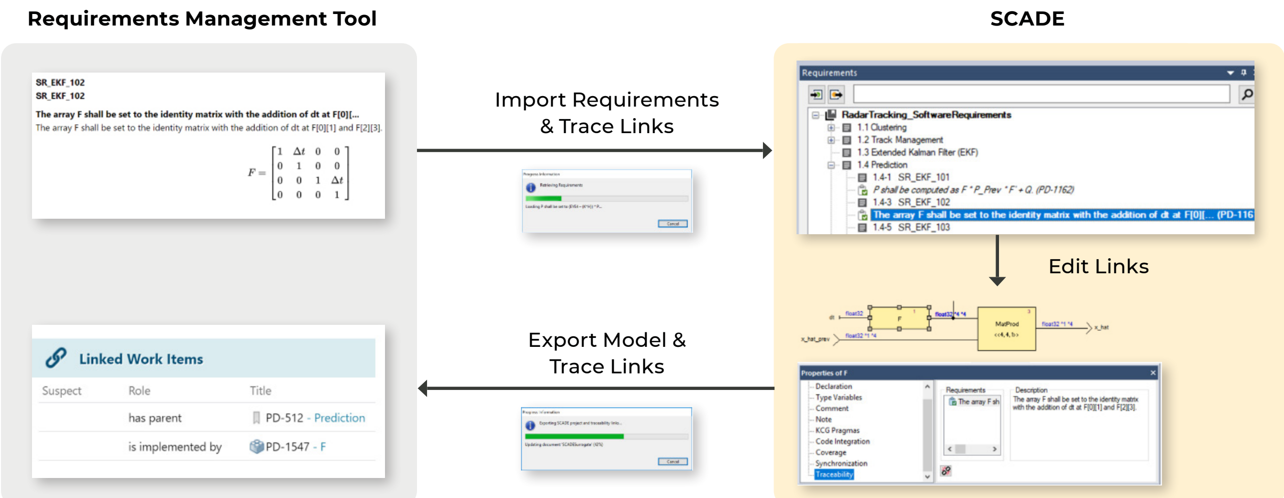


FIGURE 30: TRACEABILITY BETWEEN SOFTWARE REQUIREMENTS AND SCADE DESIGNS

Traceability between software unit designs in SCADE Suite and C source code generated by SCADE Suite KCG is handled by the production of specific traceability information (see Section 7.4 for further details).

4.3.2 Collaborative software development with SCADE

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

SCADE projects (.etp files) organize the design into modular containers. Projects are independent of the underlying model.

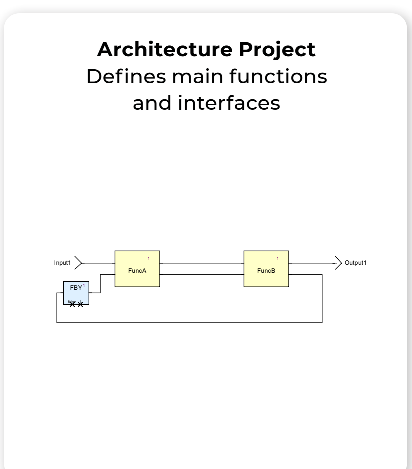
They support an efficient organization and are usually made of:

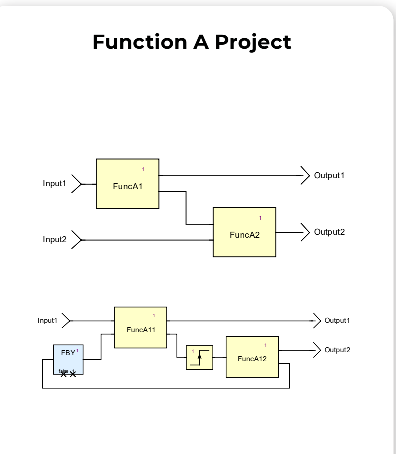
- a component project that provides a complete functional view of a given SCADE component
- a set of library projects that contains shared objects such as types, constants, and functions intentionally located in a dedicated project for re-usability purposes or due to Intellectual Properties (IP) constraints.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

In a typical project organization:

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

A typical teamwork organization is described in Figure 31.





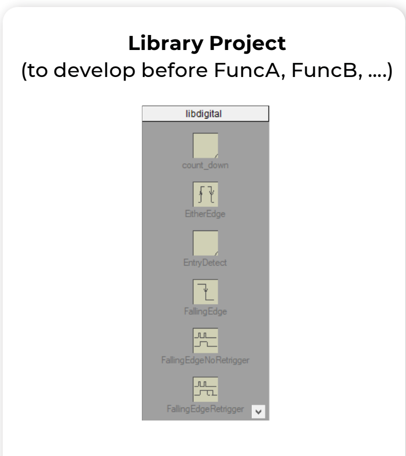


FIGURE 31: TYPICAL TEAMWORK ORGANIZATION

The best organization is to consider one single engineer working on one separate etp file. This etp file groups XSCADE files (*.xscade) or SCADE files (*.scade) corresponding to the definition of a component (see “Function A project” in Figure 31) or a library (see “Library project” in Figure 31).

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

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

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

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

4.3.3 Agile software development with SCADE

Agility is focused on enabling project stakeholders, such as customers, system engineers, safety engineers, software developers, to collaborate more closely on accelerating delivery.

Continuous Integration/Continuous Delivery (CI/CD) is a software engineering practice where team members integrate their work with increasing frequency (*e.g.,* nightly builds/nightly tests) and deploy what CI has built in a way that it can be released at any time.

SCADE tools can be run in batch mode. The generated artifacts include a qualified model design report, source code, test reports, and coverage reports. The independence of these qualified tools from the editor enables a Continuous Integration/Continuous Deployment (CI/CD) pipeline. Thus, the software code and all supporting artifacts are consistently generated.

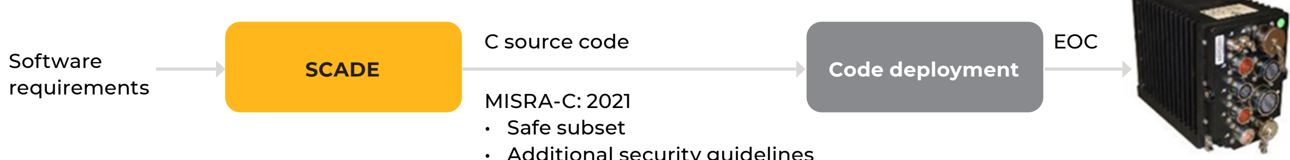


FIGURE 32: FROM REQUIREMENTS TO DEPLOYMENT WITH SCADE

Once the embedded code is generated, it is easily deployed to an embedded target. All generated code is non-proprietary, with no library dependencies, ensuring maximum flexibility to embed the code into any target environment.

Portability of the generated code supports simple integration with any embedded run-time executive or RTOS. This completes the containerization of the embedded software, ensuring a safe pipeline from requirements to design, code, and integration.

4.4 Takeaway from Using SCADE for the Product Development at the Software Level

Based on the discussion of Chapter 3, the Scade modeling notation and the SCADE toolchain provide an efficient framework to meet the requirements and recommendations of Clause 5 of [ISO 26262-6:2018] in the following ways:

- Scade is a domain-specific language for modeling safety-related embedded software.
- Scade is a modular language to specify behaviors unambiguously.
- The SCADE toolchain consistently supports a development lifecycle such as the one of Figure 3.
- The combination of SCADE Architect and SCADE Suite enables an efficient link between architectural design activities and the initial steps of software detailed design.
- The SCADE Suite tool comes with modeling guidelines proposed in [SCS-SDVST].
- The SCADE toolchain provides qualified tools for model-based design and verification, including production of design documentation, requirements-based Model-in-the-Loop testing, formal verification, and structural coverage analysis at model-level.
- The SCADE Suite KCG (and SCADE ACG) code generator(s) have been qualified and they generate MISRA C:2012 (and AUTOSAR) compliant C source code.

A detailed analysis of the level of support of the SCADE toolchain for the product development at the software level is provided in Appendix C.1.

5
SPECIFICATION OF SOFTWARE REQUIREMENTS

5.1 Objectives and Work Products

The objective of this sub-phase (Clause 6 of [ISO 26262-6:2018]) is to refine the software safety requirements and the other software requirements[6] to ensure that they are:

- suitable for software development
- compliant and consistent with the technical safety requirements
- compliant with the system design
- consistent with the hardware-software interfaces

The inputs for the specification of software requirements sub-phase are:

- technical safety requirements (TSR)
- technical safety concept (TSC)
- system architectural design specification
- hardware-software interfaces specification
- documentation of the software development environment

Work products are:

- software requirements specification
- hardware-software interfaces specification (refined)
- software verification report (initial)

5.2 Requirements and Recommendations

Specification of the software safety requirements shall consider:

- system and hardware configurations
- hardware-software interface specification
- relevant requirements of the hardware design specification
- timing constraints
- the external interfaces
- each operating mode and each transition between the operating modes of the vehicle, the system, or the hardware, having an impact on the software

The software requirements shall be verified to provide evidence for:

- suitability for software development
- compliance and consistency with the technical safety requirements
- compliance with the system design
- consistency with the hardware-software interface

[6] *In the rest of this document, we will use "software requirements" to designate the complete set of software requirements, incl. safety-related ones and others. We will use "software safety requirements" to designate the safety-related ones.*

5.3 Specification of Software Requirements with Ansys SCADE, Medini and VRXPERIENCE

In a strict linear or V-shaped system development process, requirements are established and just trickle down to software development. They can be imported into SCADE from any Requirements Management tool with full traceability. The modeling capabilities of SCADE help analyzing and understanding the requirements more formally and unambiguously.

However, today's cyclic and agile approaches are gaining momentum and can perfectly be applied to safety-related product development with SCADE: executable models and early validation by simulation help to understand and continuously improve the requirements, to evaluate them with customers and in real or simulated road tests and to move some of the required verification and validation activities upfront, where costs of failures are much lower.

Collaboration between a system engineer, a safety engineer, and a simulation engineer to establish an AEB system and software requirements is illustrated by Figure 33.

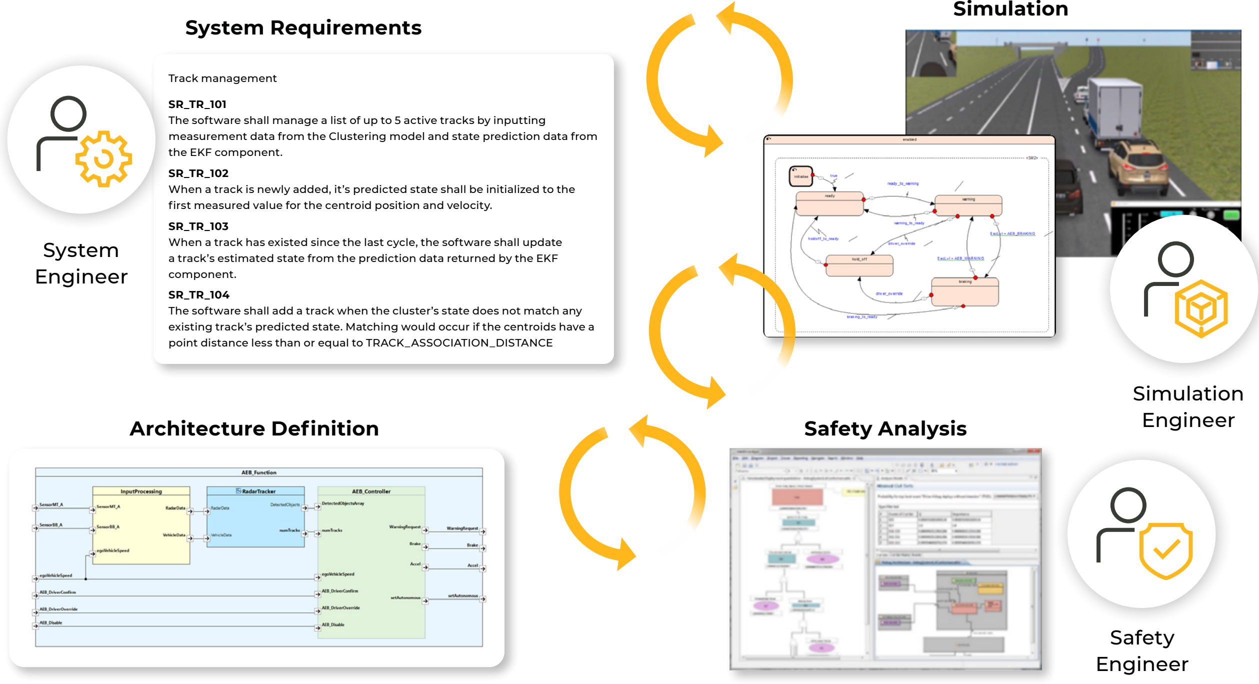


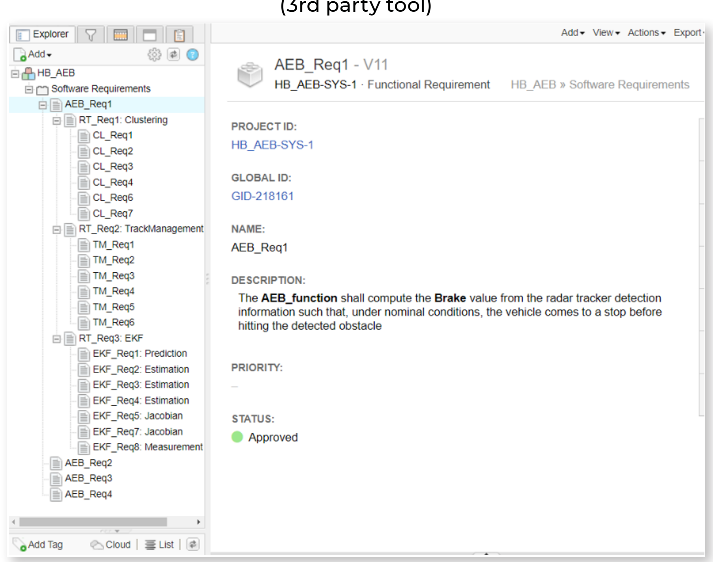
FIGURE 33: A MULTI-DISCIPLINARY APPROACH TO THE CREATION OF AN AEB SYSTEM AND ITS SOFTWARE REQUIREMENTS

In this example, collaboration is based on using of the following tools, together with the user's tool of choice for managing requirements:

- medini Analyze (functional safety analysis) [Ansys medini]
- SCADE Architect (system and software architectural design) [Ansys SCADE]
- VRXPERIENCE (driving scenario simulation) [Ansys VRXPERIENCE]

The specification of the software requirements of the above AEB function is illustrated in Figure 34. The high-level system architecture has been defined with SCADE Architect, and system safety analyses conducted thanks to Ansys medini, leading to the production of the Technical Safety Concept (TSC). Safety requirements and functional requirements are allocated to system components. Functional requirements are refined into software and hardware requirements during the system architectural design process.

Requirements view
(3rd party tool)



Requirements view
(as imported in SCADE)

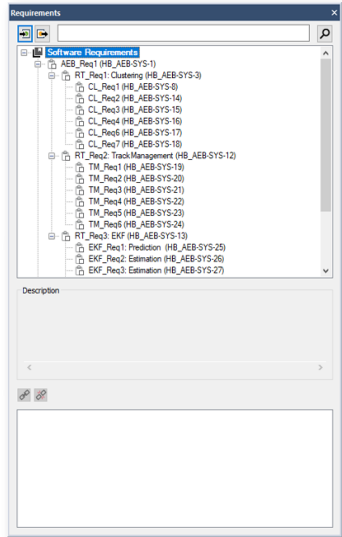


FIGURE 34: SOFTWARE REQUIREMENTS SPECIFICATION OF THE AEB FUNCTION

In this example, the software requirements are described textually for the top-level AEB_function using a third-party requirement management tool and are imported within SCADE for traceability purpose by using the SCADE LifeCycle ALM Gateway. These requirements are then refined into component-level software requirements for the Radar_Tracker and AEB_controller sub-functions during the software architectural design process (see Section 6.3).

Note that, in the above example, we are considering a typical ADAS/AV application where VRXPERIENCE is used to perform system simulation of driving scenarios to explore the Operational Design Domain (ODD) of the application.

However, if we were to consider vehicle electrification as an example, we would use instead Twin Builder [Ansys Twin Builder] that provides an integrated solution to simulate highly complex, electric vehicle and hybrid-electric vehicle powertrains and their sub-systems, to develop applications such as battery management system (BMS), electric power steering (EPS), etc.

The use of VREXPERIENCE or Twin Builder to conduct early simulations can lead to the modification of the system specifications and thus the software requirements.

5.4 Takeaway from Using the Ansys Toolchain to Specify the Software Requirements

In this Chapter, we have very briefly introduced the complete Ansys toolchain (medini, SCADE, VRXPERIENCE, and Twin Builder) to refine the software safety requirements, making sure they comply with the technical safety requirements (TSR), and are suitable for product development at the software level.

Key points of the Ansys toolchain can be listed as follows:

- This is a model-based system engineering approach (MBSE) that relies on SysML modeling language for medini and SCADE Architect, together with the connection to requirements management tools through the SCADE LifeCycle ALM Gateway.
- This is a simulation-based approach that relies on system simulation tools such as VRXPERIENCE and Twin Builder.
- This is an integrated approach with established communication between the specification, design, analysis, and simulation tools.
- The toolchain can be easily connected to other tools, such as ALM tools for project and requirements management.

A detailed analysis of the level of support of the Ansys toolchain for the specification of the software requirements is provided in Appendix C.2.

6

SOFTWARE ARCHITECTURAL DESIGN

6.1 Objectives and Work Products

The objective of this sub-phase (Clause 7 of [ISO 26262-6:2018]) is to develop the software architectural design which represents the software architectural elements and their interactions in a hierarchical structure. Static aspects, such as interfaces between the software components, as well as dynamic aspects, such as process sequences and timing behavior, are described.

The inputs for the software architectural design sub-phase are:

- system architectural design specification
- hardware-software interfaces specification
- software requirements specification

Work products are:

- software architectural specification
- safety analysis report
- dependent failure analysis report
- software verification report (initial)

Verification activities ensures that the software architecture:

- is suitable to satisfy the software safety requirements with the required ASIL, and the other software requirements
- is compatible with the target environment
- is conform to architectural design guidelines

6.2 Requirements and Recommendations

According to Clause 7.4.3 of [ISO 26262-6:2018], and to avoid systematic faults, the software architectural design shall exhibit the following characteristics:

- consistency
- comprehensibility, simplicity, and verifiability
- modularity and encapsulation
- maintainability

In addition, the following Tables describe notations, principles, and methods that could be used to achieve software architecture design and verification requirements.

TABLE 9: NOTATIONS FOR SOFTWARE ARCHITECTURAL DESIGN

Source: Table 2 in ISO 26262-6:2018

Notations		ASIL			
		A	B	C	D
1a	Natural language[a]	++	++	++	++
1b	Informal Notations	++	++	+	+
1c	Semi-formal notations[b]	+	+	++	++
1d	Formal notations	+	+	+	+

[a] *Natural language can complement the use of notations for example where some topics are more readily expressed in natural language or providing explanation and rationale for decisions captured in the notation.*

[b] *Semi-formal notations can include pseudocode or modelling with UML®, SysML®, Simulink® or Stateflow®.*

TABLE 10: PRINCIPLES FOR SOFTWARE ARCHITECTURAL DESIGN

Source: Table 3 in ISO 26262-6:2018

Principles		ASIL			
		A	B	C	D
1a	Appropriate hierarchical structure of the software components	++	++	++	++
1b	Restricted size and complexity of software components[a]	++	++	++	++
1c	Restricted size of interfaces[a]	+	+	+	++
1d	Strong cohesion within each software component[b]	+	++	++	++
1e	Loose coupling between software components[b,c]	+	++	++	++
1f	Appropriate scheduling properties	++	++	++	++
1g	Restricted use of interrupts[a,d]	+	+	+	++
1h	Appropriate spatial isolation of the software components	+	+	+	++
1i	Appropriate management of shared resources[e]	++	++	++	++

[a] *In principles 1b, 1c, and 1g “restricted” means to minimize in balance with other design considerations.*

[b] *Principles 1d and 1e can, for example, be achieved by separation of concerns which refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concept, goal, task, or purpose.*

[c] *Principle 1e addresses the management of dependencies between software components.*

[d] *Principle 1g can include minimizing the number, or using interrupts with a clear priority, in order to achieve determinism.*

[e] *Principle 1i applies for shared hardware resources as well as shared software resources in the case of coexistence. Such resource management can be implemented in software or hardware and includes safety mechanisms and/or process measures that pre-vent conflicting access to shared resources as well as mechanisms that detect and handle conflicting access to shared resources.*

TABLE 11: METHODS FOR VERIFICATION OF THE SOFTWARE ARCHITECTURAL DESIGN

Source: Table 4 in ISO 26262-6:2018

Methods		ASIL			
		A	B	C	D
1a	Walk-through of the design[a]	++	+	o	o
1b	Inspection of the design[a]	+	++	++	++
1c	Simulation of dynamic behaviour of the design	+	+	+	++
1d	Prototype generation	o	o	+	++
1e	Formal verification	o	o	+	+
1f	Control flow analysis[b]	+	+	++	++
1g	Data flow analysis[b]	+	+	++	++
1h	Scheduling analysis	+	+	++	++

[a] *In the case of model-based development, these methods can also be applied to the model.*

[b] *Control and data flow analysis can be limited to safety-related components and their interfaces.*

6.3 Software Architectural Design with SCADE Architect, SCADE Suite, and SCADE LifeCycle

As explained in Clause 7 of [ISO 26262-6:2018], the software requirements are refined through one or more iterations in the software architectural design process to develop the software architecture. The design flow with SCADE is illustrated in Figure 35 and is detailed in the next sections. It is generally made of two parts:

1. The part that is designed in SCADE and that generally relates to the embedded application software
2. The part that may be designed otherwise and can relate to basic software, low-level libraries, or legacy software that is not re-designed in SCADE

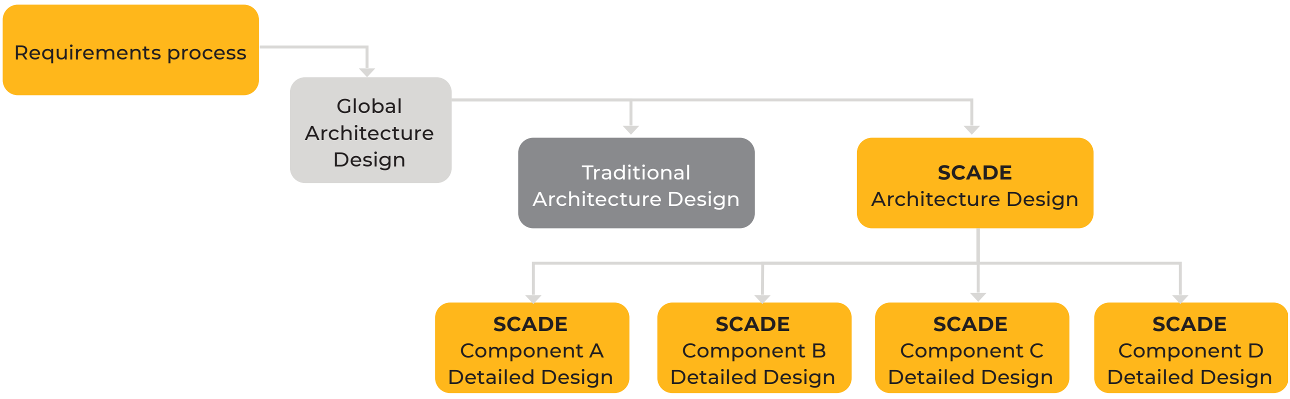


FIGURE 35: THE SOFTWARE ARCHITECTURAL DESIGN PROCESS WITH SCADE

We now explain how the combination of SCADE Architect and SCADE Suite can be used to create the software architectural design.

6.3.1 Global architectural design

The first step in the design process is to define the global application architecture, considering the use of SCADE Architect and SCADE Suite, which can be combined with more traditional techniques.

The application functionality is decomposed into its main components. The characteristics of these components serve as a basis for allocating their refinement in terms of techniques (Scade, C, …) and team. Among those characteristics, one must consider, for a software component:

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

Scade is well-adapted to the functional parts of the software, such as decision logic, filtering, regulation. It is much less appropriate for low-level software such as hardware drivers, interrupt handlers, and encoding/decoding routines, which are more likely to be developed in languages such as C.

6.3.2 Software architectural design with SCADE Architect, SCADE Suite, and SCADE LifeCycle

SCADE Architect is an architecture tool that supports SysML modeling of functions and architecture and that may be used both for the components that will later be implemented as SCADE Suite designs, and the components that will be designed otherwise. It supports customizable attributes, checks and reporting. It can import SysML models from other tools.

Coming back to the AEB example of Figure 33 and Figure 34, we now address the software architectural design with SCADE Architect. Figure 36, Figure 37, and Figure 38 illustrate an iterative architecture decomposition process where the AEB software requirements are refined and allocated to the different software components at every level of the hierarchy (from the top-level AEB function to the leaf components of the Radar_Tracker).

This process is supported with the combined use of SCADE Architect for the architecture decomposition and SCADE LifeCycle ALM Gateway for connection to the software requirements.

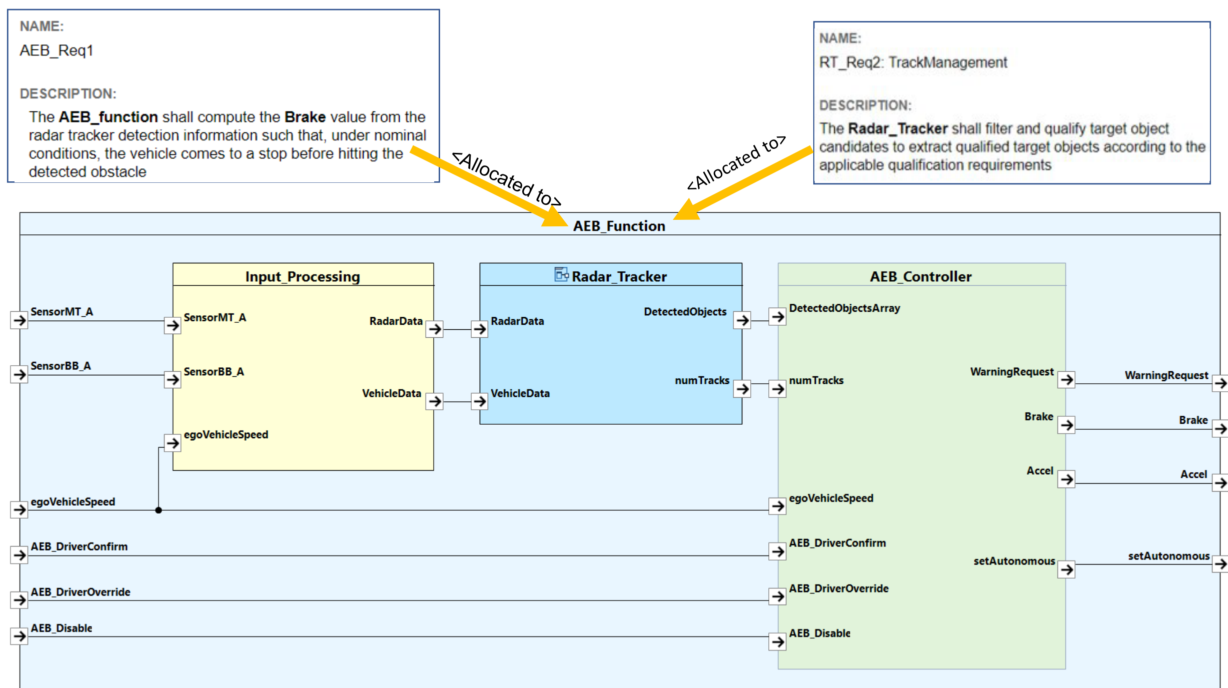


FIGURE 36: TOP-LEVEL AEB SOFTWARE ARCHITECTURE IN SCADE ARCHITECT AND ALLOCATION OF SOFTWARE REQUIREMENTS

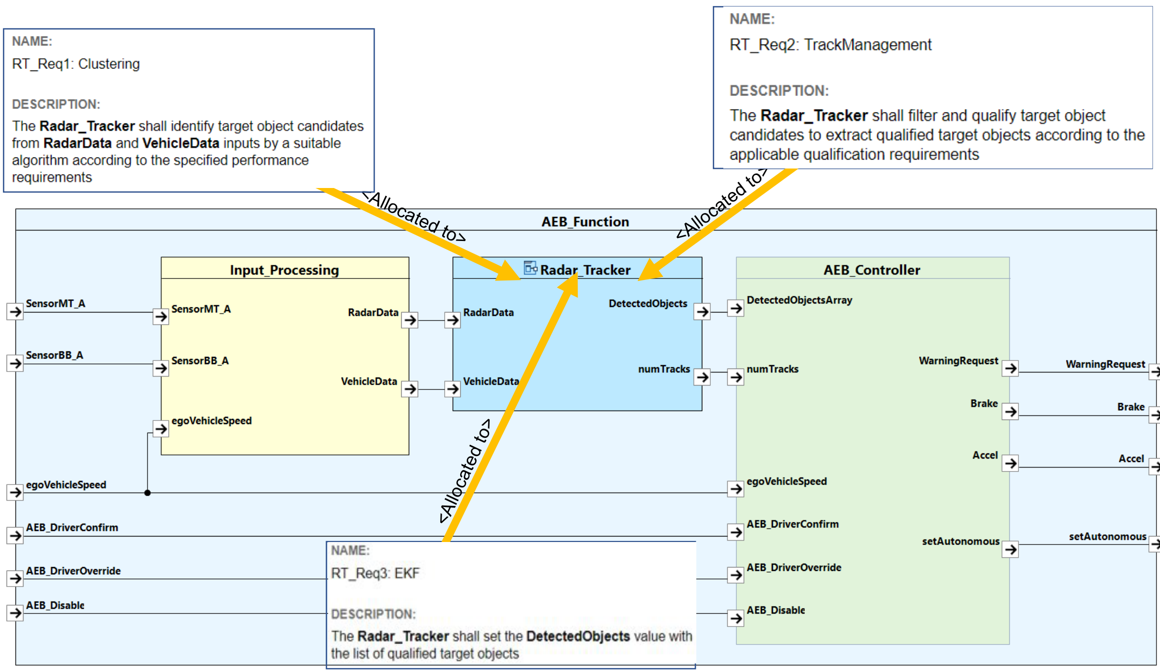


FIGURE 37: REFINEMENT OF AEB FUNCTION SOFTWARE REQUIREMENTS AND ALLOCATION TO RADAR_TRACKER COMPONENT

We now continue the decomposition of the Radar_Tracker architecture into its components (clustering, prediction and estimation), as shown in Figure 38, together with the allocation of software requirements to these components. We see here that these requirements combine English text and mathematical equations, which will later be very practical for their implementation as Scade equations (see Section 7.3.1).

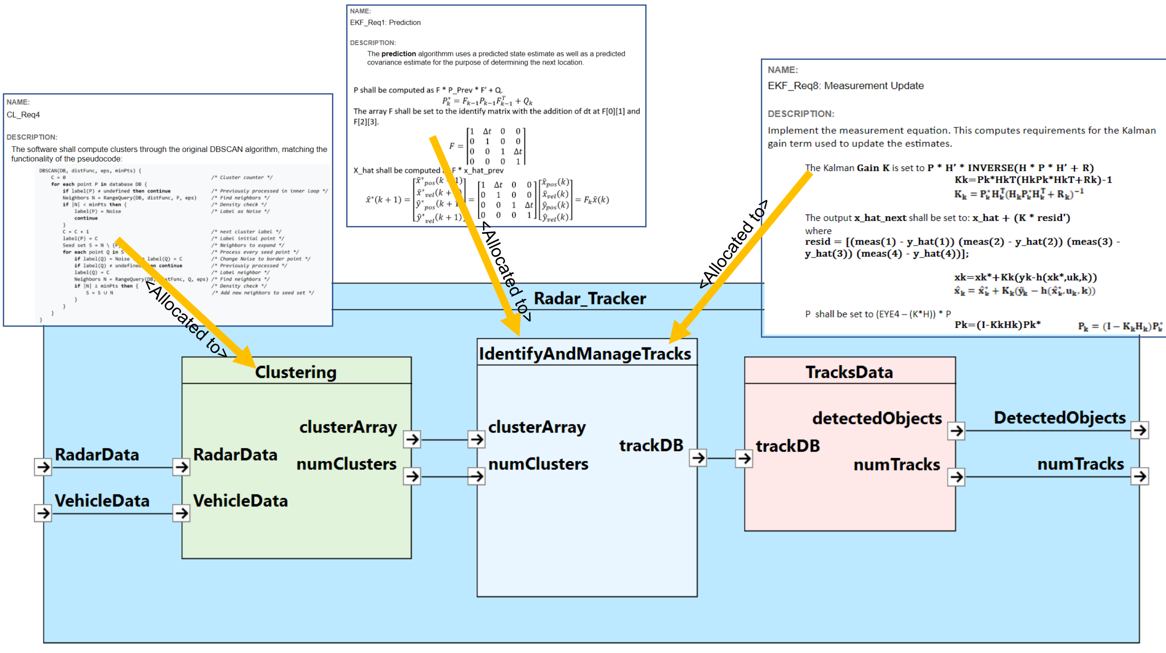


FIGURE 38: REFINEMENT OF THE RADAR TRACKER SOFTWARE REQUIREMENTS AND ALLOCATION TO THE LEAF COMPONENTS

For the components that will be implemented in SCADE Suite, Figure 39 illustrates how a software architecture model designed in SCADE Architect (on the left) can be synchronized with the corresponding SCADE Suite architecture model (on the right). The elements of the SCADE Architect design then become the top-level blocks of the SCADE Suite software design.

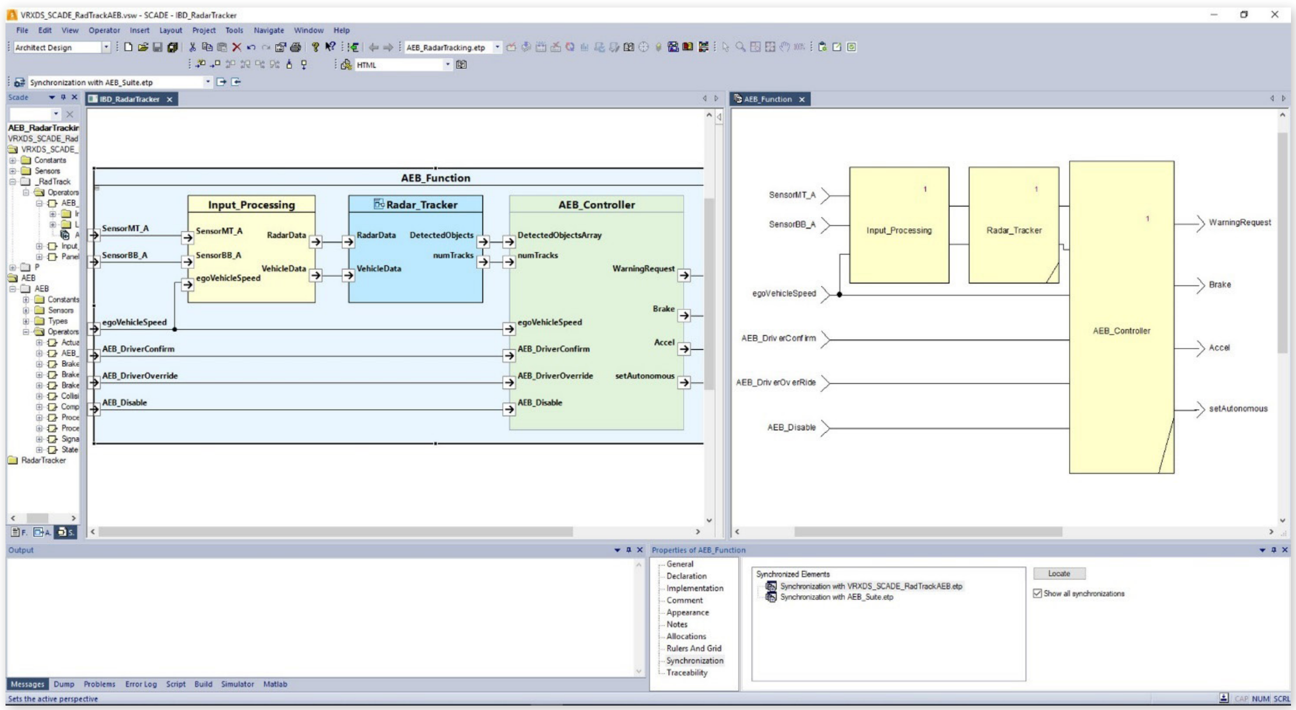


FIGURE 39: SCADE ARCHITECT AND SCADE SUITE SYNCHRONIZATION

As shown in Figure 40, the software architectural design in SCADE Suite becomes the starting point for the software unit design that will be detailed in the next Section. The strongly typed interface of SCADE Suite ensures consistency of the software design across multiple software teams.

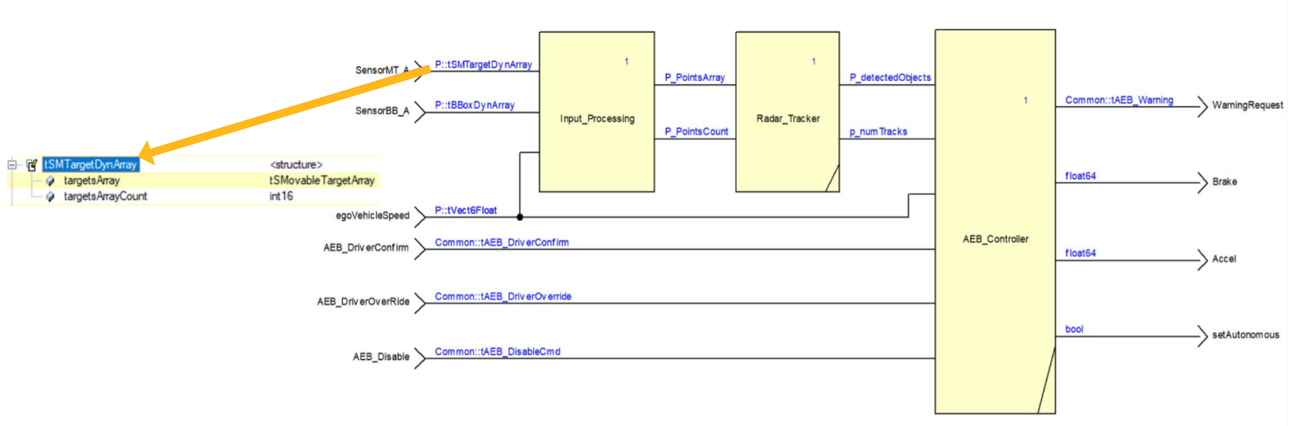


FIGURE 40: THE AEB SOFTWARE ARCHITECTURAL DESIGN IN SCADE SUITE

The purpose of the software architectural design model is to:

- identify high-level functions: typically, one develops a functional breakdown down to a depth of two or three levels of hierarchy
- define the interfaces of these functions: names, data types
- describe the data flows and control flows between these functions
- verify consistency of the data flows between these functions using SCADE Suite semantic checks
- prepare the framework for the detailed design process: define the top-level functions while ensuring consistency of their interfaces

6.4 Takeaway from Using SCADE Architect, SCADE Suite, and SCADE LifeCycle for Software Architectural Design

In this Chapter, we have seen how SCADE proposes both a semi-formal approach based on the SysML standard, and a formal approach based on the Scade language for describing the software architecture at the start the software unit design and implementation sub-phase that will be detailed in the next Chapter.

SCADE Architect and SCADE Suite, together with SCADE LifeCycle, efficiently support the requirements and recommendations of [ISO 26262-6:2018] regarding software architectural design:

- consistency: this comes with the rules checking that is available both in SCADE Architect and SCADE Suite, including rules from SysML and Scade, user-defined rules
- comprehensibility, simplicity, and verifiability: these come naturally through the simple and intuitive graphical symbology of SCADE Architect and SCADE Suite, but it also requires minimizing the complexity of the models
- modularity and encapsulation: both SCADE Architect and SCADE Suite promote modular and hierarchical designs
- maintainability: the architecture must be designed in such a way that the team has a stable framework during the initial development as well as when there are updates

However, there is no magic recipe for achieving a good model architecture with SCADE products. It requires a mix of experience, creativity, and rigor.

Here are a few suggestions:

- be reasonable and realistic: nobody can build a good architecture in one shot. Do not develop the full model from the first draft, but build two or three architecture variants, then analyze and compare them. You may otherwise have to live with a bad architecture for a long time
- review and discuss the architecture with peers
- select the architecture that minimizes connection complexity and is robust to change.

A detailed analysis of the level of support of SCADE Architect, SCADE Suite, and SCADE LifeCycle for software architectural design is provided in Appendix C.3.

7

SOFTWARE UNIT DESIGN AND IMPLEMENTATION

7.1 Objectives and Work Products

The objective of this sub-phase (Clause 8 of [ISO 26262-6:2018]) is to develop a software unit detailed design in accordance with the software architectural design, the design criteria and the allocated software requirements which supports the implementation and verification of the software unit; and to implement the software units as specified.

The inputs to the software unit design and implementation sub-phase are:

- software architectural specification
- hardware-software interfaces specification
- software requirements specification
- software verification report
- configuration data and calibration data, if any

Work products are:

- software unit design specification
- software unit implementation
- software verification report (refined)

7.2 Requirements and Recommendations

According to Sections 8.4.2/3/4/5 of [ISO 26262-6:2018], and in order to avoid systematic faults, the software unit design:

- shall be described in a notation that exhibits the following characteristics:
  - consistency
  - comprehensibility
  - verifiability
  - maintainability
- shall be sufficiently detailed so that it can be implemented

Furthermore, design and implementation principles shall be applied to achieve the following properties:

- correct execution order of functions
- correct and consistent description of interfaces, data and control flows
- simplicity and readability
- robustness

In addition, the following Tables describe notations and principles that could be used to achieve software unit design and implementation requirements.

TABLE 12: NOTATIONS FOR SOFTWARE UNIT DESIGN

Source: Table 5 in ISO 26262-6:2018

Notations		ASIL			
		A	B	C	D
1a	Natural language[a]	++	++	++	++
1b	Informal notations	++	++	+	+
1c	Semi-formal notations[b]	+	+	++	++
1d	Formal notations	+	+	+	+

[a] *Natural language can complement the use of notations for example where some topics are more readily expressed in natural language or provide an explanation and rationale for decisions captured in the notations.*

[b] *Semi-formal notations can include pseudocode or modelling with UML®, SysML®, Simulink® or Stateflow®.*

TABLE 13: PRINCIPLES FOR SOFTWARE UNIT DESIGN AND IMPLEMENTATION

Source: Table 6 in ISO 26262-6:2018

Principles		ASIL			
		A	B	C	D
1a	One entry and one exit point in subprograms and functions[a]	++	++	++	++
1b	No dynamic objects or variables, or else online test during their creation[a]	+	++	++	++
1c	Initialization of variables	++	++	++	++
1d	No multiple use of variable names[a]	++	++	++	++
1e	Avoid global variables or else justify their usage[a]	+	+	++	++
1f	Restricted use of pointers[a]	+	++	++	++
1g	No implicit type conversions[a]	+	++	++	++
1h	No hidden data flow or control flow	+	++	++	++
1i	No unconditional jumps[a]	++	++	++	++
1j	No recursions	+	+	++	++

[a] *Principles 1a, 1b, 1d, 1e, 1f, 1g and 1i may not be applicable for graphical modelling notations used in model-based development.*

7.3 Software Unit Design with SCADE Suite

Once the SCADE Suite architecture has been defined, the software units detailed design can be achieved in SCADE Suite. The objective of this activity is to produce a set of complete and consistent SCADE Suite design models.

As shown in Section 3.2.1 of this handbook, the Scade language efficiently supports the requirements and recommendations of [ISO 26262-6:2018] regarding software unit design and implementation, and other good practices for the development of safety-related embedded software.

In this section, we first consider software unit design with SCADE Suite and in the next section, we will consider the implementation with SCADE Suite KCG automatic code generation.

Key characteristics of the Scade language are:

- strong typing
- concurrency
- modularity
- re-usable components

On this basis, software requirements can be mapped to Scade design elements with a granularity that is determined by the user:

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

The following sections provide examples of Scade modeling patterns that illustrate the above concepts for various types of algorithms.

7.3.1 Filtering and control

Filtering and control algorithms are usually designed by control engineers. Their design is often formalized in the form of block diagrams and transfer functions defined in terms of "z" expressions.

The SCADE Suite graphical notation allows representing block diagrams exactly in the same way as control engineers, using the same semantics. The Scade time operators fit the z operator of control engineering. For instance, the z-1 operator of control engineering (meaning a unit delay) has equivalent operators called "pre" and "fby" in the Scade language.

For example, if a control engineer has written an equation such as:

$$Y(z) = K_1 U(z) - K_2 z^{-1} Y(z)$$

which corresponds in the discrete time domain to:

$$y_k = K_1 u_k - K_2 y_{k-1}$$
$$y_0 = init$$

This can be expressed textually in Scade as:

```
y = init -> K1*u-K2*pre(y)
```

or graphically, as shown in Figure 41 below.

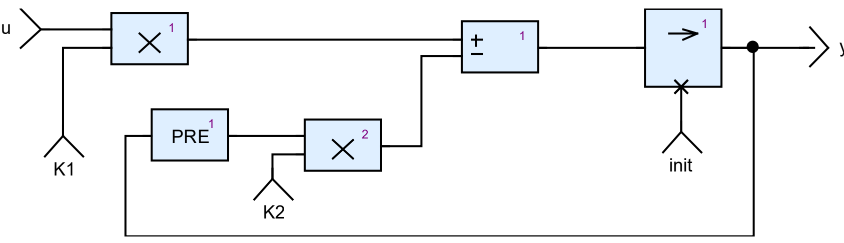


FIGURE 41: A FIRST ORDER FILTER

It is possible to implement both Infinite Impulse Response (IIR) and Finite Impulse Response (FIR) filters. In a FIR filter, the output depends on a finite number of past input values; in an IIR filter such as the one above, the output depends on an infinite number of past input values because there is a loop in the diagram.

Thanks to its built-in generic map and fold array operators, the Scade language can readily express complex controls involving large data structures. This is illustrated in the tracking algorithm below, which is implementing the IdentifyAndManageTracks component of Radar_Tracker in Figure 38. Existing tracks are updated, new tracks are created, and stale tracks are deleted.

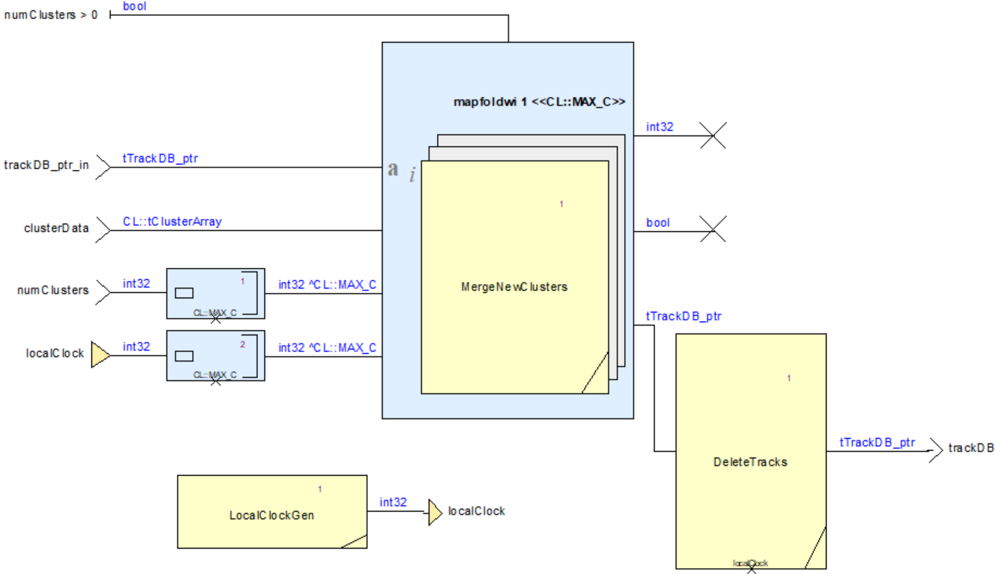


FIGURE 42: ALGORITHM TO ITERATE EACH DETECTED CLUSTER OF RADAR POINTS THROUGH EXISTING TRACK DATABASE

7.3.2 Decision logic

In safety-related embedded software, decision logic is often more complex than filtering and control.

The controller must handle:

- identification of the situation
- detection of abnormal conditions
- decision making
- management of redundant computation chains

In Scade, a variety of techniques are available for handling decision logic:

- logical operators (such as and/or/xor) and comparators
- selecting flows, based on conditions, with the “if” and “case” construct
- building complex functions from simpler ones. SCADE Suite supports encapsulation and modularity with the concept of user-defined operators
- conditional activation of operators depending on Boolean conditions
- state machines that in Scade, unlike in some other languages, are always fully deterministic (*e.g.*, for each situation where more than one transition could be possible, there is always an explicit priority)

In Figure 43, we give an example of a typical state machine as it could appear in an AEB system.

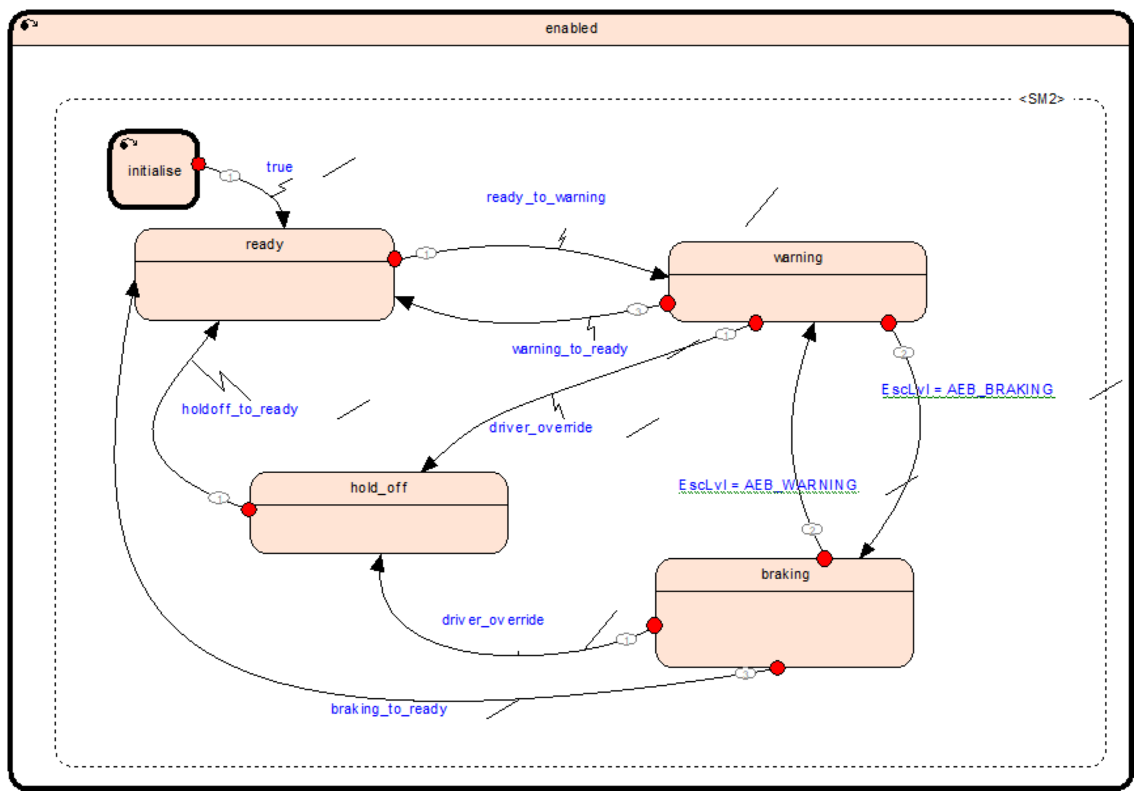


FIGURE 43: A SCADE STATE MACHINE DESCRIBING THE AUTOMATIC EMERGENCY BRAKING (AEB) DECISION LOGIC

When starting with SCADE Suite, one may ask which of the above-mentioned techniques to select for describing decision logic. Here are some hints for the selection of the appropriate technique.

Selecting state machines or logical expressions:

- Does the output depend on the past? If it only depends on current inputs, this is just combinatorial logic: simply use a logical expression in the data flow. A state machine that jumps to state Xi when condition Ci is true independently from current state, is degraded and does not need to be a state machine.
- Does the state have strong qualitative influence on behavior? This favors a state machine.

7.3.3 Re-usable components and library management

A SCADE Suite library[7] object can be developed as any other SCADE Suite software component, considering the following:

- Library components are usually identified during the design process of a given application and can be considered in most cases as implementation choices, not necessarily described in the upper-level software requirements of the application.
- Good practices consist in defining functional requirements, or assumptions and guarantees of their usage, for these library components as a separate document and in developing and verifying the components from its requirements.
- When a library is shared between several applications, a self-contained development package may be considered, including its own project plans and standards, requirements, design data, verification reports, safety analyses reports, quality assurance reports and software configuration management reports.

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

Section 12 of [ISO 26262-8] provides guidance regarding the “Qualification of software components” which has the objective of providing their “suitability in terms of re-use in items developed in compliance with the ISO 26262 series of standards”. Requirements for the software components should be available, together with relevant design, implementation, and verification artefacts, depending on the ASIL of the application software that is including the software component.

Some general-purpose components (*e.g.*, matrix product, integrator, rising edge detector) should not be redone and maintained multiple times but should rather be shared among projects in a library. Some libraries may also be managed for sharing components at the application level (special type of filter). Development and verification artifacts are managed in shared libraries.

Using library operators has advantages:

- It saves time.
- It relies on validated components.
- It makes models more readable and maintainable. For instance, a call to an Integrator is much more readable than the set of lower-level operators and connections that implement an Integrator.
- It enforces consistency throughout the project.
- It factors the code.

7.3.4 Scade language concepts for re-usability

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

- library
- genericity/polymorphism
- parameterization by size

Figure 44 shows a predefined SCADE Suite library (libmath.etp as mathematical library can be re-used for application design). Users can create their own library and reference them in the upper-level application (*e.g.*, libmath library in ACC project).

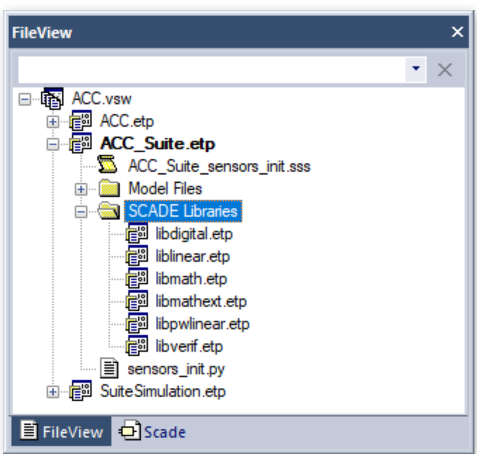


FIGURE 44: CONCEPT OF SCADE SUITE LIBRARY

A library may include generic operators (called polymorphic operators). Such operators are defined independently from the type of their arguments and can be instantiated with various types. The Figure below illustrates a GenericToggle operator instantiated once with integer and another time with Boolean.

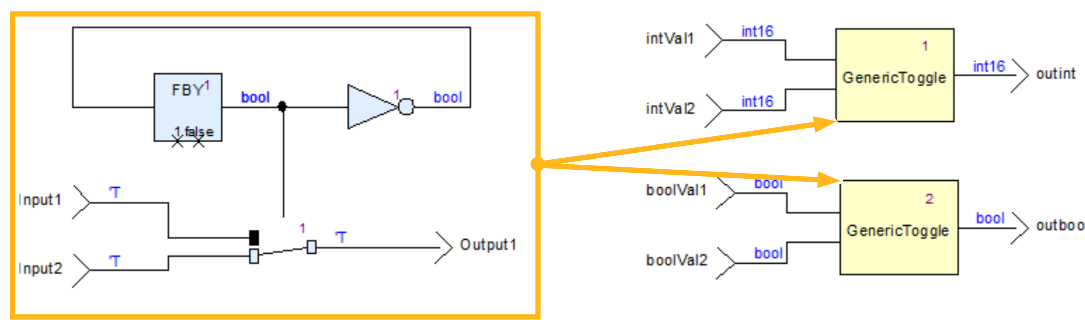


FIGURE 45: EXAMPLE OF A GENERIC OPERATOR INSTANTIATED WITH INT AND BOOL TYPES

For algorithms on arrays (iterative schemes based on map and fold operators), the size of input/output arrays for an operator can be parameterized. The size identifier is part of the formal interface of this operator. Figure 46 shows an operator (MaxParametric) that computes the maximum value of a set of integer values implemented as an array. It is parameterized by size and can be instantiated with a constant value (literal 5 in this example).

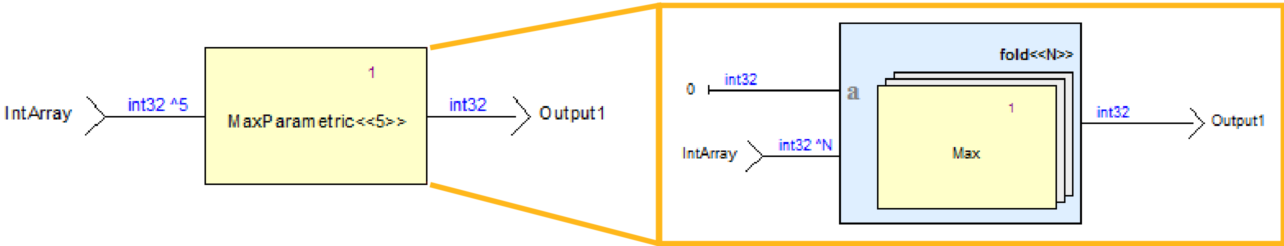


FIGURE 46: EXAMPLE OF AN OPERATOR PARAMETERIZED BY SIZE

7.3.5 Robustness management

As required by Clause 7.4.12 of ISO 26262-6:2018, "safety mechanisms for error detection and error handling shall be applied", and Clause 8.4.5 of ISO 26262-6:2018, "Design principles for software unit design and implementation at the source code level shall be applied to achieve: … f) robustness", we will now propose ways to address robustness systematically.

Robustness of safety-related embedded software cannot be addressed locally. It requires a general robustness policy for the whole system and should be addressed at each step of the development and verification processes:

1. The robustness policy should be defined while setting up the development environment and related guidelines such as the Software Design Standards (see Chapter 4 of this handbook and proposed SCADE Suite modeling guidelines in [SCS-SDVST]).
2. There should be explicit decisions about robustness and failure handling in the software requirements. The software requirements, including requirements for library components, should specify responses to abnormal input data and to any invalid data that may be produced by computation described in the software requirements (e.g., for X=Y/Z, the requirement should specify the expected behavior for Z near zero, except if there is evidence that Z is far from zero, or more precisely that Y/Z cannot generate a division by zero). This is required to achieve accuracy and determinism of requirements and to perform requirements-based testing for robustness tests.
3. The robustness policy should be addressed in the Software Architectural Design document. As an example, the way for handling arithmetic exceptions should be defined at this global level.

COMMUNICATION WITH THE EXTERNAL ENVIRONMENT

As recommended by Clause 7.4.12 NOTE 2, “Safety mechanisms for error detection can include ... plausibility checks”, we propose to establish a mandatory design rule to never trust an external input without appropriate verification and to build consolidated data from the appropriate combination of available data.

By using SCADE Suite component libraries one can, for instance, insert:

- a voting function
- a low pass filter and/or limiter for a numeric value
- a Confirmator for Boolean values, as shown in Figure 47

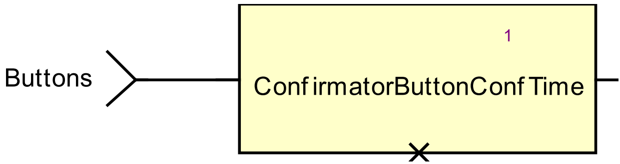


FIGURE 47: INSERTING A CONFIRMATOR IN A BOOLEAN INPUT FLOW

Plausibility checks also apply to the detection of unintended changes in calibration data, as recommended by Table C.1 of ISO 26262-6:2018.

In a similar way, outputs to actuators must be value-limited and rate-limited, which can be ensured by inserting Limiter operators before the output, as shown in Figure 48 below.

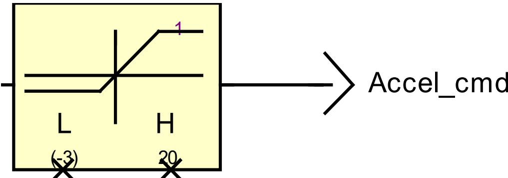


FIGURE 48: INSERTING A LIMITER IN AN OUTPUT FLOW

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

DEFENSIVE PROGRAMMING

As recommended by Clause 5.4.3 (Table 1, point 1d), defensive programming is a well-known technique to make a design robust.

It means the following:

- Normal and abnormal input domains are identified.
- The Scade operator is designed in a way that it reacts safely to abnormal inputs.
- It is not critical for the environment of this function to care about normal conditions.

For example, such a defensive programming strategy for a square root operator amounts to implementing a specific behavior (according to the upper-level requirements) when the input is negative.

This approach is systematic, and the direct benefit is robustness. The potential drawback is run-time cost, even in cases when there is evidence that the normal conditions hold, for example square root of (x**2+y**2).

Another alternative to optimize run-time efficiency is to consider a design by contract approach as presented below.

DESIGN BY CONTRACT

This approach allows for alleviating the design from the overhead of some defensive constructs when given assumptions hold true on a given operator. For instance, the assumption for a non-robust square root function is that the input is non-negative. In this context, it is the responsibility of the SCADE Suite operator using the square root function to ensure that this assumption is fulfilled.

The Scade language supports defining assumptions (called *'assume'*) as illustrated below in red.

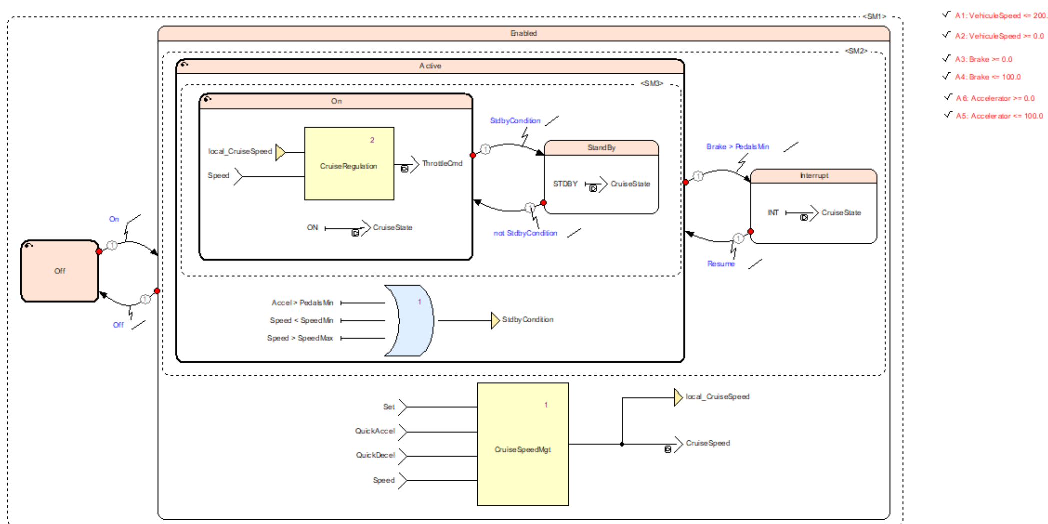


FIGURE 49: EXAMPLE ASSUMPTIONS FOR AN ACC OPERATOR

In addition to assumptions, the Scade language supports claims enabling to formalize some operator properties (called "guarantee").

SCADE Suite Design Verifier can be used to check that the guarantees always hold provided that the assumptions are fulfilled. This is described in Section 8.3.2/Formal verification of functional properties.

This approach is efficient from an run-time performance point of view, as it does not need to design-in the defensive constructs as described in the previous section, but extreme care must be taken when verifying design b -contract designs.

Figure 50 presents an example of robust architecture mixing the two approaches.

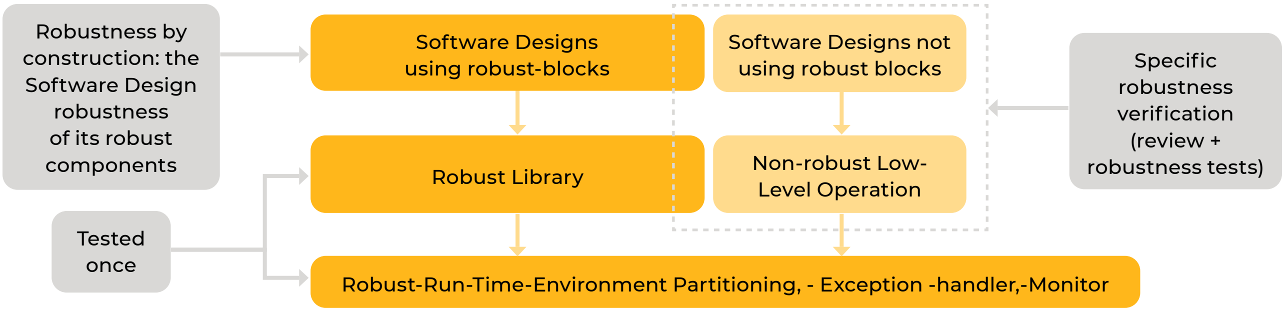


FIGURE 50: EXAMPLE OF ROBUST ARCHITECTURE

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

- The corresponding software application inherits robustness from its low-level robust components.
- The verification strategy of such robust components is optimized because the library operator is tested once according to its robustness requirements.

On the right part, the approach is not optimal because the low-level operations are not systematically robust: a specific and integral robustness analysis is required to ensure the robustness of the whole software application and the corresponding verification effort should be higher.

See Section 9.4 for more information about the verification strategy regarding robustness of a SCADE Suite application.

7.4 Software Unit Implementation with SCADE Suite KCG

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

As illustrated by Figure 51 below, a typical software design and implementation process will combine the SCADE flow with automatic code generation with a traditional flow involving manual coding.

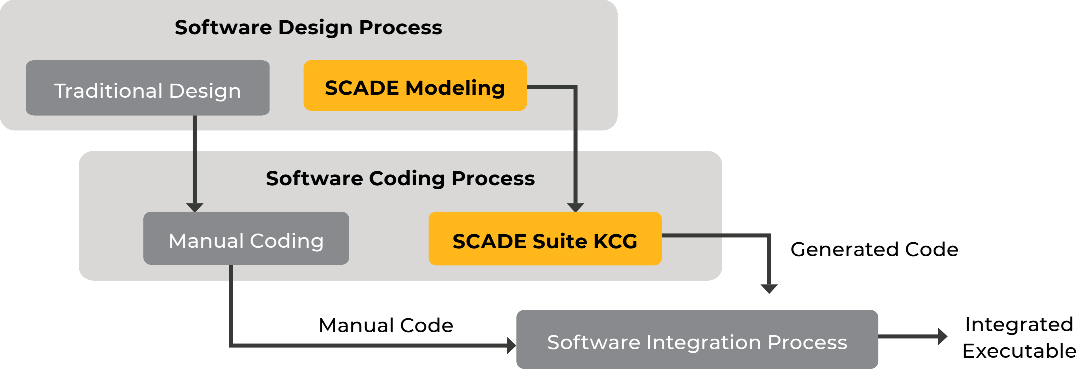


FIGURE 51: THE SOFTWARE CODING AND INTEGRATION PROCESS WITH SCADE SUITE

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

7.4.1 Properties of the generated code

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

- The code is portable: it is ISO C [ISO-IEC-9899] compliant.
- The code is MISRA C [MISRA C:2012] compliant (see Section 8.3.4 for more details).
- The code structure reflects the model architecture for data-flow parts when there is no expansion and/or optimization during code generation. For control-flow parts, traceability between state names and C code is ensured.
- The code is readable and traceable to the input model using corresponding names, specific comments, and a traceability file.
- Memory allocation is fully static (no dynamic memory allocation).
- There is no recursive call.
- Only bounded loops are allowed, since they use constant values known at code generation time.
- Execution time is bounded.
- Expressions are explicitly parenthesized.

- No dynamic address calculation is performed (no pointer arithmetic).
- There are no implicit type conversions.
- There is no expression with side-effects (no i++, no a += b, no side-effect in function calls).
- No functions are passed as arguments.

Traceability from the generated code to a SCADE Suite data flow is illustrated in Figure 52.

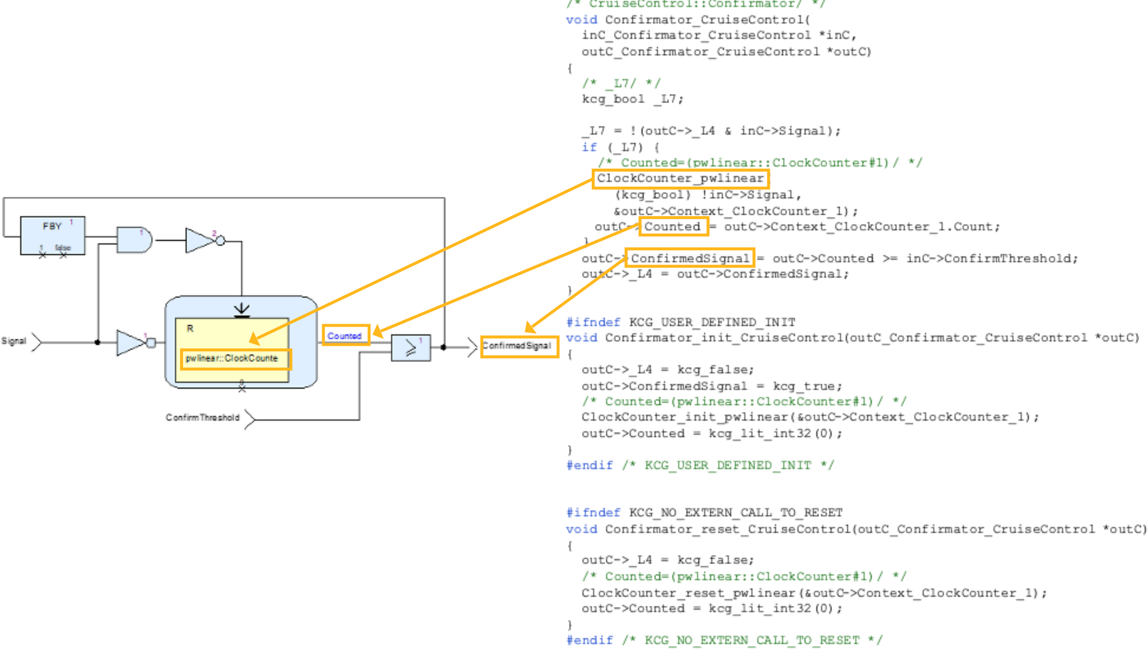


FIGURE 52: SCADE SUITE DATA FLOW TO GENERATED C SOURCE CODE TRACEABILITY

Traceability from the generated code to a SCADE Suite state machine is illustrated in Figure 53.

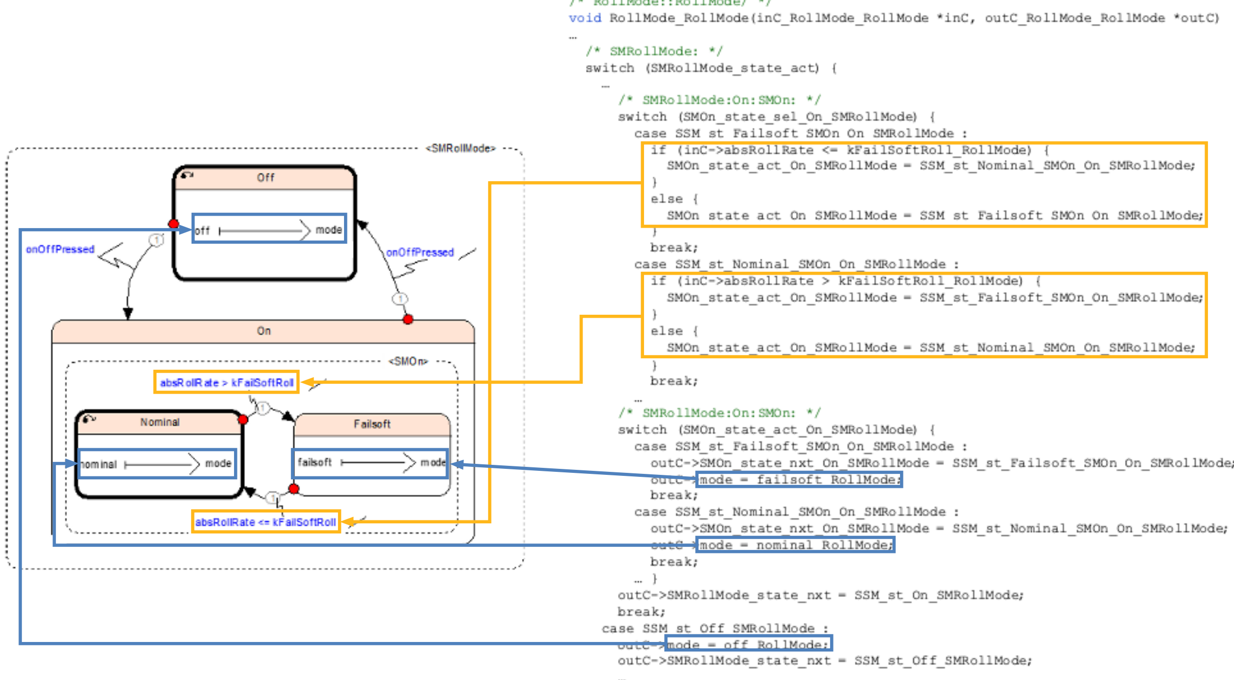


FIGURE 53: SCADE SUITE STATE MACHINE TO GENERATED C SOURCE CODE TRACEABILITY

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

7.4.2 Tuning code to target and project constraints

Various code generation options can be used to tune the generated code to a particular target and project constraints. Static analysis methods are available in SCADE Suite, using SCADE Suite Timing and Stack Optimizer/Verifier (TSO/TSV) to help tuning the code generation options for performance. Specified as a Scade model, the application software can be analyzed from the execution time point of view allowing to tune modeling choices and code generation options according to users' needs. Basically, there are two ways to generate code from an operator:

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

This is illustrated in Figure 54.

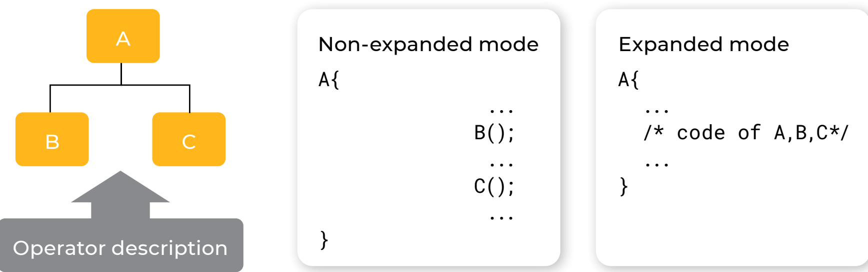


FIGURE 54: NON-EXPANDED AND EXPANDED MODES

Both code generation modes (Non-expanded or Expanded) can be composed at will, performing a call for some operators and inlining for other operators.

Note that the expansion directives (see Non-expanded mode and Expanded mode above) and some interface directives (see definition below about **`global_root_context`** option and **`separate_io`** option/pragma) may have an impact on the structure of the generated code, on the integration of the generated code, and even on the verification strategy.

These options and directives can be considered as a design choice and should be identified very early in the software development life cycle, preferably during architecture decomposition:

- The **`global_root_context`** SCADE Suite KCG option is a code generation mode where the inputs, outputs and context variables of the root operators are defined as C global variables and not passed as arguments of the root C functions. This change on the signature of root C functions impacts the integration of KCG generated code.
- The **`separate_io`** SCADE Suite KCG option and/or pragma applies to an operator. When it is set, the code generated for the cycle function is different: outputs are no more in the context but passed as separate parameters. As for the global root context, it impacts integration of the generated code.

7.4.3 Code generation from multiple software units

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

This process is usually sufficient because it ensures global consistency of the code generated from a single SCADE Suite component. Yet, it may not be appropriate in the context of complex software

architecture, or when having libraries. A complex SCADE Suite application can result from several components (interacting or not together) where each component corresponds to a single library model with a given root node. It is the case for instance, when the SCADE Suite application includes several tasks, and each task is designed with a separate model.

As shown in Figure 55, there are two alternatives for generating code:

- Generating all code in one run, using the "multi-root operators" SCADE Suite KCG option (see [Ansys_SCADE] for further information on options). This applies whether root operators are defined in the same model or not. When operators do not belong to the same model, a new integration model, which references the input models as libraries, is created (see integration model in Figure 55).
- Generating code for each root node separately and then integrating all C generated codes into the application.

The coding process described in the first alternative is highly recommended unless there is a major reason for not using it. It is the safest and cleanest way to integrate the different root nodes. It is also highly recommended as a means for performing verification and validation of the global behavior.

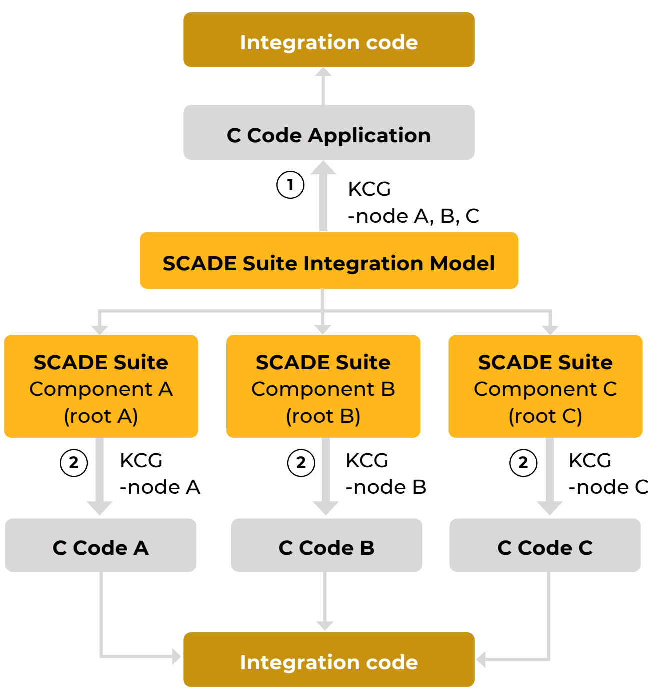


FIGURE 55: CODE GENERATION FROM MULTIPLE COMPONENTS

Even if the use of some KCG directives such as manifest pragma and/or global prefix option (see below) may support the application of the second alternative, it requires a strict coding and integration process with additional verification activities to check the consistency of the interfaces and of the integration:

- The manifest pragma is used to control the type names generated by KCG. It ensures better stability of the code between two code generation sessions.
- The global prefix KCG option is used to prevent name conflicts during integration of generated code. It adds a prefix (user-specified) in front of the names of C global identifiers.

We will further discuss the software integration process in Chapter 9.

7.5 Takeaway from Using SCADE Suite for Software Unit Design and Implementation

As described in Chapter 3 of this handbook, the Scade notation of SCADE Suite fulfils the requirements and recommendations of [ISO 26262-6:2018] regarding software unit design and implementation, as it has been formally defined (see Table 13), and it exhibits the following characteristics:

- consistency
- comprehensibility
- verifiability
- maintainability

In addition, the definition of the Scade language and the implementation of the SCADE Suite KCG code generator guarantee that the software unit design and implementation principles of Table 14 are obeyed:

- Scade generated code only has one entry and exit point.
- The generated code is MISRA compliant.
- No dynamic objects are created at run-time.
- Initialization of variables is statically verified.
- Variable scope is controlled.
- There are no user-defined global variables. There are no user-defined pointers.
- There are no implicit conversions.
- There are no hidden data or control flows.
- There are no jumps.
- There is no recursion.

The Scade language includes both a graphical and a textual representation. It supports a unified modeling style that enables the design of complex algorithms and complex decision logic. Both styles can be combined without restriction while the modularity of the design is continuously supported.

A detailed analysis of the level of support of SCADE Suite for the software unit design and implementation provided in Appendix C.4.

8 SOFTWARE UNIT VERIFICATION

8.1 Objectives and Work Products

The objectives of this sub-phase (Clause 9 of [ISO 26262-6:2018]) are to:

- provide evidence that the software unit design satisfies the allocated software requirements and is suitable for implementation
- verify that the defined safety measures are properly implemented
- provide evidence that the implemented software unit complies with the unit design and fulfils the allocated software requirements with the required ASIL
- provide sufficient evidence that the software unit contains neither undesired functionalities nor undesired properties regarding functional safety

The inputs to the software unit verification sub-phase are:

- software architectural specification
- hardware-software interfaces specification
- software requirements specification
- configuration data and calibration data, if any
- software unit design specification
- software unit implementation
- software verification report

Work products are:

- software verification specification
- software verification report (refined)

8.2 Requirements and Recommendations

According to Section 9.4.2 of [ISO 26262-6:2018], the software unit design shall be verified to provide evidence for:

- compliance of the unit design with the software requirements
- compliance of the source code with the unit design
- compliance of the implementation with the hardware-software interface
- confidence in the absence of unintended functionality
- sufficient resources to support the functionality
- implementation of the safety measures

In Sections 9.4.2/3/4, the following Tables describe methods that could be used to achieve the above software unit verification requirements.

TABLE 14: METHODS FOR SOFTWARE UNIT VERIFICATION

Source: Table 7 in ISO 26262-6:2018

Methods		ASIL			
		A	B	C	D
1a	Walk-through[a]	++	+	o	o
1b	Pair-programming[a]	+	+	+	+
1c	Inspection[a]	+	++	++	++
1d	Semi-formal verification	+	+	++	++
1e	Formal verification	o	o	+	+
1f	Control flow analysis[b, c]	+	+	++	++
1g	Data flow analysis[b, c]	+	+	++	++
1h	Static code analysis[d]	++	++	++	++
1i	Static analyses based on abstract interpretation[e]	+	+	+	+
1j	Requirements-based test[f]	++	++	++	++
1k	Interface test[g]	++	++	++	++
1l	Fault injection test[h]	+	+	+	++
1m	Resource usage evaluation[i]	+	+	+	++
1n	Back-to-back comparison test between model and code, if applicable[j]	+	+	++	++

[a] *For model-based development these methods are applied at the model level, if evidence is available that justifies confidence in the code generator used.*

[b] *Methods 1f and 1g can be applied at the source code level. These methods are applicable both to manual code development and to model-based development.*

[c] *Methods 1f and 1g can be part of methods 1e, 1h or 1i.*

[d] *Static analyses are a collective term which includes analysis such as searching the source code text or the model for patterns matching known faults or compliance with modelling or coding guidelines.*

[e] *Static analyses based on abstract interpretation are a collective term for extended static analysis which includes analysis such as extending the compiler parse tree by adding semantic information which can be checked against violation of defined rules (e.g. data-type problems, uninitialized variables), control-flow graph generation and data-flow analysis (e.g. to capture faults related to race conditions and deadlocks, pointer misuses) or even meta compilation and abstract code or model interpretation.*

[f] *The software requirements at the unit level are the basis for this requirements-based test. These include the software unit design specification and the software safety requirements allocated to the software unit.*

[g] *This method can be used to provide evidence for the integrity of used and exchanged data.*

[h] *In the context of software unit testing, fault injection test means to modify the tested software unit (e.g. introduce faults into the software) for the purposes described in 9.4.2. Such modifications include injection of arbitrary faults (e.g. by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).*

[i] *Some aspects of the resource usage evaluation can only be performed properly when the software unit tests are executed on the target environment or if the emulator for the target processor adequately supports resource usage tests.*

[j] *This method requires a model that can simulate the functionality of the software units. Here, the model and code are stimulated in the same way and results compared with each other.*

EXAMPLE In the case of model-based design results of non-floating-point operations can be compared.

TABLE 15: METHODS FOR DERIVING TEST CASES FOR SOFTWARE UNIT TESTING

Source: Table 8 in ISO-26262-6:2018

Methods		ASIL			
		A	B	C	D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes[a]	+	++	++	++
1c	Analysis of boundary values[b]	+	++	++	++
1d	Error guessing based on knowledge or experience[c]	+	+	+	+

[a] *Equivalence classes can be identified based on the division of inputs and outputs, such that a representative test value can be selected for each class.*

[b] *This method applies to interfaces, values approaching and crossing the boundaries and out of range values.*

[c] *Error guessing tests can be based on data collected through a "lessons learned" process and expert judgment.*

TABLE 16: STRUCTURAL COVERAGE METRICS AT THE SOFTWARE UNIT LEVEL

Source: Table 9 in ISO 26262-6:2018

Methods		ASIL			
		A	B	C	D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage)	+	+	+	++

NOTE 3: In the case of model-based development, the analysis of structural coverage can be performed at the model level using analogous structural coverage metrics for models.

EXAMPLE: 4 The analysis of structural coverage performed at the model level can replace the source code coverage metrics if it is shown to be equivalent, with rationales based on evidence that the coverage is representative of the code level.

NOTE : If instrumented code is used to determine the degree of structural coverage, it can be necessary to provide evidence that the instrumentation has no effect on the test results. This can be done by repeating representative test cases with non-instrumented code.

Finally, Section 9.4.5 of [ISO 26262-6:2018] requires that "the test environment for software unit testing shall be suitable for achieving the objectives of the unit testing considering the target environment".

8.3 Software Unit Verification with SCADE Suite, SCADE Test Environment for Host, and SCADE LifeCycle

According to the methods listed in Table 15 and Table 16, and considering the fact that, with SCADE Suite, we are in the case of model-based development and qualified code generation, the software unit verification activities are performed at model-level.

We will first focus on the following model-level verification steps:

- model accuracy and consistency
- compliance of the model with the software requirements
- compatibility of the model with the target computer

8.3.1 Model accuracy and consistency

Since SCADE Suite relies on the Scade formal notation, the corresponding design models are formally verifiable.

Such verification is handled by SCADE Suite Semantic Checker[8] that performs an in-depth analysis of the software unit design consistency, including:

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

8.3.2 Compliance of the model with the software requirements

Compliance of the software units design with the software requirements is verified through a combination of techniques applied to the Scade models:

- peer reviews (Walk-through and Inspection)
- Model-in-the-Loop (MiL) testing
- formal verification

PEER REVIEWS WITH SCADE LIFECYCLE REPORTER

Peer reviews can be performed based on the report generated by SCADE LifeCycle Reporter.

According to the recommendations and requirements of Section 9.4.2 of [ISO 26262-6:2018], at design model level, these reviews will focus on the following points:

- **traceability** between software requirements and software units design models
- **compliance** of the software unit design models with the software requirements
- **robustness analysis** of the software units design

The notation used for SCADE Suite models has several advantages, while performing design reviews, compared to other approaches:

- Its formal definition: the description is not subject to interpretation
- Its graphical representation is simple and intuitive

SCADE LifeCycle Reporter has been qualified for [ISO 26262:2018] at TCL3. This qualification ensures completeness and consistency of the generated report according to the input model. For further details on SCADE LifeCycle Reporter qualification, see Appendix E.3.

To achieve an agile development workflow, SCADE LifeCycle Reporter has been complemented by SCADE LifeCycle Model Change a tool that is able to determine which parts of a Scade model have been changed in a given iteration, thus allowing the user to only review the modified parts of the model, as shown in Figure 56 .

[8] *SCADE Suite Semantics Checker is made of the front-end module of SCADE Suite KCG which has been qualified at TCL3, and therefore its results can be trusted.*

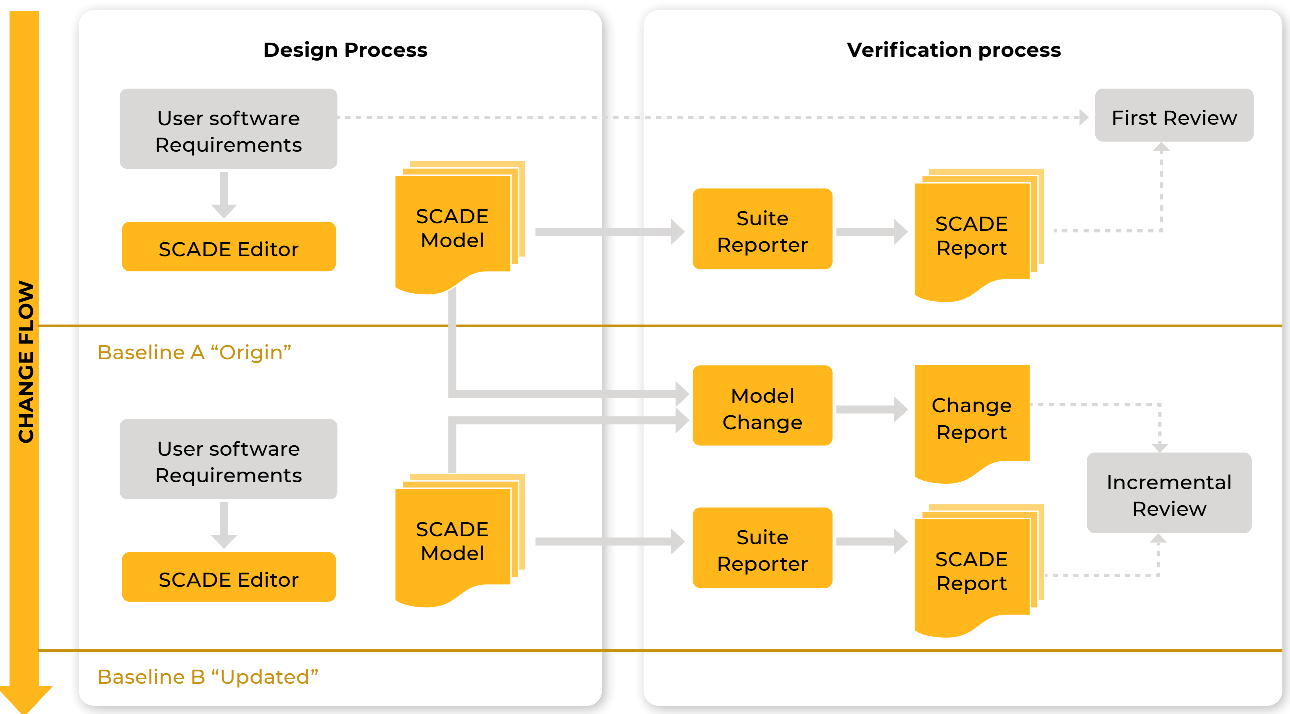


FIGURE 56: INCREMENTAL REVIEWS WITH SCADE LIFECYCLE MODEL CHANGE

SCADE LifeCycle Model Change has been qualified for [ISO 26262:2018] at TCL3. This qualification ensures proper identification of the model parts that have changed when moving to the next iteration, thus making the review process incremental and, overall, more efficient. For further details on SCADE LifeCycle Model Change qualification, see Appendix E.3.

MODEL-IN-THE-LOOP TESTING WITH SCADE TEST ENVIRONMENT FOR HOST

Model-in-the-Loop testing allows exercising the behavior of a model. Its main purpose is to provide repeatable evidence of compliance of the model to the software requirements by exercising requirements-based tests. The position of SCADE Test Environment for Host within the software development and verification flow is shown below.

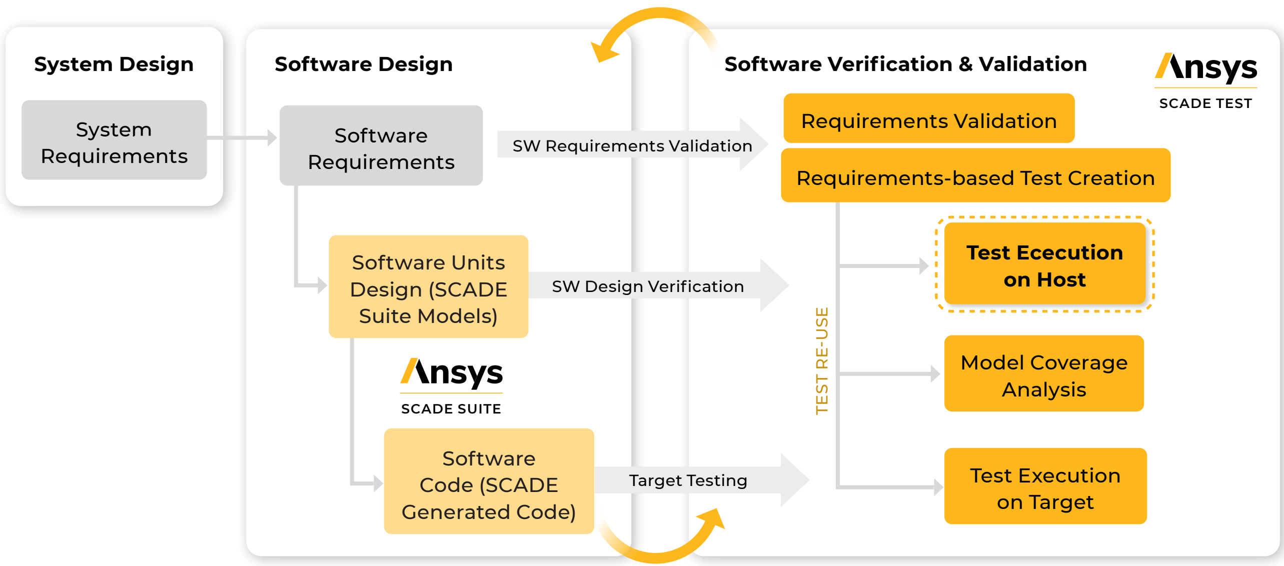


FIGURE 57: POSITIONING OF SCADE TEST ENVIRONMENT FOR HOST WITHIN THE VERIFICATION FLOW

Moreover, Model-in-the-Loop testing is an efficient way to detect functional issues very early in the software unit design and/or software requirements.

Testing Scade models requires the following activities:

- Test cases and procedures are developed from the software requirements from which the Scade model was developed.
- Test cases and procedures shall address both testing with correct input value ranges and robustness testing.
- Traceability between software requirements and test cases and procedures is established.
- Test cases and procedures are reviewed to confirm that they are correct *wrt.* software requirements and test strategy.
- Scade models are exercised by software requirements-based test cases and procedures in the host environment.
- SCADE Model-in-the-Loop testing results are reviewed to confirm that they are complete and correct, and all deficiencies are explained.

Note 1: The above requirements-based test cases and procedures will first be used to perform Model-in-the-Loop testing of the Scade models, as described above will then be re-used to perform Executable Object Code (EOC) testing on target (see Section 9.4)

Note 2: Integration of the software application will be performed in steps (see Integration objectives in Section 10.1 of [ISO 26262-6:2018]). When we refer to EOC in Note1 and in Section 9.4 of this handbook, we do not mean the complete EOC of the software application, we mean the executable code corresponding to the SCADE part(s) of the software application that are integrated.

SCADE Test Environment for Host provides an integrated environment that allows verification engineers to both create and manage test cases (see Figure 58) and then to run on host the test cases created from the software requirements (see Figure 59).

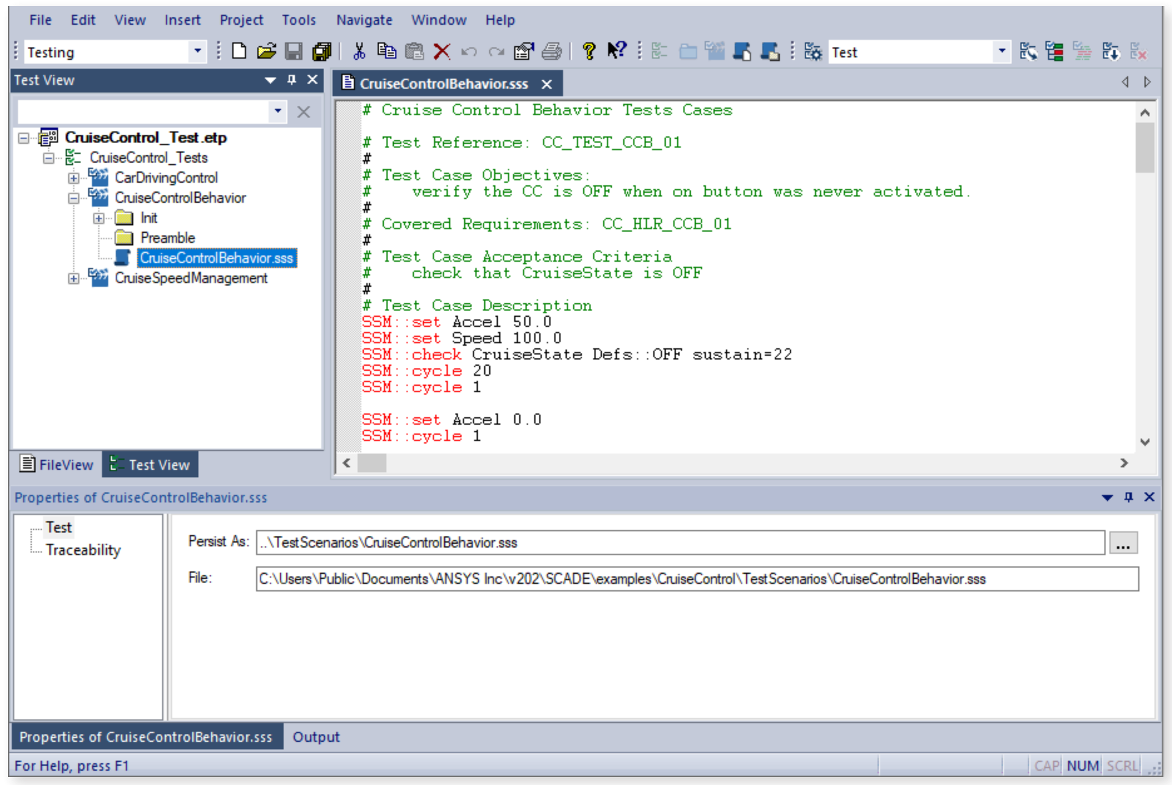
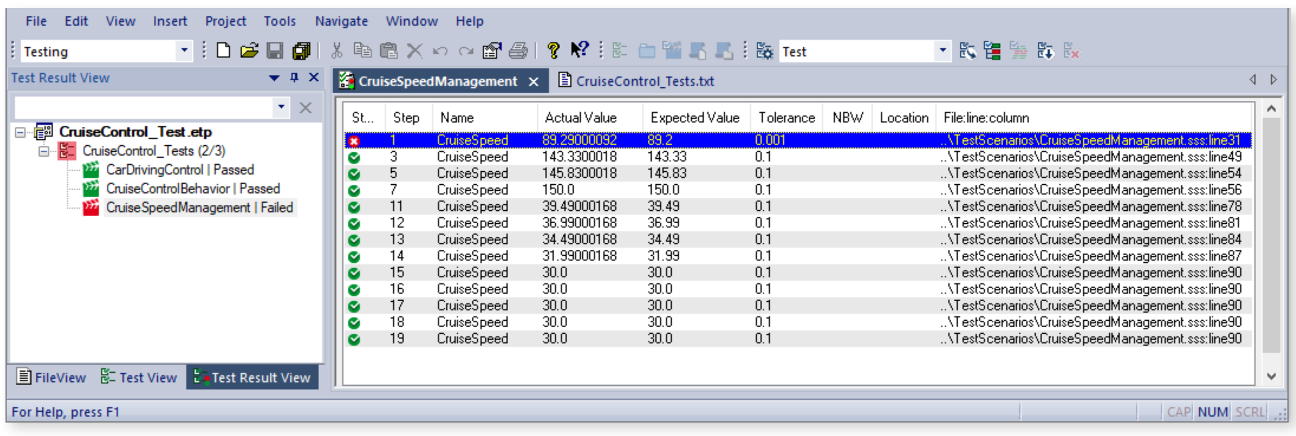
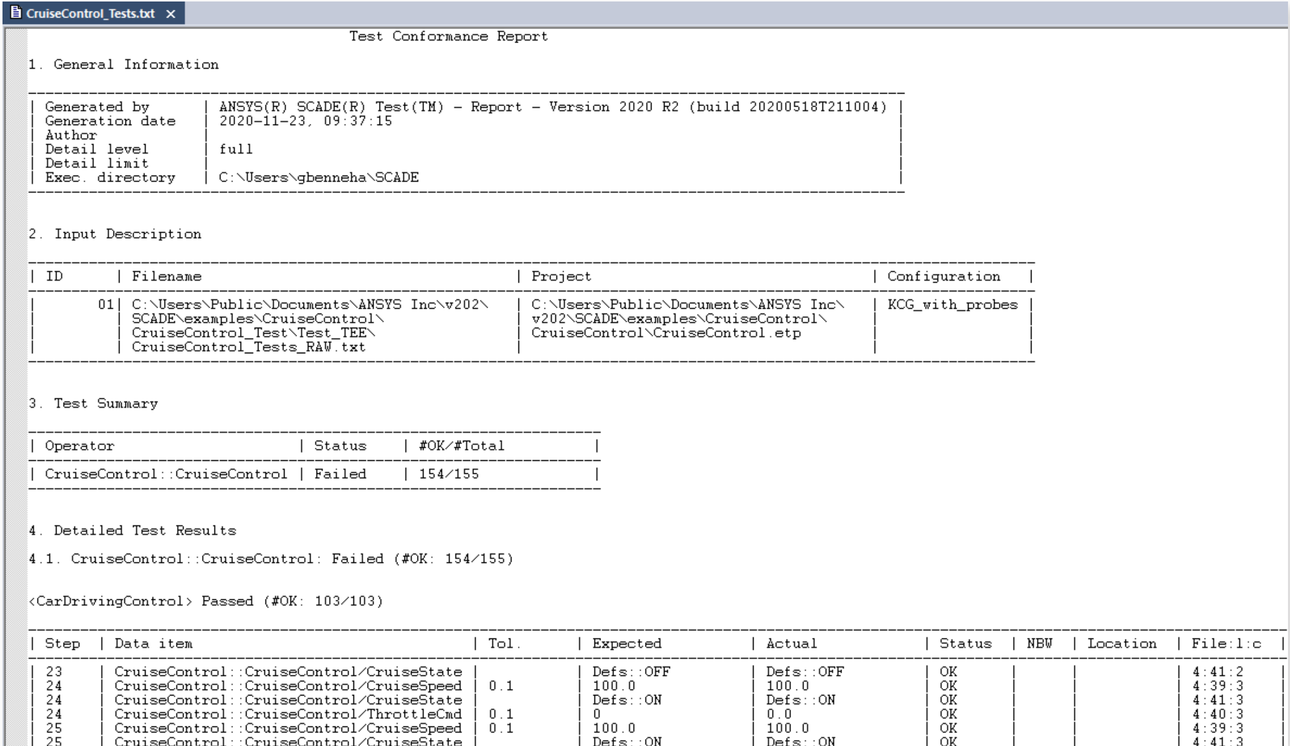


FIGURE 58: TEST CASES CREATION AND MANAGEMENT IN SCADE TEST ENVIRONMENT FOR HOST

With SCADE Test Environment for Host, the generation of test result reports (containing expected and testing results) is automated, enabling significant time and cost savings over manual verification.



St...	Step	Name	Actual Value	Expected Value	Tolerance	NBW	Location	File:line:column
	1	CruiseSpeed	89.29000092	89.2	0.001			..\TestScenarios\CruiseSpeedManagement.sss:line31
	3	CruiseSpeed	143.3300018	143.33	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line49
	5	CruiseSpeed	145.8300018	145.83	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line54
	7	CruiseSpeed	150.0	150.0	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line56
	11	CruiseSpeed	39.49000168	39.49	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line78
	12	CruiseSpeed	36.99000168	36.99	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line81
	13	CruiseSpeed	34.49000168	34.49	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line84
	14	CruiseSpeed	31.99000168	31.99	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line87
	15	CruiseSpeed	30.0	30.0	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line90
	16	CruiseSpeed	30.0	30.0	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line90
	17	CruiseSpeed	30.0	30.0	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line90
	18	CruiseSpeed	30.0	30.0	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line90
	19	CruiseSpeed	30.0	30.0	0.1			..\TestScenarios\CruiseSpeedManagement.sss:line90



CruiseControl_Tests.txt

Test Conformance Report

1. General Information

Generated by	ANSYS(R) SCADE(R) Test(TM) - Report - Version 2020 R2 (build 20200518T211004)
Generation date	2020-11-23, 09:37:15
Author	
Detail level	full
Detail limit	
Exec. directory	C:\Users\gbenneha\SCADE

2. Input Description

ID	Filename	Project	Configuration
01	C:\Users\Public\Documents\ANSYS Inc\v202\SCADE\examples\CruiseControl\CruiseControl_Test\Test_TEE\CruiseControl_Tests_RAW.txt	C:\Users\Public\Documents\ANSYS Inc\v202\SCADE\examples\CruiseControl\CruiseControl\CruiseControl.etp	KCG_with_probes

3. Test Summary

Operator	Status	#OK/#Total
CruiseControl::CruiseControl	Failed	154/155

4. Detailed Test Results

4.1. CruiseControl::CruiseControl: Failed (#OK: 154/155)

<CarDrivingControl> Passed (#OK: 103/103)

Step	Data item	Tol.	Expected	Actual	Status	NBW	Location	File:l:c
23	CruiseControl::CruiseControl/CruiseState		Defs::OFF	Defs::OFF	OK			4:41:2
24	CruiseControl::CruiseControl/CruiseSpeed	0.1	100.0	100.0	OK			4:39:3
24	CruiseControl::CruiseControl/CruiseState		Defs::ON	Defs::ON	OK			4:41:3
24	CruiseControl::CruiseControl/ThrottleCmd	0.1	0	0.0	OK			4:40:3
25	CruiseControl::CruiseControl/CruiseSpeed	0.1	100.0	100.0	OK			4:39:3
25	CruiseControl::CruiseControl/CruiseState		Defs::ON	Defs::ON	OK			4:41:3

FIGURE 59: MODEL-IN-THE-LOOP TESTING RESULTS ON HOST

SCADE Test Environment for Host has been qualified for [ISO 26262:2018] at TCL3. The qualification evidence allows users to claim credit from SCADE Test Model-in-the-Loop testing for the verification of the compliance of a SCADE Suite model with its corresponding software requirements. For further details on SCADE Test Environment for Host qualification, see Appendix E.4.

Furthermore, **qualification of both SCADE Test for Model-in-the-Loop testing and SCADE Suite KCG for C source code generation eliminates the need for "Back-to-back comparison test between model and code" (entry 1n of Table 15)**. This is because the generated C source code implements the same behavior as the one that is executed while running Model-in-the-Loop testing.

FORMAL VERIFICATION WITH SCADE SUITE DESIGN VERIFIER

Formal methods are complementary to reviews and test for the verification of software. SCADE Suite Design Verifier[9] provides a powerful verification technique based on formal verification.

Formal software verification consists of a set of activities using a mathematical framework to reason about software behaviors and properties in a rigorous way.

The recipe for formal verification of safety properties is:

1. Define a formal model of the software; namely a mathematical model representing the states of the software and its behaviors
2. Define for the formal model a set of formal properties to verify
3. Perform state space exploration to mathematically analyze the validity of the safety property

When using the Scade language, the model is formal, so there is no additional formalization effort required. This step is automated in SCADE Suite Design Verifier.

Let us now consider two cases related to formal verification of software units using SCADE Suite Design Verifier:

- formal verification related to the use of arithmetic operators
- formal verification of functional properties

— *Formal verification regarding robustness of arithmetic operators*

Here is the list of predefined checks available on arithmetic operators:

- integer division by zero exception
- float division by zero leading to infinite values
- integer arithmetic overflow exception
- float overflow leading to NaN (Not a Number) values

These checks formally verify that the arithmetic operations of a model are always done within their domain of definition. Table 18 below describes, for each option of SCADE Suite Design Verifier (Overflow, Division by Zero, ...) the error detections that are performed depending on the types of the operation (integer, floating point).

TABLE 17: ARITHMETIC ERROR DETECTION PERFORMED WITH SCADE SUITE DESIGN VERIFIER

	Integers		IEEE-754 Floating-point exceptions [IEEE-754]		
SCADE Suite Design Verifier checks options	**Overflow**	**Division by zero**	**Overflow**	**Division by zero**	**Invalid operations**
Overflow	X				X
Division by zero		X			
Infinity			X	X	
Not a Number					X

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

— *Formal verification of functional properties*

Verifying functional properties requires first to formalize the property to be checked. SCADE Suite Design Verifier uses Scade as the property specification language.

Let us take a cruise control system to illustrate the steps and assume one wants to verify the following safety property:

```
"When the Cruise Control is not ON (not regulating),
the throttle must be equal to the accelerator."
```

In a Scade operator, one would express the safety property shown in Figure 60 below, reflecting the above property. This operator is called an observer.

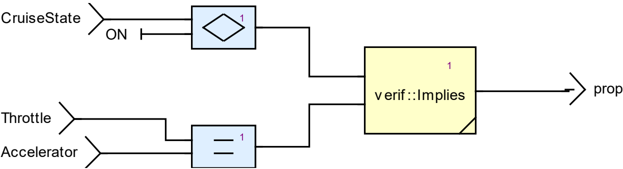


FIGURE 60: OBSERVER OPERATOR CONTAINING THE SAFETY PROPERTY

Then, we would connect the observer operator to the controller in a verification context operator, as shown in Figure 61 below.

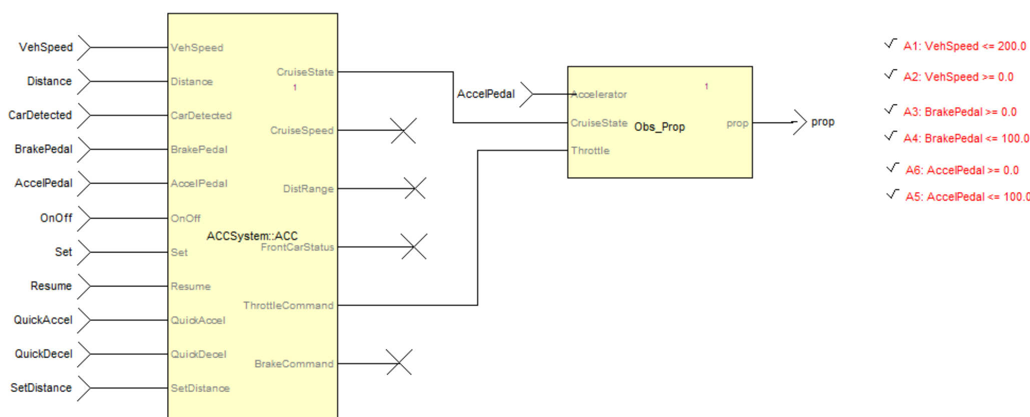


FIGURE 61: CONNECTING THE OBSERVER OPERATOR TO THE CONTROLLER

SCADE Suite Design Verifier then performs automatically and statically the complete state space exploration to mathematically analyze the validity of the functional properties.

Result can be either a sequence of input that invalidates the property, as shown by Figure 62 and Figure 63, or a guarantee that the property holds for any sequence of inputs.

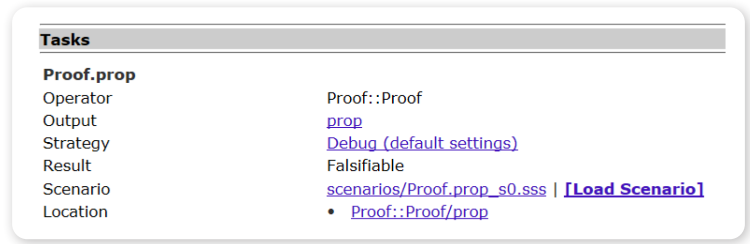


FIGURE 62: EXAMPLE A DESIGN VERIFIER REPORT WHEN A SEQUENCE OF INPUTS INVALIDATES THE PROPERTY

```
# ----------------------------------------------------------------
# Simulation scenario file for SCADE Simulator
# Task:     Proof.prop
# Model:    ACC_Suite
# Operator: Proof::Proof
# ----------------------------------------------------------------

SSM::set Controller::Kv 9.4833979668062001e-302
SSM::set Controller::Kd -3.7524496871499317e-93
SSM::set Controller::Kf -1.9073490893717048e-06
SSM::set Controller::Kb 2.0722615146145237e-317
SSM::set Controller::Kp_t 7.9050503334599447e-323
SSM::set Controller::Ki_t 8.3971397167178263e-320
SSM::set Proof::Proof/VehSpeed 5.4940114179082471
SSM::set Proof::Proof/Distance -3.2733906078961419e+151
SSM::set Proof::Proof/CarDetected t
SSM::set Proof::Proof/BrakePedal 0.8828125
SSM::set Proof::Proof/AccelPedal 1.9999999999999998
SSM::set Proof::Proof/OnOff t
SSM::set Proof::Proof/Set f
SSM::set Proof::Proof/Resume f
SSM::set Proof::Proof/QuickAccel t
SSM::set Proof::Proof/QuickDecel t
SSM::set Proof::Proof/SetDistance f

SSM::cycle
```

FIGURE 63: EXAMPLE OF SEQUENCE PROVIDED TO FALSIFY THE PROPERTY

SCADE Suite Design Verifier is not a qualified tool.

8.3.3 Compatibility with target computer

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

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

- Model complexity
- Execution time and memory consumption
- Compatibility of generated code with target platform

MODEL COMPLEXITY ANALYSIS

ISO 26262-6:2018 recommends establishing modeling guidelines to enforce low complexity (Clause 5.4.3, Table 1/1a) and principles for software architectural design, including restricting the size and complexity of software components (Clause 7.4.3, Table 3/1b).

Our main issue here is to monitor the complexity of Scade models to avoid potential issues during software development and target execution. It is strongly recommended to define rules related to the management of Scade models complexity in a Software Model Standards document (see [SCS-SDVST]).

Two levels of rules must be considered for Scade models:

- **SCADE Suite built-in rules:** they are predefined rules directly from the definition of the Scade formal notation. The Scade Language Reference Manual [SCS-KCG-LRM] defines what a correct Scade model is, and what behavior a correct Scade model defines. The former is called “static semantics” as formally defined in [SCS-KCG-LRM], the latter is called dynamic semantics and is also defined in the same document in a semi-formal way (text and mathematics). The SCADE Suite KCG front-end first implements all the static checks and stops whenever the defined static semantics is not satisfied; then it generates a code that implements the dynamic semantics.
- **User design rules related to Scade models:** they are additional rules defined by the user in its Software Model Standards for readability, verifiability, and maintainability purposes.

Typical model complexity metrics have been defined:

- the number of coverage points (see Section 8.4)
- the maximum number of diagrams for an operator
- the maximum number of user-operators within a diagram
- the maximum number of nested levels of conditional operators

These are defined in the SCADE Suite Development Standards document [SCS-SDVST].

Such rules must be checked either automatically or manually. In the context of automatic verification, the user can develop its own design rules by using SCADE Suite Rules Checker scripting capabilities. For further information on scripting capabilities, refer to SCADE Suite User Manual [Ansys SCADE]. SCADE Suite Rule Checker has not been qualified by Ansys. If user evaluation of this tool leads to a Tool Confidence Level greater than 1, additional verification means must be performed by the tool user.

EXECUTION TIME AND MEMORY CONSUMPTION ANALYSIS

Clause 7.4.13 of ISO 26262-6:2018 requires "an upper estimation of required resources... including a) the execution time; b) the storage space...". Resource usage evaluation is required as a part of software unit verification (Table 7/1m) and as a part of verification of software integration (Table 10/1d).

The objective of the analyses that we propose is to anticipate potential timing and stack usage problems during the software design phase.

— *Timing problems*

The ability of an application to complete its task on time using a given CPU is usually addressed during target integration testing. Schedulability analysis must be performed to demonstrate the properties of the integrated system with respect to timing requirements.

Hence it is necessary to determine an upper bound for execution time, which results from a process called Worst-Case Execution Time (WCET) analysis.

Measurement of WCET raises several challenges that impose major costs and risks on the integration testing phase of any software development project:

- Measurement is only possible when all elements of the system are available: application software, system software, target system, and a complete set of test cases. It is often too late when a problem is found in these project phases. Late changes of software and/or target result in very high costs and risky delays.
- Measurement is not precise or implies code instrumentation which may alter test results in non-predictable ways.
- Tracing of execution time phenomena back to code or even to the model is very tedious, if even possible, and imposes serious challenges on the root cause analysis of such effects.
- Measurements cannot be demonstrated to be safe (*i.e.*, is it really the worst case we encountered?).

— *Stack usage problems*

Stack overflow is also a safety issue. The absence of stack overflow is a property that must be demonstrated during target integration verification. However, the nature and complexity of the problem makes prediction and avoidance very hard to achieve and even harder to demonstrate. A common and traditional method for verifying stack usage is to write a short program which fills the stack with a given bit-pattern, and then execute the application and count how many stack registers still have the bit-pattern.

But how can you be sure that you really have the most pessimistic execution order and data usage in your application?

SCADE Suite includes two different modules that support timing and stack analysis of models:

- **Timing and Stack Optimizer** (TSO) computes the WCET and stack size estimation for a generic platform. TSO is usually used to compare different versions of a model to determine the most efficient design. SCADE Suite users can use it to monitor the performances of their design with respect to WCET and stack usage. This tool is relevant for early verification of the compatibility between the model and the target platform.
- **Timing and Stack Verifier** (TSV) computes precise WCET and stack size for a model on a specific hardware target. Such analysis runs with respect to specific target processor and C compiler, and requires fine-grained tool configuration to comply with the hardware characteristics. Even if TSV is still relevant during early verification of the target compatibility analysis, its operating mode is quite complex (due to the number of parameters to be set) and it is usually relevant only when precise WCET and stack size measurements are required during final integration testing on the target platform.

Timing and Stack Optimizer and Timing and Stack Verifier are fully integrated into the SCADE Suite environment. The analysis results are directly shown, and hyperlinks are available for direct reference to the model constructs matching each WCET and/or stack size results.

Figure 64 illustrates global visualization results.

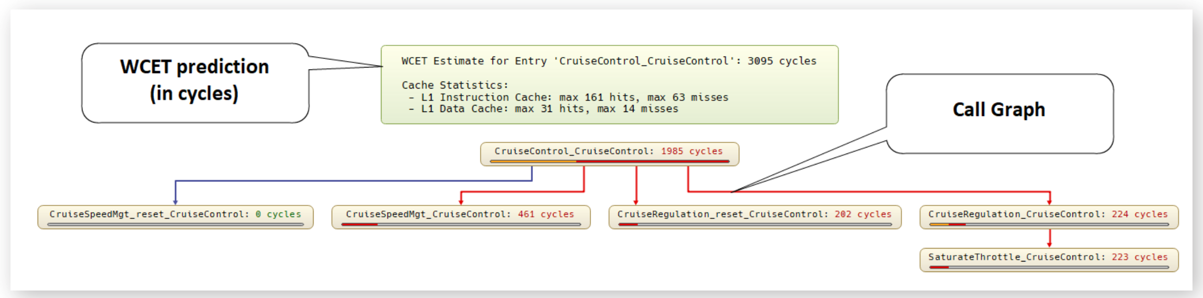
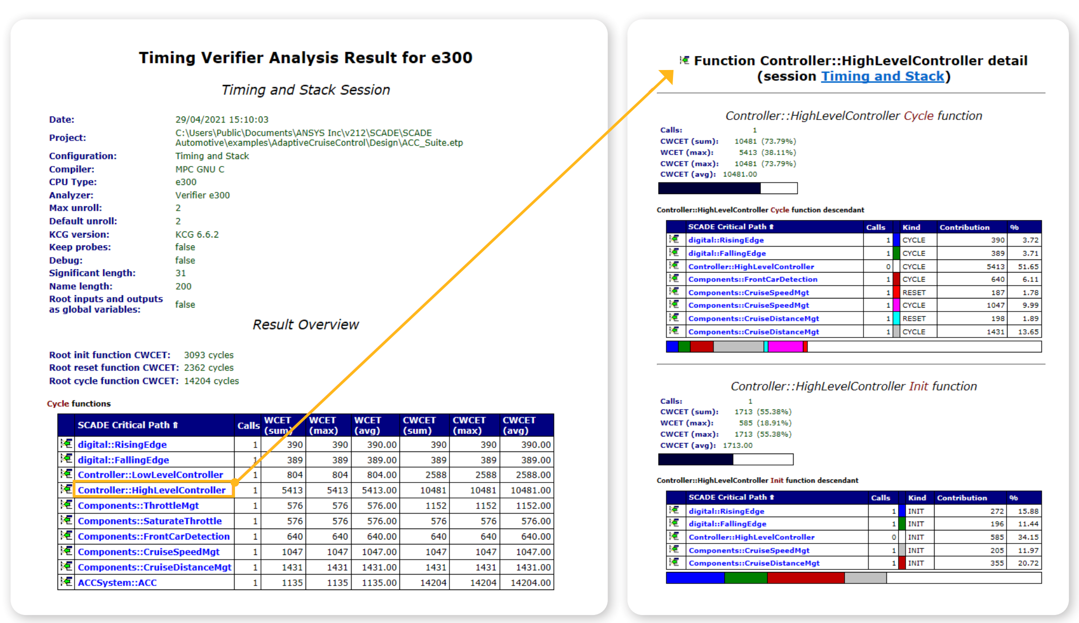


FIGURE 64: TIMING AND STACK ANALYSIS GLOBAL VISUALIZATION

Figure 65 illustrates global and detailed results of Timing Analysis.



Timing Verifier Analysis Result for e300

Timing and Stack Session

Date:	29/04/2021 15:10:03
Project:	C:\Users\Public\Documents\ANSYS Inc\v212\SCADE\SCADE Automotive\examples\AdaptiveCruiseControl\Design\ACC_Suite.etp
Configuration:	Timing and Stack
Compiler:	MPC GNU C
CPU Type:	e300
Analyzer:	Verifier e300
Max unroll:	2
Default unroll:	2
KCG version:	KCG 6.6.2
Keep probes:	false
Debug:	false
Significant length:	31
Name length:	200
Root inputs and outputs as global variables:	false

Result Overview

Root init function CWCET: 3093 cycles
Root reset function CWCET: 2362 cycles
Root cycle function CWCET: 14204 cycles

Cycle functions

SCADE Critical Path ⇑	Calls	WCET (sum)	WCET (max)	WCET (avg)	CWCET (sum)	CWCET (max)	CWCET (avg)
digital::RisingEdge	1	390	390	390.00	390	390	390.00
digital::FallingEdge	1	389	389	389.00	389	389	389.00
Controller::LowLevelController	1	804	804	804.00	2588	2588	2588.00
Controller::HighLevelController	1	5413	5413	5413.00	10481	10481	10481.00
Components::ThrottleMgt	1	576	576	576.00	1152	1152	1152.00
Components::SaturateThrottle	1	576	576	576.00	576	576	576.00
Components::FrontCarDetection	1	640	640	640.00	640	640	640.00
Components::CruiseSpeedMgt	1	1047	1047	1047.00	1047	1047	1047.00
Components::CruiseDistanceMgt	1	1431	1431	1431.00	1431	1431	1431.00
ACCSystem::ACC	1	1135	1135	1135.00	14204	14204	14204.00

Function Controller::HighLevelController detail (session Timing and Stack)

Controller::HighLevelController Cycle function

Calls: 1
CWCET (sum): 10481 (73.79%)
WCET (max): 5413 (38.11%)
CWCET (max): 10481 (73.79%)
CWCET (avg): 10481.00

Controller::HighLevelController Cycle function descendant

SCADE Critical Path ⇑	Calls	Kind	Contribution	%
digital::RisingEdge	1	CYCLE	390	3.72
digital::FallingEdge	1	CYCLE	389	3.71
Controller::HighLevelController	0	CYCLE	5413	51.65
Components::FrontCarDetection	1	CYCLE	640	6.11
Components::CruiseSpeedMgt	1	RESET	187	1.78
Components::CruiseSpeedMgt	1	CYCLE	1047	9.99
Components::CruiseDistanceMgt	1	RESET	198	1.89
Components::CruiseDistanceMgt	1	CYCLE	1431	13.65

Controller::HighLevelController Init function

Calls: 1
CWCET (sum): 1713 (55.38%)
WCET (max): 585 (18.91%)
CWCET (max): 1713 (55.38%)
CWCET (avg): 1713.00

Controller::HighLevelController Init function descendant

SCADE Critical Path ⇑	Calls	Kind	Contribution	%
digital::RisingEdge	1	INIT	272	15.88
digital::FallingEdge	1	INIT	196	11.44
Controller::HighLevelController	0	INIT	585	34.15
Components::CruiseSpeedMgt	1	INIT	205	11.97
Components::CruiseDistanceMgt	1	INIT	355	20.72

FIGURE 65: TIMING VERIFIER ANALYSIS REPORTS

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

COMPATIBILITY OF GENERATED CODE WITH TARGET PLATFORM

Moving further with the ISO 26262-6:2018 requirements regarding software models and components complexity (Clauses 5.4.3 and 7.4.3), we now need to address potential limitations that can be due to compilation of the generated source code for the target platform.

SCADE Suite includes a Compiler Verification Kit (CVK) with the objective of verifying that the type of code generated by SCADE Suite KCG is correctly compiled/executed with a given cross-compiler on the target platform. For example, a cross-compiler may have limitations in the level of imbrication of some constructs and these limitations will impose corresponding limitations to the complexity of the Scade models that should be allowed.

SCADE Suite CVK contributes to early verification of the correctness and consistency of the development environment with the development standards and the target platform.

CVK relies on a sample-based approach that is relevant due to the characteristics of generated code: regular patterns that strictly conform to restricted coding standards defined in [SCS-KCG-TOR] documentation. For further information related to CVK principles and CVK development strategy, refer to Appendix F.

8.3.4 Impact of SCADE Suite KCG code generator qualification

The SCADE Suite KCG Code Generator has been qualified for [ISO 26262:2018] at TCL3 tool confidence level (see Section 2.5 regarding Confidence in the Use of Software Tools). For further details on SCADE Suite KCG qualification, see Appendix E.1.

We will now consider the benefits of SCADE Suite KCG qualification, but also the application conditions that must be obeyed when KCG is used.

BENEFITS OF SCADE SUITE KCG QUALIFICATION

Qualification of SCADE Suite KCG provides the following benefits:

— *Source code complies with the software architectural design*

The qualification of SCADE Suite KCG ensures that the source code generated from any correct set of Scade models complies with the software architectural design.

The architecture of SCADE Suite KCG generated code is determined by the SCADE Suite users. The definition of the architecture includes the model structure, expansion directives, and interface directives as explained in Section 7.4.

Note: If the models are not correct, no code is generated.

— *Source code complies with the software units detailed design*

The qualification of SCADE Suite KCG ensures that the source code generated from any correct set of models reflects these models accurately and consistently. This evidence is based on the requirements of KCG [SCS-KCG-TOR] that include:

- The verification that the model complies with the syntactic/semantic rules of the input language
- A code generation scheme ensuring that the source code generated from any correct set of Scade models complies with the detailed algorithms specified in these models.

Note: If the models are not correct, no code is generated.

— ***Source code is verifiable***

The qualification of SCADE Suite KCG ensures that the code structures generated from any correct set of models have a clear meaning, reflecting elements of the models. No activity at code level is required.

— ***Source code conforms to coding standards***

The qualification of SCADE Suite KCG ensures that the source code generated from any correct set of models complies with its coding standards. Coding rules for SCADE Suite KCG are defined in SCADE Suite KCG Tool Operational Requirements (TOR) document [SCS-KCG-TOR].

As discussed earlier, KCG generates a small and safe subset of the C language. In addition, the generated code complies with MISRA C:2012, as defined in [MISRA C:2012] and [MISRA C:2012/AMD1]. Compliance to MISRA C:2012 and AMD1 is demonstrated in the compliance report [SCS-KCG-MISRA-C-COMPL].

— ***Source code is traceable to the software units design models***

The qualification of SCADE Suite KCG ensures that the source code generated from any correct set of models is traceable to the detailed design contained in these models.

APPLICATION CONDITIONS OF SCADE SUITE KCG

We now detail the application conditions for the use of SCADE Suite KCG as they are described in the KCG safety case [SCS-KCG-Safety Case] and as they must be followed to guarantee the above benefits.

The most significant SCADE Suite KCG application conditions that pertain to installation and use of KCG, as well as Scade modeling are described in Table 19 below. The application conditions that pertain to integration are described in Section 9.3.4 (Integration of external code, see Table 24).

Note: For the complete and formal description of the KCG application conditions, the reader must refer to [SCS-KCG-Safety Case].

The categories, which are given to help users understand at which stage of development these conditions should be applied, are explained below:

- **Tool installation and use:** This category concerns installing KCG in the user development environment and checking integrity of the installation as well as applying measures communicated by the tool developer.
- **Scade modeling:** This concerns the development of the Scade model itself.
- **Integration:** This concerns the development of external objects (imported types or operators) and their integration with the Scade generated code, the process of integrating the Scade generated code in the hand-coded user application parts, using KCG generated code C API, producing the SCADE part of the Executable Object Code (EOC).

Some application conditions reference MISRA Guidelines. MISRA guidelines are classified as “Directive” when an exhaustive description is not possible, or “Rule” when complete description is possible. They are referenced as MISRA-Dn.m or MISRA-Rn.m accordingly, where n.m are the rule numbers as in the MISRA standard.

TABLE 18: SCADE SUITE KCG APPLICATION CONDITIONS (INSTALLATION, USE, AND SCADE MODELING)

Source: Extract from Table 9 in [SCS-KCG-Safety Case]

Category	Id	Application condition
Installation and Use	USR-001	The user shall check the integrity of the tool installation. See the installation procedure [SCS-KCG-SIP] for performing that verification. If several versions of KCG need to be installed, then the user should take care of launching the correct version.
	USR-002	The user shall ensure integrity of the platform used for execution of KCG (correct hardware, correct OS installation, protection against unauthorized access, detection of random hardware failures). To verify integrity of KCG execution, use a robust host platform or run KCG twice and compare results. Any deviation from the original qualification environment may cause KCG to work improperly.
	USR-006	The user shall analyze the KCG log file to verify that it explicitly reports zero error/warning or otherwise, analyze the reported errors/warnings.
Scade modeling	USR-008	The Scade part of the application software shall be developed in compliance with the requirements/objectives for its software integrity level as defined by the applicable safety standard (*e.g.*, review, configuration management). This includes integrity requirements for the model files and for the items generated by KCG.
	USR-022	The user shall ensure that in the Scade model, elements that are directly propagated to the generated source comply with the target language standard. These constructs are literals and pragma doc text, ... This is necessary condition for MISRA-D1.1.
	USR-028	The user shall not use the unary minus in the model on an unsigned type. This is necessary condition for MISRA-R10.1.
	USR-029	The user shall not use the equality and/or inequality operator on floating type data in Scade models. Instead, the difference of two floating-point values shall be compared against a user-defined threshold. This is necessary condition for MISRA-D1.1.
	USR-039	The user shall not use the underlying bit representations of floating-point values in the model and/or manual code. This is necessary condition for MISRA-D1.1.
	USR-031	If a function in the model or in the directly called imported code returns error information, then that error information shall be tested in the model. Covers MISRA-D4.7.
	USR-032	The user shall not use identifiers reserved for C standard libraries in the input Scade model (*e.g.*, malloc, exit, see sections 7.1.3 and 7.13 of [ISO-IEC-9899]). This includes identifiers generated by effect of naming prefix and/or significance length options. Covers MISRA-R21.1 and MISRA-R21.2.
	USR-037	The user shall ensure that the model arithmetics respect the definition domain given in [SCS-KCG-LRM] and [SCS-KCG-TOR] or is defined by the user specific development toolchain (target language and compiler) for parts that are implementation defined. This is necessary condition for MISRA-D1.1 and MISRA-R1.3. Violating this rule may lead to unexpected runtime behavior such as overflow, division by zero.
	USR-038	The user should take care of the risk of floating-point absorption when ordering computations.

8.4 Coverage Analysis with SCADE Test Model Coverage

According to Table 17 above, structural coverage analysis is required to verify that every element of a software unit was fully exercised when requirements-based tests are performed. The objective of this activity is to verify that the software units are fully covered by test cases.

In the context of SCADE model-based development, SCADE Suite is used to represent the software units design and, according to NOTE 3 of Table 17, model coverage analysis is used as a means of assessing how far the behavior of a design model was explored.

Model coverage analysis focuses on the functional origin of coverage holes, whether they are due to

- lack of testing
- inadequate software requirements
- dead, deactivated, or unintended functionality

It complements the software requirements to design traceability analysis.

SCADE Test Model Coverage takes as inputs a SCADE Suite model and a set of requirements-based test cases and procedures and it generates a model coverage report and evidence for model structural coverage.

The positioning of SCADE Test Model Coverage within the software development and verification flow is shown below.

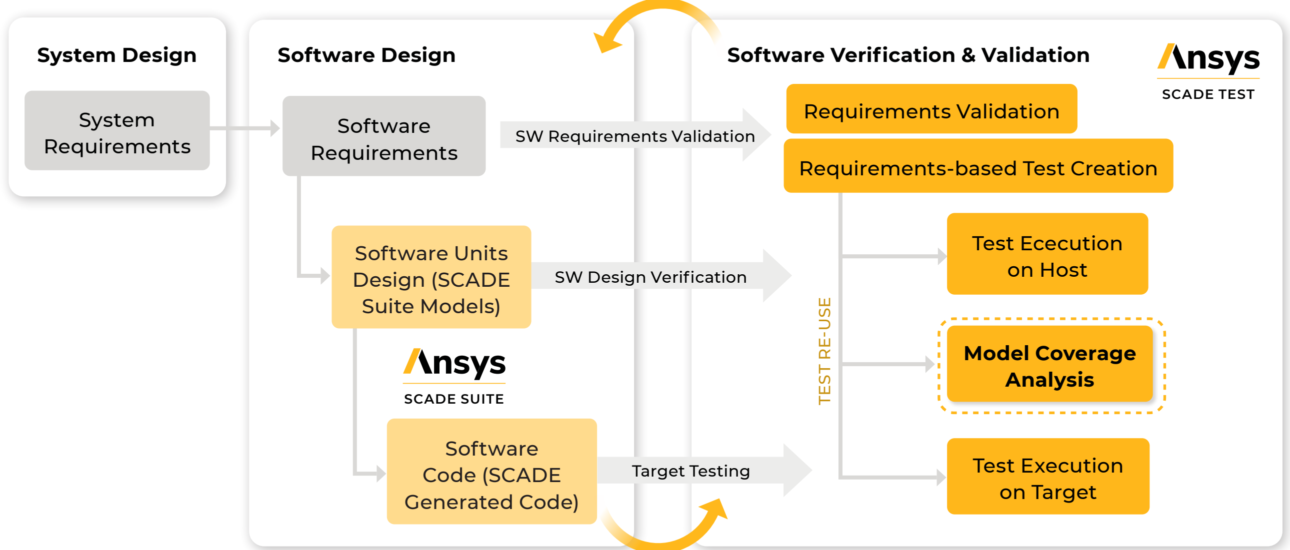


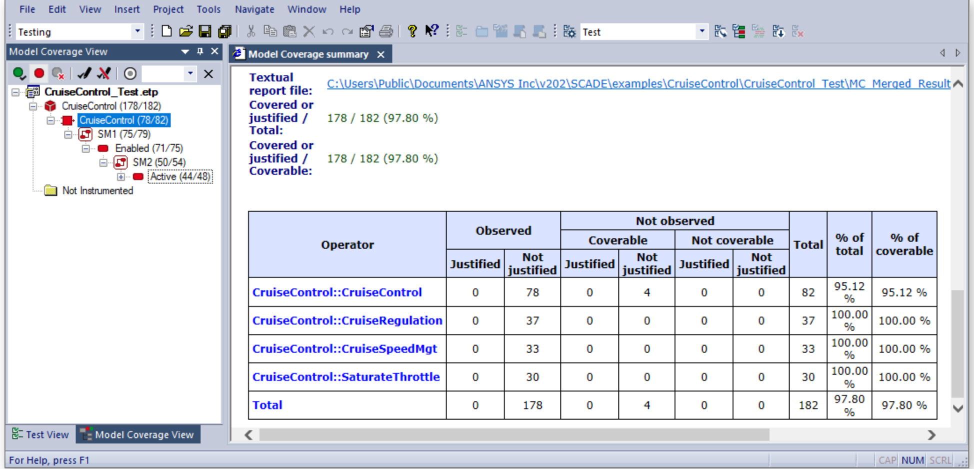
FIGURE 66: POSITIONING OF SCADE TEST MODEL COVERAGE WITHIN THE VERIFICATION FLOW

8.4.1 Using SCADE Test Model Coverage

The activities performed by a user of SCADE Test Model Coverage are:

1. **Model Coverage Acquisition:** Running test cases with the SCADE Test Environment for Host module, while measuring the coverage of each operator.
2. **Model Coverage Analysis:** Identifying the operators that are not fully covered.
3. **Model Coverage Resolution:** Adding test cases or providing the explanation or the necessary fixes for each operator that is not fully covered. Fixes can be in the software requirements, in the model, or both.

The use of SCADE Test Model Coverage is illustrated in Figure 67 . The coverage result for each operator and child elements is indicated via colors and coverage ratios about observed coverage points. The tool provides also detailed explanations about operator features that are not fully covered.



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

FIGURE 67: MODEL COVERAGE ANALYSIS AND RESOLUTION WITH SCADE MODEL TEST COVERAGE

The above report is qualified for ISO 26262:2018 (see Appendix E.5). In addition, there is SCADE Test HMI support that provides good visibility into the coverage holes.

Model coverage holes may reveal the following deficiencies:

1. **Shortcomings in software requirements-based test cases and/or procedures:** in that case, resolution consists in adding missing requirements-based test cases and/or procedures.
2. **Inadequacies or shortcomings in the software requirements:** in that case, resolution consists in fixing 1) the software requirements and 2) the design model, and assessing the effects and needs for reverification.
3. **Previously unidentified requirements:** in that case, resolution consists in adding the missing software requirements and assessing the effects and needs for reverification.
4. **Unintended functionality in model:** in that case, resolution consists in removing dead model parts, if appropriate, on or justifying their presence and safety as they may correspond to functionality activated upon a specific configuration (*e.g.,* vehicle dependent). In both cases the effects and needs for reverification must be assessed.

EXAMPLE 1: INSUFFICIENT TESTING

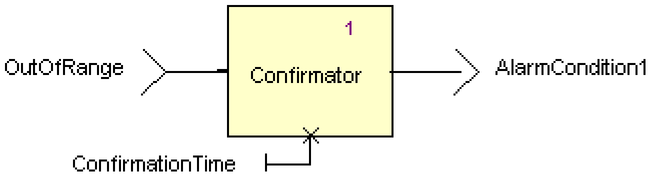


FIGURE 68: A CONFIRMATOR

- **Analysis:** Assume that Model Coverage detects that AlarmConditon1 in Figure 68 was not raised during testing activities and that the analysis concludes that the requirement is correct, but testing is not sufficient.
- **Resolution:** Develop additional tests.

EXAMPLE 2: LACK OF ACCURACY IN THE SOFTWARE REQUIREMENT

Assume that Model Coverage detects that the Integrator in Figure 69 was never reset (R) during the tests. Is the “reset” behavior an unintended function?

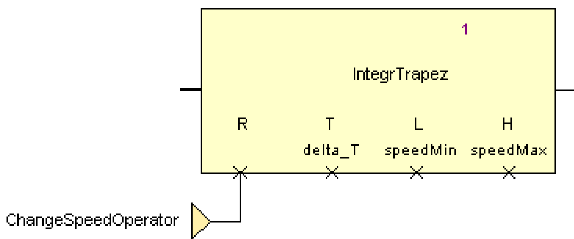


FIGURE 69: AN INTEGRATOR

- **Analysis:** If resetting the Integrator is an intended function, but the software requirement did not specify that changing the speed regulation mode implies resetting all filters, no test case was exercised for this situation.
- **Resolution:** Complement the software requirement and add a test case.

8.4.2 Model coverage criteria

The model coverage criteria of SCADE Test Model Coverage were designed to satisfy the following objectives:

- Match ISO 26262-6:2018 model coverage principles
- Fit the entire Scade language: data flow constructs as well as logic-oriented constructs (state machines, clocked blocks)
- Provide a sound and accurate assessment of the fact that every model construct and flow are exercised by Model-in-the-Loop testing

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

- Models describe the software functionality, while C programs describe its implementation. It creates a major difference in terms of abstraction level (feature coverage in SCADE *vs.* code coverage in C) and of coverage of multiple instances (each instance of a Scade operator is analyzed for coverage).
- Models are based on functional data flows and state machines, while most programming languages and their criteria are sequential.

For Scade models, we use tags to represent coverage points, as show in Figure 70 . Model coverage criteria are based on tag propagation and observation through observable outputs of the model. Setting coverage criteria amounts to defining where tags are introduced in the model and what is the semantic of tag propagation to be used for Boolean primitives. For criteria that distinguish Boolean flows (see ODC and OMC/DC in the text below), two tags are introduced by the “bool_tag” primitive: one when the flow takes value true and the other when it is false. Each tag introduced in the model is expected to reach an observation point (red circle on output in Figure 71). **A point is covered if the model is stimulated by an input sequence leading to the observation of the corresponding tag.** The overall coverage measure is the ratio of observed tags to introduced tags.

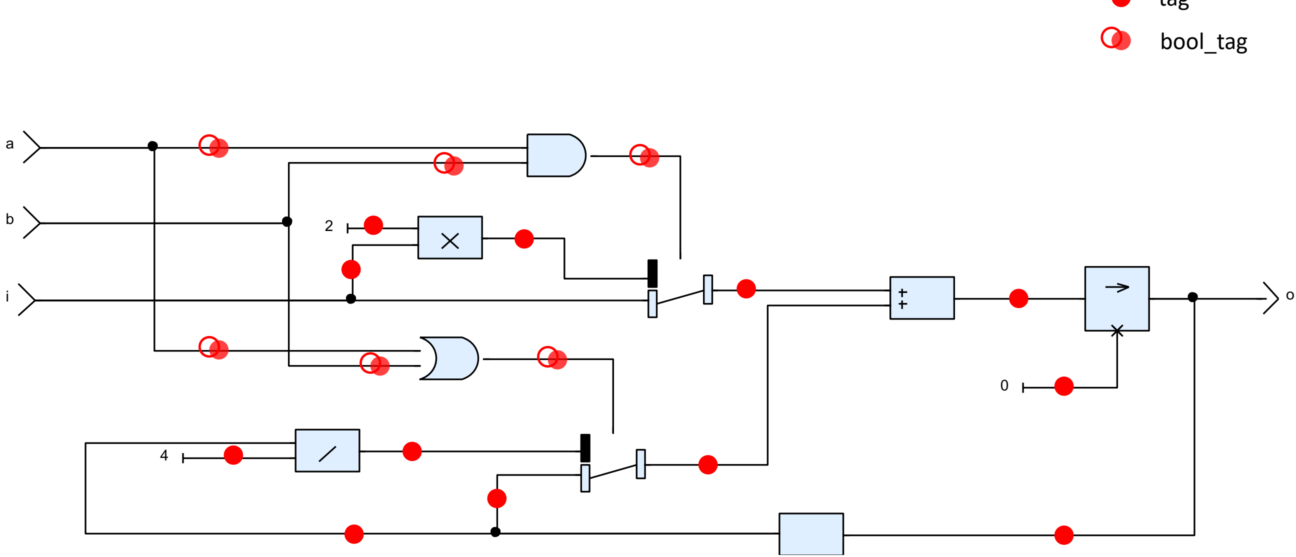


FIGURE 70: TAG PROPAGATION AND OUTPUT OBSERVATION FOR SCADE SUITE MODEL COVERAGE

The model coverage criteria for Scade models are:

1 INFLUENCE

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

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

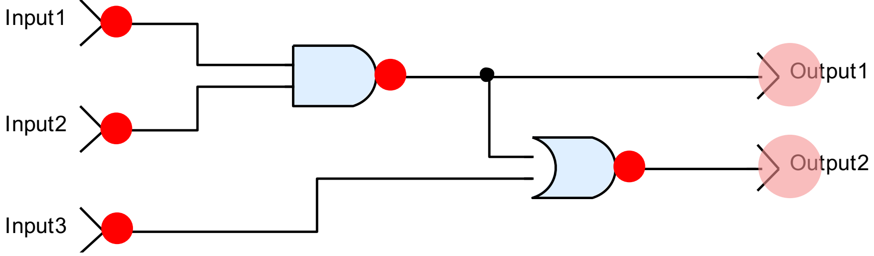


FIGURE 71: TAGS AND OBSERVATION FOR INFLUENCE

2 OBSERVABLE DECISION COVERAGE (ODC)

This criterion measures coverage based on tags that can distinguish between the influence of True and the influence of False for the monitoring of Boolean flows. With this criterion, the propagation rules for Boolean primitives are the same as for Influence. The semantics of tag propagation of this criteria ignores the MC/DC masking effect[10] of Boolean flows on coverage measurements.

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

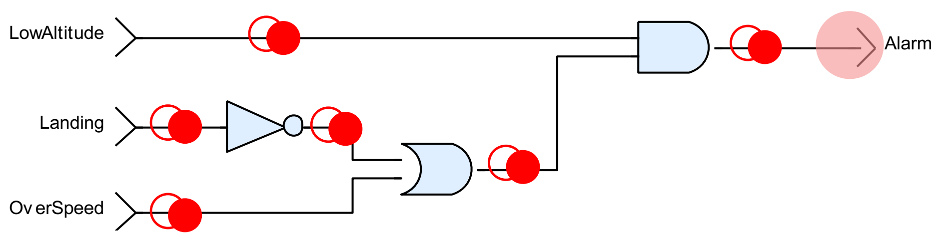


FIGURE 72: TAGS AND OBSERVATION FOR ODC

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

This criterion measures coverage based on the same tags as ODC (see Figure above) and a semantics of tag propagation that considers the masking effect over coverage measurements (see Note above).

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

Table 19 summarizes all coverage criteria used by SCADE Test Model Coverage.

TABLE 19: COVERAGE CRITERIA IN SCADE TEST MODEL COVERAGE FOR SCADE MODELS

Coverage Criterion	Applies to	Synopsys
Influence	Any flow type	All connection points were tested as able to influence an output.
Observable Decision Coverage (ODC)	Boolean expressions	All connection points were tested as able to influence an output and all Boolean flows have taken both True/False values while influencing an output without considering the masking effect of Boolean operators.
Observable Modified Condition/Decision Coverage (OMC/DC)	Boolean expressions	All connection points were tested as able to influence an output, and all Boolean flows have taken both True/False values while influencing an output while considering the masking effect of Boolean operators.

According to NOTE 4 of Table 17, “the analysis of structural coverage performed at the model level can replace the source code coverage metrics if it is shown to be equivalent, with rationales based on evidence that the coverage is representative of the code level”.

The coverage criteria of SCADE Test Model Coverage (OMC/DC, ODC, Influence) are defined as a correspondence to code coverage criteria (MC/DC, Branch Coverage, Statement Coverage) in such a way that, when model coverage is achieved for a matching criterion, say OMC/DC, then structural coverage of SCADE Suite KCG-generated code holds for the corresponding criterion, say MC/DC.

[10] *Take as an example “A = (B and C) or D”. When considering the masking effect, test cases where D is True cannot be considered to determine if the “and” has been implemented correctly. For more details, see [NASA-MCDC].*

This is shown in Table 21 below and detailed in [MCOV-FAQ11] and [MCOV-FAQ11-Ext].

TABLE 20: MODEL TO CODE LEVEL COVERAGE IMPLICATION

Model-level Coverage Criterion	Code-level Coverage Criterion
OMC/DC	MC/DC
ODC	Branch Coverage
Influence	Statement Coverage

SCADE Test Model Coverage has been qualified for [ISO 26262:2018] at TCL3. For further details on SCADE Test Model Coverage qualification, see Appendix E.5.

4 ADDITIONAL USER CRITERIA

It is also possible to add coverage points to the structural ones that have been defined above.

These new points can be used to support the testing activity by adding coverage objectives relative to this activity, and not only to the structure of the software. The present section gives an example where additional criteria are introduced to support an **equivalence class testing** approach.

Adding nonstructural coverage points is done by introducing coverage observers that define the specific coverage objectives. A coverage observer is a standard Scade operator. It introduces additional coverage points by using the Model Coverage primitives provided in a library. These observers are separated from the design project; their design is part of the coverage analysis activity. This allows to add functional or equivalence class criteria.

Let us consider a limiter used to limit CruiseSpeed within the [SpeedMin, SpeedMax] range in the ACC example.

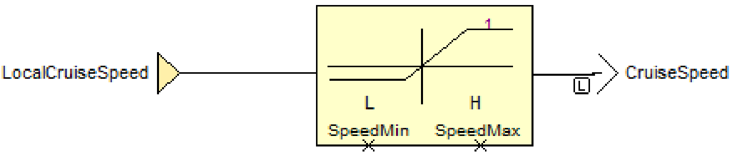


FIGURE 73: LIMITER OPERATOR USED TO LIMIT CRUISESPEED

When testing integration of this operator, ISO 26262:2018 recommends using equivalence testing to perform integration testing.

Let us suppose the testing activity leads to the identification of the following equivalence classes for LocalCruiseSpeed:

- [-infinite, SpeedMin] corresponding to “lower” equivalence class
- [SpeedMin, SpeedMax] corresponding to “in_between” equivalence class
- [SpeedMax, +infinite] corresponding to “higher” equivalence class
- [SpeedMin, SpeedMin + epsilon] corresponding to “near_low” equivalence class
- [SpeedMax – epsilon, SpeedMax] corresponding to “near-high” equivalence class

It can be represented in the Figure below:

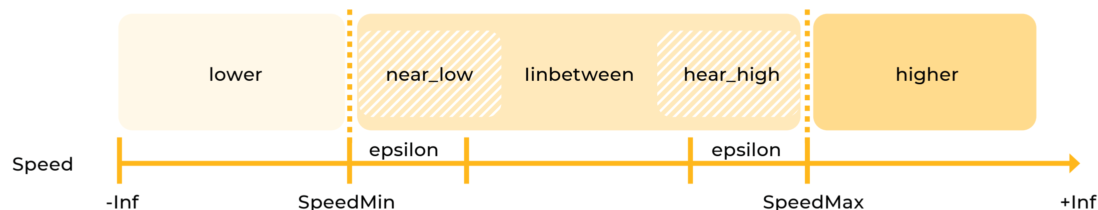


FIGURE 74: EQUIVALENCE CLASSES FOR LIMITER

To add the coverage criteria corresponding to the equivalence classes, an observer is defined using the Scade language, as illustrated below.

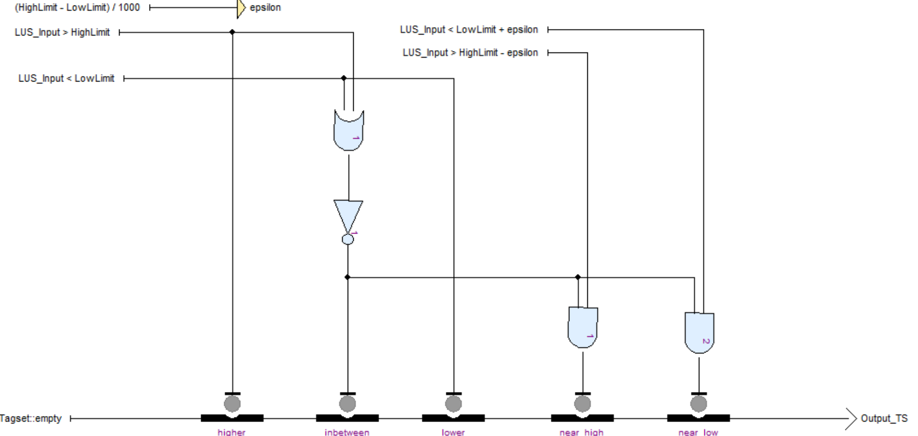


FIGURE 75: LIMITER OBSERVER DEFINING EQUIVALENCE CLASSES CRITERIA

With this observer added to Model Coverage settings, we get the equivalence class coverage points in the qualified report provided as shown below.

```
--------------------------------------------------------------------------------
Additional points:
--------------------------------------------------------------------------------
NOT OBSERVED     ADDITIONAL INFLUENCE (pwlinear::LimiterUnSymmetrical#1)>LimiterUnSymmetrical_OBS/_L10=
                 "higher"
OBSERVED         ADDITIONAL INFLUENCE (pwlinear::LimiterUnSymmetrical#1)>LimiterUnSymmetrical_OBS/_L11=
                 "inbetween"
NOT OBSERVED     ADDITIONAL INFLUENCE (pwlinear::LimiterUnSymmetrical#1)>LimiterUnSymmetrical_OBS/_L12=
                 "lower"
OBSERVED         ADDITIONAL INFLUENCE (pwlinear::LimiterUnSymmetrical#1)>LimiterUnSymmetrical_OBS/_L13=
                 "near_high"
OBSERVED         ADDITIONAL INFLUENCE (pwlinear::LimiterUnSymmetrical#1)>LimiterUnSymmetrical_OBS/_L14=
                 "near_low"
--------------------------------------------------------------------------------
```

FIGURE 76: COVERAGE REPORT INCLUDING EQUIVALENCE CLASSES

8.5 Takeaway from Using SCADE Suite, SCADE Test, and SCADE LifeCycle for Software Unit Verification

This Chapter has established how SCADE Suite, SCADE Test, and SCADE LifeCycle fulfill the requirements and recommendations of [ISO 26262-6] regarding verification of the software units.

This can be seen in four ways, depending on the tool that is used:

1. SCADE Suite
   - checks model static properties (e.g., correct typing and initializations)
   - supports efficient design model reviews
   - checks for numerical robustness of algorithms (e.g., division by zero)
   - supports formal verification of model functional properties
   - generates MISRA compliant source code from models, with a qualified code generator
   - supports the integration of multiple software units designed in Scade
   - allows to evaluate performance of the generated code
2. SCADE Test Environment for Host
   - supports efficient creation of requirements-based test cases and running them on host with a qualified testing tool
   - eliminates the need for back-to-back comparison tests between model and code for reason that the code generator is qualified, and that the code behaves the same way as the model
3. SCADE Test Model Coverage
   - performs structural coverage analysis at model level
   - enable creating additional criteria for equivalence classes
   - guarantees that coverage at model level implies code coverage at the proper level (statement coverage, branch coverage, and MC/DC), when SCADE Suite KCG is used to generate the source code
4. SCADE LifeCycle
   - supports model reviews (Reporter)
   - supports incremental model reviews (Model Change)

A detailed analysis of the level of support of SCADE Suite, SCADE Test Environment for Host, SCADE Test Model Coverage, and SCADE LifeCycle for software unit verification is provided in Appendix C.5.

9

SOFTWARE INTEGRATION AND VERIFICATION

9.1 Objectives and Work Products

The objectives of this sub-phase (Clause 10 of [ISO 26262-6:2018]) are to:

- define the integration steps and integrate the software elements until the embedded software is fully integrated
- verify that the defined safety measures [...] at the software architectural level are properly implemented
- provide the evidence that the integrated software units and software components fulfil their requirements according to the software architectural design
- provide sufficient evidence that the integrated software contains neither undesired functionalities nor undesired properties regarding functional safety

The inputs to the software integration and verification sub-phase are:

- software architectural specification
- hardware-software interfaces specification
- software requirements specification
- configuration data and calibration data, if any
- software units design specification
- software units implementation
- software verification specification
- software verification report

Work products are:

- software verification specification (refined)
- software verification report (refined)
- embedded software

9.2 Requirements and Recommendations

According to Section 10.4.2 of [ISO 26262-6], the software integration shall be verified to provide evidence of:

- compliance to software architectural design
- compliance with hardware-software interface specification
- achievement of the specified functionality and properties
- sufficient resources to support the functionality
- effectiveness of the safety measures resulting from the safety-oriented analyses

The following Tables describe methods that can be used to achieve software integration and verification the above requirements.

TABLE 21: METHODS FOR VERIFICATION OF SOFTWARE INTEGRATION

Source: Table 10 in ISO 26262-6:2018

Methods		ASIL			
		A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	++	++
1d	Resource usage evaluation[c, d]	++	++	++	++
1e	Back-to-back comparison test between model and code, if applicable[e]	+	+	++	++
1f	Verification of the control flow and data flow	+	+	++	++
1g	Static code analysis[f]	++	++	++	++
1h	Static analyses based on abstract interpretation[g]	+	+	+	+

[a] *The software requirements allocated to the architectural elements are the basis for this requirements-based test.*

[b] *In the context of software integration testing, fault injection test means to introduce faults into the software for the purposes described in 10.4.3 and in particular to test the correctness of hardware-software interface related to safety mechanisms. This includes injection of arbitrary faults in order to test safety mechanisms (e.g. by corrupting software interfaces). Fault injection can also be used to verify freedom from interference.*

[c] *To ensure the fulfilment of requirements influenced by the hardware architectural design with sufficient tolerance, properties such as average and maximum processor performance, minimum or maximum execution times, storage usage (e.g. RAM for stack and heap, ROM for program and data) and the bandwidth of communication links (e.g. data buses) have to be determined.*

[d] *Some aspects of the resource usage evaluation can only be performed properly when the software integration tests are executed on the target environment or if the emulator for the target processor adequately supports resource usage tests.*

[e] *This method requires a model that can simulate the functionality of the software components. Here, the model and code are stimulated in the same way and results compared with each other.*

[f] *Static analyses are a collective term which includes analysis such as architectural analyses, analyses of resource consumption and searching the source code text or the model for patterns matching known faults or compliance with modelling or coding guidelines, if not already verified at the unit level.*

[g] *Static analyses based on abstract interpretation are a collective term for extended static analysis which also includes analysis such as extending the compiler parse tree by adding semantic information which can be checked against violation of defined rules (e.g. data-type problems, uninitialized variables), control-flow graph generation and data-flow analysis (e.g. to capture faults related to race conditions and deadlocks, pointer misuses) or even meta compilation and abstract code or model interpretation, if not already verified at the unit level.*

NOTE 2: For model-based development, the verification objects can be the models associated with the software components.

TABLE 22: METHODS FOR DERIVING TEST CASES FOR SOFTWARE INTEGRATION TESTING

Source: Table 11 in ISO 26262-6:2018

Methods		ASIL			
		A	B	C	D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes[a]	+	++	++	++
1c	Analysis of boundary values[b]	+	++	++	++
1d	Error guessing based on knowledge or experience[c]	+	+	+	+

[a] *Equivalence classes can be identified based on the division of inputs and outputs, such that a representative test value can be selected for each class.*

[b] *This method applies to parameters' or variables' values approaching and crossing the boundaries and out of range values.*

[c] *Error guessing tests can be based on data collected through a "lessons learned" process and expert judgment.*

TABLE 23: STRUCTURAL COVERAGE AT THE SOFTWARE ARCHITECTURE LEVEL

Source: Table 12 in ISO 26262-6:2018

Methods		ASIL			
		A	B	C	D
1a	Function coverage[a]	+	+	++	++
1b	Call coverage[b]	+	+	++	++

[a] *Method 1a refers to the percentage of executed software sub-programs or functions in the software (for definition see IEC 61508-7:2010, C.5.8).*

[b] *Method 1b refers to the percentage of executed software sub-programs or function with respect to each implemented call of these sub-programs or functions in the software.*

NOTE 2: In the case of model-based development, software integration testing can be performed at the model level using analogous structural coverage metrics for models.

9.3 Software Integration with SCADE Suite

9.3.1 Integration aspects of a SCADE application

The integration of a SCADE application is about:

- interface with the external environment (Inputs/Outputs)
- SCADE Suite module integration
- integration of external data and code
- scheduling and tasking

9.3.2 Interface with the external environment

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

9.3.3 SCADE Suite module integration

A module refers here to the C code generated by SCADE Suite KCG from a SCADE Suite component. Depending on the selected code generation process (see preferred first alternative of Figure 55 in Section 7.4), the user must manage the integration of one or several modules with the rest of the software application.

The SCADE Suite KCG directives for tuning the generated code (such as options and pragmas defined in Section 7.4) shall be considered by the user as early as possible while integrating the generated code.

Moreover, module integration depends on the implementation of predefined Scade types (see Section 3.2.1) which must be mapped to C types. A default type definition is given in the generated code, but it is possible to redefine these default types by providing the implementation of each basic type in a user configuration file.

9.3.4 Integration of external code

SCADE Suite allows to reference external code in models.

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

Coming back to the application conditions of SCADE Suite KCG that were introduced in Section 8.3.4, we now detail the application conditions that pertain to integration of external code in the Table below, including the integration of external objects into Scade generated code, and the integration of the Scade generated code in the hand-coded parts of the user application.

Note: For the complete and formal description of the KCG application conditions, the reader must refer to [SCS-KCG-Safety Case].

TABLE 24: SCADE SUITE KCG APPLICATION CONDITIONS (INTEGRATION)

Source: Extract from Table 9 in [SCS-KCG-Safety Case]

Category	Id	Application condition
Integration	USR-041	The user shall ensure conformance of the external application code to the synchronous principle and preservation of integrity of the generated code. See [SCS-KCG-TOR] section 18 for full reference.
Integration	USR-042	External code shall conform to integration rules regarding its memory and its expected Application Programming Interface (API), as specified in [SCS-KCG-TOR] Section 18. Covers MISRA-D4.14.
Integration	USR-011	The user shall ensure that - External Code is developed and verified as other manual code, in compliance with the applicable standard and project plans for the targeted software level, including MISRA if applicable for C target language. - Imported numeric constant expressions are correctly evaluated by the compiler. External code may cause failures if it violates interfacing rules and/or causes side effects. Note that compliance of external C code includes assurance that its worst-case execution time (WCET) is bounded and predictable.
Integration	USR-027	The user shall check that imported objects are defined only once. An identifier with an external linkage shall have exactly one external definition. This is necessary condition for MISRA-R5.1, MISRA-R5.2, MISRA-R5.4, MISRA-R5.5, MISRA-R5.6
Integration	USR-013	The user shall check that the target for the application is compatible with the application design. This includes but is not limited to numeric types size/accuracy, available memory and stack and worst-case execution time.
Integration	USR-014	KCG generated code should not be changed. The user is responsible for any change to the C or Ada source code generated from KCG.
Integration	USR-023	The user shall use identifiers that comply with the compiler/linker limitation about symbols (for example length limitations, case sensitivity). For the C language target, the user shall use the "-significance_length" option that fits the compiler/linker requirements. This is necessary condition for MISRA-R5.1, MISRA-R5.2, MISRA-R5.4, MISRA-R5.5, MISRA-R5.6. Note: The [ISO-IEC-9899] standard requires that compilers use a significance length of at least 31.
Integration	USR-026	The user shall analyze the documentation and behavior of the cross compiler for compliance to [ISO-IEC-9899] with respect to integer division for the C target language. This is necessary condition for MISRA-D1.1.
Integration	USR-033	In addition to runtime errors prevention measures, user shall define a strategy to handle potential remaining runtime errors. For the generated code, these are arithmetic run-time errors. For the imported code, this also includes any other type of error, *e.g.*, pointer/memory errors. This is necessary condition for MISRA-D4.1.

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

9.3.5 Scheduling and tasking

Scheduling must be addressed in the preliminary design phase, but for the sake of simplicity it is described below. First, the section recalls the execution semantics of Scade models, and then examines how to implement scheduling of a model in single or multirate mode, while in single tasking or multitasking mode.

SCADE SUITE EXECUTION SEMANTICS

The SCADE Suite execution semantics is based on a cycle-based execution model as described in Section 3.2.2. This model can be represented with Figure 77.

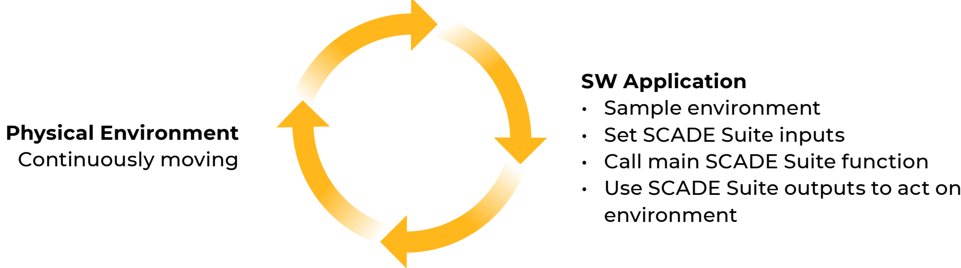


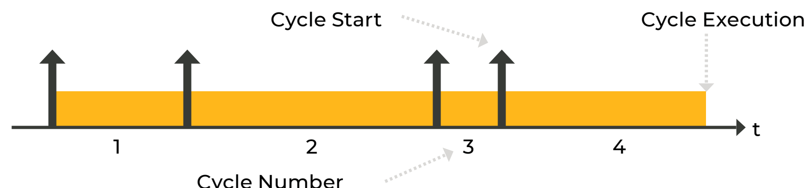
FIGURE 77: EXECUTION SEMANTICS OF SCADE SUITE

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

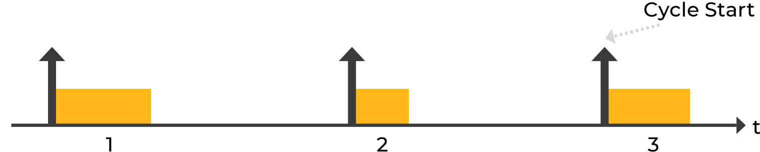
BARE SYSTEM IMPLEMENTATION

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

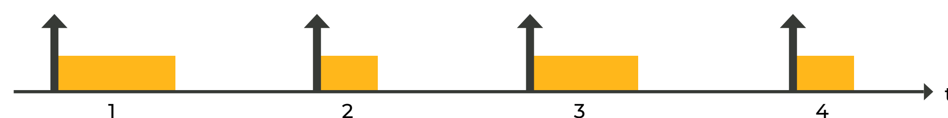
- **Polling:** a new cycle is started immediately after the end of the previous one in an infinite loop.



- **Event triggered:** a new cycle is started when a new start event occurs.



- **Time triggered:** a new cycle is started regularly, based on a clock signal.



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

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

SINGLE-TASK INTEGRATION OF SCADE SUITE FUNCTION WITH AN RTOS

A SCADE Suite design can be easily integrated in an RTOS task in the same way that it is integrated in a general-purpose code, as shown in Figure 78. The infinite loop construct is replaced by a task. This task is activated by the start event of the design, which can be a periodic alarm or a user activation.

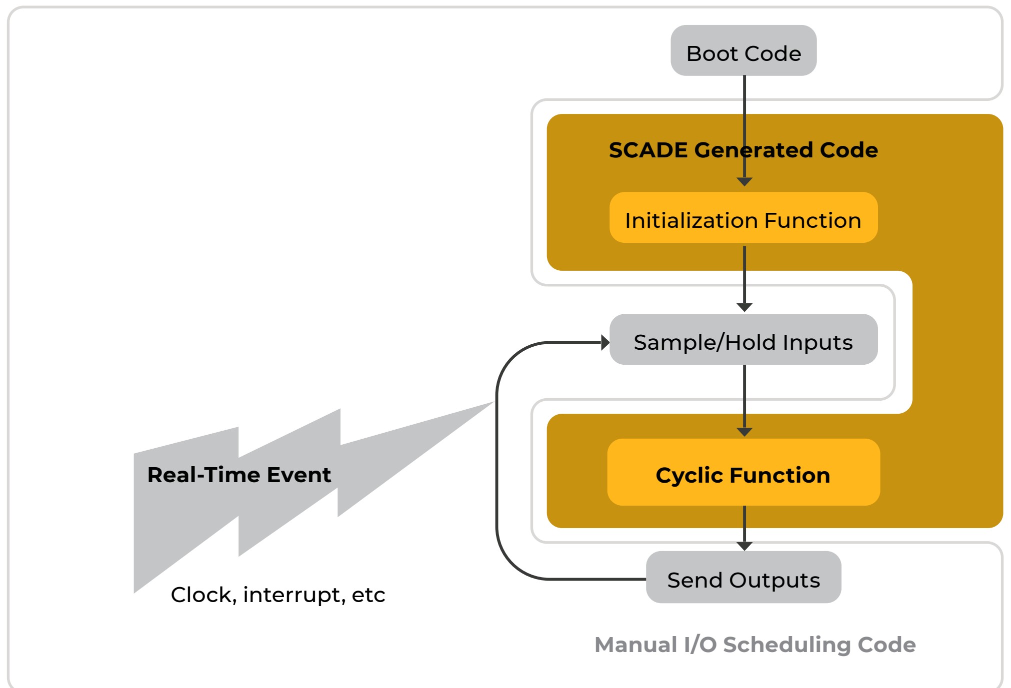


FIGURE 78: SCADE SUITE CODE INTEGRATION

This architecture can be designed by hand for any RTOS.

SCADE Suite provides automation of this code production through the SCADE Code Integration Toolbox allowing to develop user-specific adaptors for QNX™ from BlackBerry, VxWorks® 653 from Wind River®, for Integrity® from Green Hills® Software, for PikeOS from SYSGO, which have all been certified for ISO 26262:2018 at ASIL D, and for many platforms at major suppliers and integrators. The specific integration for AUTOSAR RTE is described in Section 9.3.6.

Note that concurrency is expressed **functionally** in Scade models and that SCADE Suite KCG considers the model structure to generate sequential code, considering this functional concurrency and the data flow dependencies. There is no need for the user to spend time sequencing parallel

flows, neither during modeling nor during implementation. There is no need to develop multiple tasks with complex and error-prone synchronization mechanisms. Note that other code, such as hardware drivers, may run in separate tasks, provided they do not interfere with the SCADE Suite generated code.

MULTIRATE, SINGLE-TASK APPLICATIONS

SCADE Suite can be used to design multirate applications in a single RTOS task. Some parts of the design can be executed at a slower rate than the top-level loop. Putting a slow part inside an *activate*[11] operator can do this. Slowest rates are derived from the fastest rate, which is always the top-level rate. This ensures a deterministic behavior.

The following application has two rates: Sys1 (as fast as the top-level) and Sys2 (four times slower), as shown in Figure 79.

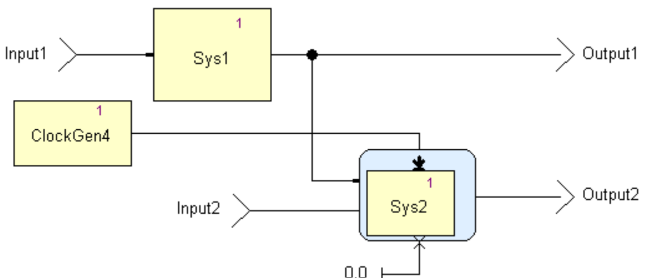


FIGURE 79: MODELING A BI-RATE SYSTEM

The schedule of this application is as shown in Figure 80 below:

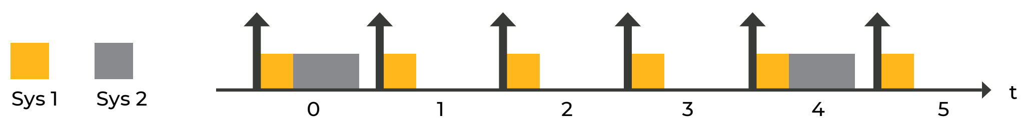


FIGURE 80: TIMING DIAGRAM OF A BI-RATE SYSTEM

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

The solution consists in splitting the slow into several smaller slow parts and distributing their execution on several fast rates. This is a simple way to design a multirate application. Scheduling of this application is fully deterministic and can be statically defined.

[11] *The Boolean activate operator, the blue rectangle of Figure 79, has an input condition (on top) used to trigger the execution of the computation that is described inside the block, thus allowing the introduction of various rates of execution for different parts of a model. The operator execution only occurs when a given activation condition is true.*

The previous application example can be redesigned as shown in Figure 81:

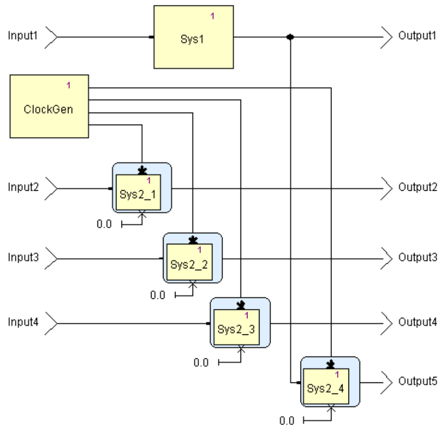


FIGURE 81: MODELING SLOW SYSTEM OVER FOURS CYCLES

The slow part, Sys2, is split into four subsystems. These subsystems are executed sequentially, one after the other, in four cycles, as shown in Figure 82 below:

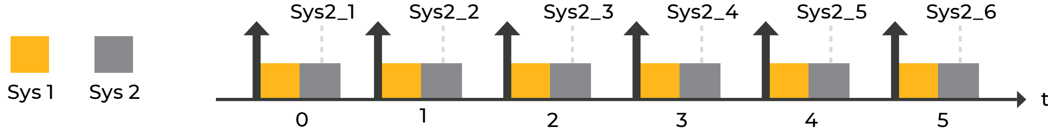


FIGURE 82: TIMING DIAGRAM OF DISTRIBUTED COMPUTATIONS

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

Such design has advantages but also constraints:

- **Advantages:**
  - Static scheduling: fully deterministic, no time slot exceeded or crushed, no RTOS deadlock
  - Data exchanges between subsystems handled by SCADE Suite wrt. dataflow execution order
  - SCADE Model-in-the-Loop testing and formal verification can be performed
  - Same code interface as a monorate application
- **Constraints**:
  - Need to know the WCET (Worst Case Execution Time) of each subsystem to validate scheduling in all cases
  - Split of slow subsystems can be difficult with high-rate ratio (e.g., 5ms and 500ms)
  - Constraint for design evolutions and maintenance

MULTITASKING IMPLEMENTATION

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

It is possible to build a global SCADE Suite model, which formalizes the global behavior of the application, while implementing the code on different tasks. While it is also possible to build and implement separate independent models, this global model allows representative Model-in-Loop testing and formal verification of the complete system. The distribution over several tasks requires specific analysis and implementation (see [Camus] and [Caspi] for details).

9.3.6 Integration of AUTOSAR software components

As described in Section 3.3.6 and Figure 28, a specific integration workflow is available for AUTOSAR software components developed in Scade.

Before going into the details of the SCADE AUTOSAR integration workflow, we will describe the AUTOSAR integration concepts and how they relate to the Scade language.

AUTOSAR INTEGRATION CONCEPTS

— VariableAccess

A software component (SWC) has ports to connect with other SWCs, that “carry” the data through typed interfaces with fields. A SWC can contain several internal variables that are shared amongst the Runnables. These variables can be read and written using dedicated ports. A Runnable accesses to the SWC ports through data access.

The AUTOSAR standard allows:

- multiple read or write on the same port, leading to multiple read or write to the same port field (linked to an external data or an internal variable)
- multiple inputs (or outputs) of the same Runnable accessing the same port field (linked to an external data or an internal variable) leading to multiple read or write to the same data

All these accesses correspond to a **VariableAccess** in the AUTOSAR terminology.

— Implicit communication

Implicit communication in AUTOSAR is managed by the AUTOSAR RTE to ensure that:

- All writes are performed before calling a Runnable, so the Runnable reads stable values.
- All Runnable’s writes are performed when its execution terminates.

This behavior is fully aligned with the Scade language paradigm. For this reason, all inputs/outputs of a Runnable associated with an Implicit communication (between SWCs or with internal variables or parameters) are synchronized as Scade inputs/outputs. In case of a read/write operation, one input and one corresponding output are created to separate the data and to make the input stable by buffering.

In case of several accesses to the same data using different Runnable inputs and/or outputs, one input, or output, or pair is created. The binding of the generated C code with the RTE functions for each access is done in the order of the signals in the Scade model, which follows the Runnable description.

— Explicit communication

Explicit communication in AUTOSAR is performed each time a write is performed, in contrast with Implicit, where only the last write is emitted. Therefore, it is possible to have several emissions on a given port and therefore have several reads with different values. Explicit communication is used for performance reason or to ensure up-to-date information with Basic Software components (BSW) like Non-volatile Memory (NvM).

Note: it is not specified in the standard if a communication must be done at each cycle.

The support of explicit access concerns communications or inter-runnable variables. Explicit accesses for communications are given by *VariableAccesses* in the *DataReceivePointByValue, DataReceivePointByArguments, DataSendPoint, ReadLocalVariable and WriteLocalVariable roles or in the ReadLocalVariable and WrittenLocalVariable of VariableDataPrototypes* in the

ExplicitInterRunnableVariable role. Each explicit access generates a Scade imported operator, that offers the proper interface and whose implementation calls the corresponding RTE function (see [SCS-ACG-TOR]).

The following examples show how an AUTOSAR explicit communication is automatically transformed into a Scade imported function and how this function is implemented.

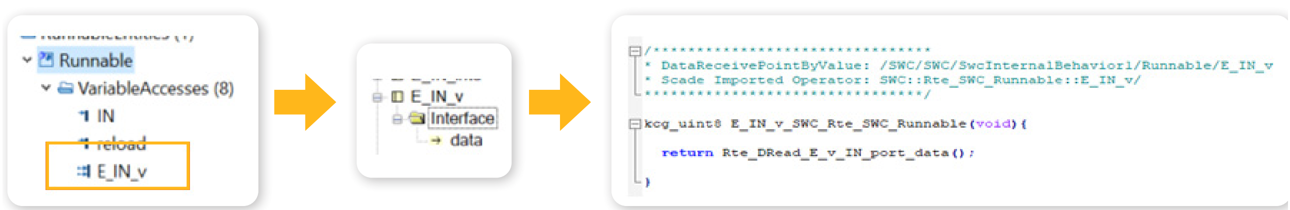


FIGURE 83: EXPLICIT READ OF A VARIABLEDATAPROTOTYPE IN PORTPROTOTYPE

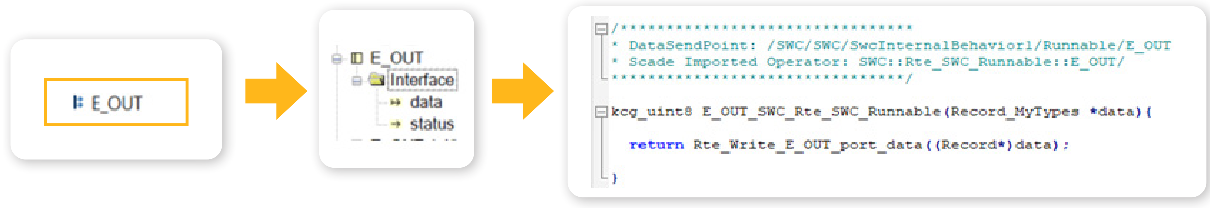


FIGURE 84: EXPLICIT WRITE OF A VARIABLEDATAPROTOTYPE IN PORTPROTOTYPE

Note: The generated imported function has specific annotations that allow to traceback to the original ARXML artefact.

— *Server calls*

A **Server call** corresponds to a specific AUITOSAR RTE API function that can:

- provide data to an external SWC
- get data from an external SWC
- have a specific action on an external SWC

Non exhaustive examples are:

- read diagnostic information from the diagnostic manager
- read/write a data element from the Non-volatile Memory (NvM) manager
- restore default values in NvM

These functions can be considered as inputs or outputs, and all have a return status that indicates if the call succeeded or failed. For instance, for NvM, a call to the GetErrorStatus API function returns a code indicating the nature of the error.

Figure 85 lists the supported **Server calls** for the NvM manager. Other services follow the same API style.

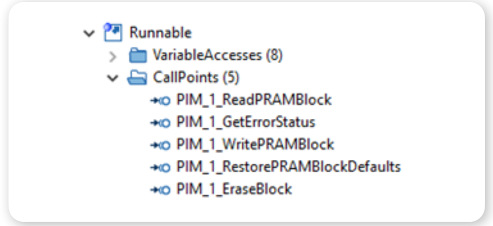


FIGURE 85: LIST OF SERVER CALL POINTS FOR A RUNNABLE

Figure 86 shows how the ReadPRAMBlock **Server call** is implemented in a SCADE imported operator. A ReadPRAMBlock **Server call** copies data from the NvM into a Per-Instance Memory PIM instance.

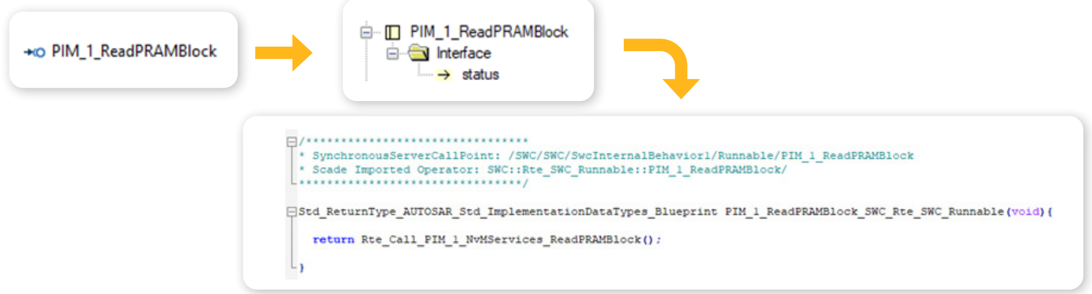


FIGURE 86: MODELING AND IMPLEMENTATION OF THE READPRAMBLOCK SERVICE

— *Per-Instance Memory (PIM)*

A Per-Instance Memory (PIM) is a chunk of memory declared in the scope of the internal behavior of a SWC. This memory can be typed (a record for instance) or not (an array of bytes). A PIM is accessed using a RTE function that returns the corresponding pointer. The memory is freely readable/writable with no protection mechanism. A PIM can be used to store data between executions of a Runnable, to exchange data between Runnables of the same SWC, or as an interface with services (to get/set data).

For exchanging data with other Runnables, one can also use an Inter-Runnable Variable. This variable is accessed using **VariableAccess** concepts, therefore read/write are controlled.

PIM usage is mandatory with Non-Volatile Memory manager service, in the case of load/save of data at boot-time and shutdown-time. Once it is loaded, the content is accessed by Runnables. It is expected that the data is quite large (*e.g.,* a full array of configuration values), so performance must be considered to avoid copies at each cycle.

A PIM and a NvM can be associated for data transfer through the system configuration. The association is given in a specific memory configuration section of the ARXML file, which is not in the SWC description section. For instance, the NvM_WritePRAMBlock function has its name derived from the port carrying the service call. It is a unique function that performs the copy from a NvM to a PIM, without having the NvM or the PIM as parameters. As the configuration gives the pairing, the RTE generator provides a proper behavior.

From the AUTOSAR RTE point of view, a PIM is accessed using a pointer returned by a function associated with the PIM. It is not a specific input or output of a Runnable, but this function must be called from the Runnable code, as given in the Runnable code specification. There is no means to specify in AUTOSAR that a Runnable has access to a given PIM.

The SCADE Automotive Package provides the following access to a PIM:

- When a PIM is used by a Runnable, a specific annotation for read and/or write is added to the Runnable (see Figure 87). This annotation is used during the synchronization to associate the Scade operator corresponding to the Runnable with the synchronized data corresponding to the PIM.
- A PIM corresponds to an I/O of the Runnable implemented by a Scade operator to highlight its use in the application. Two possible implementations are provided:
    1. **Buffered:** A PIM becomes an input and/or an output of the Scade operator, depending on if it is read, write, or both. In the last read/write case, one input and one output are created. The input is a buffered copy of the PIM before the cycle execution, and the output is bound to the PIM itself. The input/output type is the type as defined in the ARXML (an array, a structure, ...)

2. **Imported type/pointer:** A PIM is an input of the Scade operator, but it represents a direct access to the PIM memory. Its type is an *imported type,* meaning it is abstract from the Scade point of view. The underlying type is a C pointer, as the return type of the PIM access function generated during the contract phase. The input pointer is then used for *"set"* and *"get"* accesses.

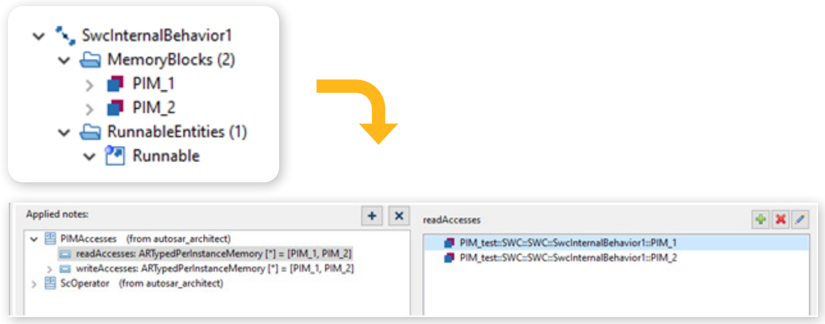


FIGURE 87: PIM AND RUNNABLE ASSOCIATION

The objective of the **Buffered PIM** is to comply with the Scade semantics and to avoid side effects. The data provided by the AUTOSAR RTE function is copied into a buffer, which is passed as input. The output parameter of the KCG generated C function is directly associated with the PIM. The output is directly written to the PIM. The inconvenience of this solution is that there is a systematic copy at each cycle. This may introduce execution time penalties if the PIM content is large and rarely updated.

With the **Imported Type/pointer PIM** solution, the PIM input is considered as an imported type, which is the return type of the PIM access RTE function. Passing the PIM data into the model is at the lowest cost as it corresponds to copying a scalar. There are no more copies of a whole data structure. As the PIM input type is imported, dedicated imported functions to access the data are needed. The solution is provided using *setter/getter* functions. These imported operators are generated during the synchronization from an AUTOSAR architecture to a SCADE Suite project. SCADE ACG generates the corresponding C code.

Figure 88 illustrates two different PIMs: one is a structure with two fields, the other one is an array. After synchronization, dedicated packages are created with imported PIM-related type definitions and the setter/getter imported functions are created. The Runnable has inputs with imported type for each used PIM.

The setter/getter function prototypes correspond to:

- an input to access the PIM
- an input or an output for the required data (set or get), for each field or cell. For a PIM which is a structure, the required field is the name of the function (*e.g.*, get_field1()). For a PIM which is an array, an additional input gives the indices of the element to get or set
- an output which copies the value of the input PIM (the pointer value)

The PIMs are read/written by the Runnable which has two PIM inputs. The declaration of the imported type associated with the Record PIM, and the corresponding set/get functions for each field. The details for the *get_field1* function are its declaration in Scade and its implementation in C.

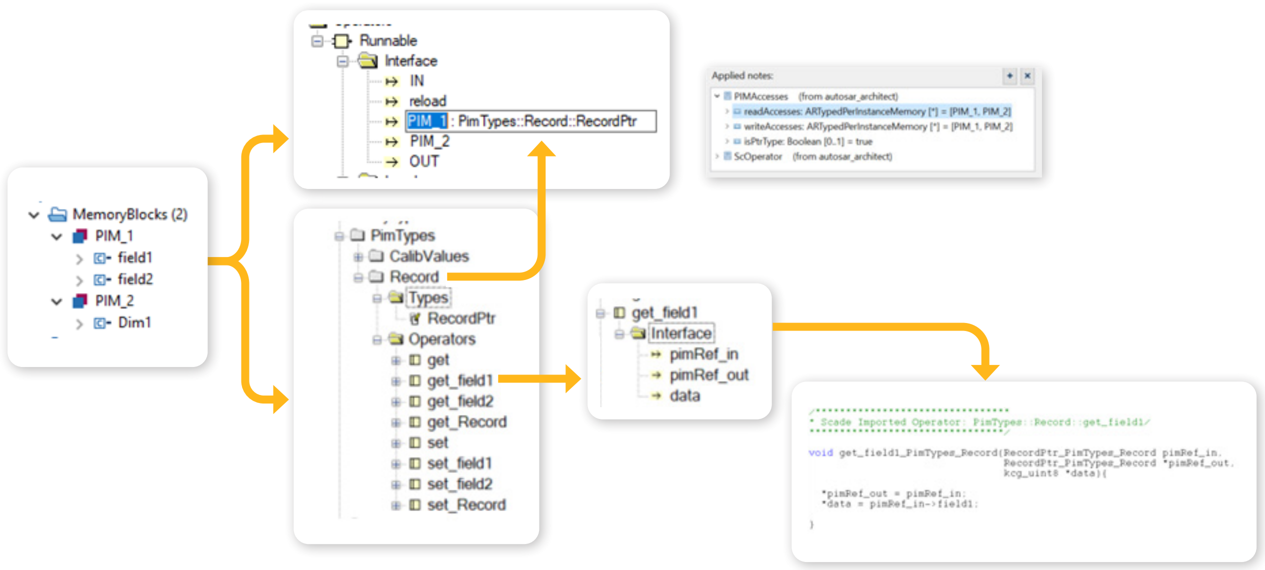


FIGURE 88: PIM SYNCHRONIZATION

THE SCADE AUTOSAR INTEGRATION PROCESS

Let us now take a step back and consider the overall integration process with SCADE Architect and SCADE Suite. Figure 89 provides a high-level description of the SCADE AUTOSAR workflow

1. An AUTOSAR software architecture design described in an ARXML Authoring Tool is imported into SCADE Architect via an ARXML System Description file. The design may be updated in SCADE Architect and re-exported.
2. Selected Runnables are synchronized from ARXML in SCADE Suite as Scade root operators. This synchronization is bi-directional.
3. The SCADE Suite user designs the behavior of a Runnable from the software requirements specification.
4. AUTOSAR-compliant C code for each Runnable is automatically generated from the SCADE Suite model using the AUTOSAR Code Generator (SCADE ACG) tool.

The SCADE Automotive Code Generator for AUTOSAR (ACG) produces C code that can be readily integrated to AUTOSAR RTE functions. The ACG code generator has been qualified for ISO 26262:2018 at TCL3. For further details on SCADE ACG qualification, see Appendix E.2.

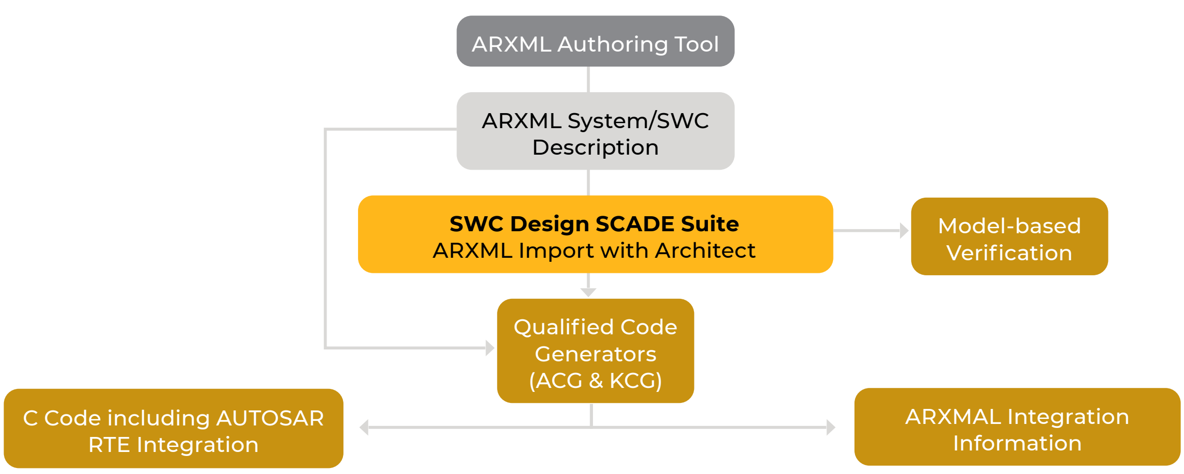


FIGURE 89: CODE GENERATION FOR AN AUTOSAR SOFTWARE COMPONENT

The detailed workflow for Runnables development is described in Figure 90 and is made of the following four steps:

1. A given software component (SWC) is designed in SCADE Architect as containing Runnables ①, with their inputs/outputs (VariableAccesses) and the Server calls that must be performed in the Runnable's behavior (CallPoints). The Per-Instance Memories (PIM) ② are also given within the SWC (MemoryBlocks).
2. The AUTOSAR project is synchronized as a SCADE Suite project. Each selected Runnable becomes a Scade root operator ③. The SCADE model inputs/outputs of the new operator correspond to the AUTOSAR implicit inputs/outputs of the Runnable. There are also specific inputs/outputs for PIMs. The synchronization process also produces:
   - Specific imported operator definitions related to explicit communications and Server calls ④
   - Specific type and imported operator definition related to PIM usage ⑤
3. The design of the root "Runnable" operators is done following the requirements and using the generated imported operators for the corresponding RTE calls.
4. Once the operators design is achieved, the final C code is generated by ACG:
   - ACG calls KCG to generate the C code from the Scade "Runnable" operators ⑥.
   - ACG generates the C code corresponding to the imported operators produced by the synchronization. The code of such imported operators is a call to the corresponding RTE API function ⑦.
   - ACG generates the AUTOSAR RTE compliant Runnable C function ⑧. The code of that function simply calls the KCG generated function, binding the input/output parameters to the proper RTE input/output function calls.

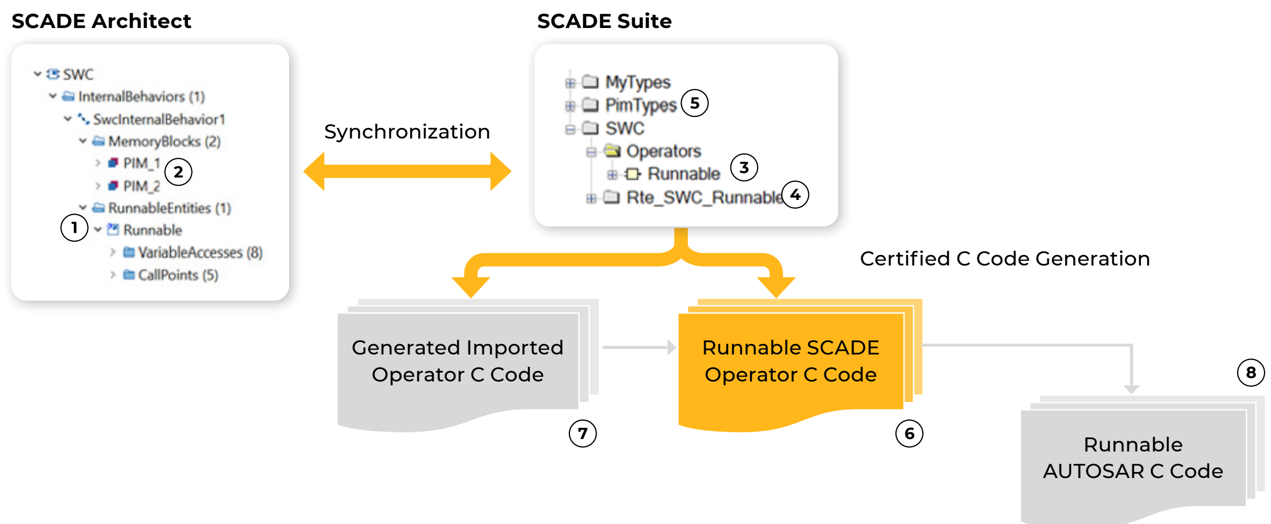


FIGURE 90: RUNNABLE DEVELOPMENT FLOW IN SCADE ARCHITECT AND SCADE SUITE

SAFETY IMPACT ANALYSIS OF THE AUTOSAR FEATURES SUPPORTED BY SCADE

We now need to understand the safety impact of the mechanisms that we have described above (implicit and explicit communication, Server calls, and PIM). For each of these features, we need to identify if there can be an impact regarding the determinism of the application. When a safety impact is confirmed, we provide a design rule that can be used to preserve determinism.

We will illustrate this safety impact analysis through the example of **Explicit communication** while considering the cases allowed by the AUTOSAR standard regarding this type of communication:

1. **Single access** is the easy case since there must be only one read of a given VariableAccess or one write to a given output port at each cycle. In this case, there is no safety impact, but as the read/write are done using imported operators, it must be checked that the corresponding operators are called only once in each cycle.
2. **Multiple accesses** to the same VariableAccess at each cycle: this can be done using the imported operators:

   – using the same imported operator: i.e., several reads of the same input in the same cycle are possible, as shown in Figure 91

   – using different imported operators: i.e., several reads of from different inputs referencing the same VariableAccess

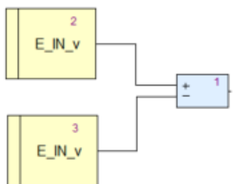


FIGURE 91: MULTIPLE READ OPERATIONS

In Figure 91, there are two instances of the E_IN_v operator, each reading the value of the corresponding input. The two calls correspond to two values. The two arguments of the subtraction must be computed before doing the subtraction itself, but there is no specific order to evaluate them. As the two E_IN_v calls are independent, no read order is defined. If the values 4 and 5 are sent in that order on the port E_IN_v, the computation could be (4-5) or (5-4).

This implementation violates the determinism of the application which may lead to system failure and violation of a safety goal.

Two solutions are possible:

- Either it is guaranteed that there is only one value sent during the execution cycle: in that case, only one E_IN_v must be performed during a cycle, and its results can be stored in a local variable. The Scade model implementing the Runnable should contain only one call, or if there are several calls there must be exclusive (*e.g.,* in different states of a state machine).
- Or there are several values sent during the execution cycle, and a specific logic must be designed to ensure the execution order. Figure 92 shows a possible design using *activate if* to create a dependence to ensure determinism of the example described in Figure 91.

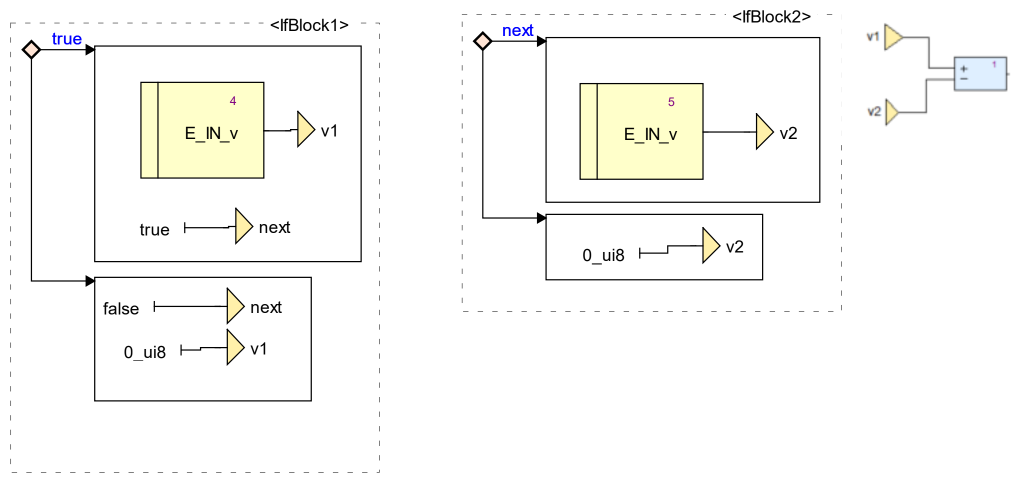


FIGURE 92: HANDLING MULTIPLE READS

This is similar in the case of outputs. In Figure 93, the two calls to E_OUT_int8 are independent. Therefore, we do not know which value (4 or 5) is produced at the end.

This implementation violates the determinism of the application which may lead to system failure and violation of a safety goal.

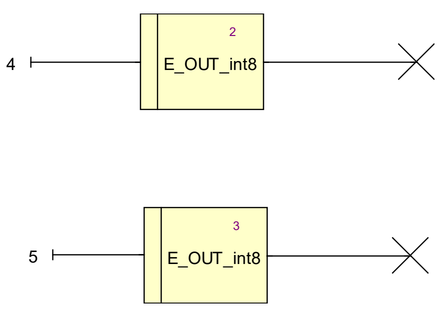


FIGURE 93: PARALLEL OUTPUTS

A similar design based on the Scade "*activate … if*" construct as in Figure 92 can be used to force the order of the output.

An exhaustive analysis regarding the safety impact of AUTOSAR Implicit and Explicit communication, Server calls, and PIM is provided in [SCS-ACG-Safety Analysis]. This study provides applicable design and verification conditions that can be used to preserve determinism of the application. Table 25 below provides a description of the additional application conditions for SCADE ACG 2.1 produced by this safety analysis. The overall ACG application conditions are provided in [SCS-ACG-RN].

Note: For the complete and formal description of the ACG application conditions, the reader must refer to [SCS-ACG-RN] and [SCS-KCG-Safety Case].

TABLE 26: SCADE AUTOMOTIVE CODE GENERATOR FOR AUTOSAR (ACG) ADDITIONAL APPLICATION CONDITIONS

Source: Table 1 in [SCS-ACG-Safety Analysis]

Id	Category	Description	Origin	Applicability	Verification Means
ACG-001	Multiple Accesses	In the Architecture, a Runnable **VariableAccess** shall be referenced by: – only one port for reading and/or one port for writing – or only one port for read/write	The same **VariableAccess** can be referenced by different ports. Therefore, it is possible to read (or write) the same data using different inputs (or outputs). This does not fulfill the Scade language semantics and may break determinism of the application.	– Runnable variable accesses	Review (Architecture): The user shall check and ensure that for each **VariableAccess**, there is only: – one read and/or one write – or only one read/write

Id	Category	Description	Origin	Applicability	Verification Means
ACG-002	Single Read	Imported operator providing an input data from the application environment must be called only once during a Runnable execution cycle. Therefore, if such imported operator has several instances in the Scade model, the activation of the instances shall be exclusive.	To ensure the determinism of the application, the Scade semantics relies on the fact that inputs do not change during execution cycle.	– Explicit read access – Server call: e.g., NvM to PIM read – PIM read access	Review (Design): – The user shall check and ensure that an imported operator providing an input data from the application environment is called only once during a Runnable execution cycle. – If several explicit reads or calls are performed during a Runnable execution cycle, the user shall check and ensure that they are exclusive (*e.g.*, in different states).
ACG-003	Single Write	An imported operator sending an output data to the application environment must be called only once during a Runnable execution cycle. If they are called several times in a Runnable cycle, the calls must be exclusive.	To ensure determinism of the application, the Scade semantics relies on the fact that an output is produced (and thus is considered as valid) only after the execution cycle termination and that this output value does not change until the termination of the next execution cycle.	– Explicit write access – Server call: e.g., PIM to NvM write – PIM write access	Review (Design): – The user shall check and ensure that an imported operator sending an output data to the application environment is called only once during a Runnable execution cycle. – If several explicit writes or calls are performed during a Runnable execution cycle, the user shall check and ensure that they are exclusive (*e.g.*, different states)
ACG-004	Multiple Read	Adequate design must be established to ensure that multiple calls of an imported operator, providing an input data from the application environment, that are performed in the same Runnable execution cycle are properly ordered as stated in the specification.	The Scade language is declarative and only data dependencies matter. Scade semantics gives no execution ordering for two data independent equations. Having several values for the same inputs does not fulfill the Scade language semantics and may break the determinism of the application.	– Explicit read access – Server call: e.g., PIM to NvM read – PIM read access	Review (Design): The user shall check and ensure that, if multiple calls of an imported operator providing input data from the environment are performed in the same Runnable execution cycle, then the multiple calls are properly ordered by a specific design. Note: independent verification shall ensure that write and read operations performed by producers and consumers (Runnables and/or servers) are done consistently.

Id	Category	Description	Origin	Applicability	Verification Means
ACG-005	Multiple Write	Adequate design must be established to ensure that multiple calls of an imported operator providing an output data to the application environment that are performed in the same Runnable execution cycle are properly ordered as stated in the specification.	The Scade language is declarative and only data dependencies matter. Scade semantics gives no execution ordering for two data independent equations. Having several values for the same output does not fulfill the Scade language semantics and may break the determinism of the application.	– Explicit write access – Server call: e.g., PIM to NvM write – PIM write access	Review (Design): The user shall check that, if multiple calls an imported operator sending output data to the environment are performed in the same Runnable execution cycle, then the multiple calls are properly ordered by a specific design. Note: independent verification shall ensure that write and read operations performed by producers and consumers (Runnables and/or servers) are done consistently.
ACG-006	Mixed Read and Write	Adequate design must be established to ensure that calls of imported operators are properly ordered in case of: – these operators read or write an external data of the application environment – and that a mix of read and write calls are performed in the same Runnable execution cycle.	Using imported operators also for multiple read/write of an external data, being a PIM or a data manage by a server. This does not fulfill the Scade language semantics and may break the determinism of the application.	– Server calls – PIM-related set/get calls – Variable accesses	Review of order of PIM read/ write operations (Design): – The user shall check and ensure that PIM write/ read accesses using a PIM imported type are properly ordered. Review of order of server calls read/write operations (Design): – The user shall check and ensure that server calls (write/read of a given data) are properly ordered. *Note: independent verification shall ensure that write and read operations performed by producers and consumers (Runnables and/or servers) are done consistently.*

9.4 Software Verification with SCADE Test Target Execution and SCADE Test Model Coverage

The verification process with SCADE Test Target Execution is an efficient and optimized testing process that fully satisfies the requirements of [ISO 26262-6:2018] while optimizing testing efforts:

1. **The SCADE testing process is efficient:** test cases and procedures are primarily developed from software requirements. This verification strategy focuses first on functionality and integration issues that are often poorly and lately addressed in a traditional verification process.
2. **The SCADE testing process optimizes testing efforts:** in the context of ASIL C and D applications, the development of test cases and procedures usually requires a huge effort to satisfy all testing objectives. When using SCADE, this testing effort is significantly reduced as the same requirement-based test cases and procedures (see Section 8.3.2) are used for both model Model-in-the-Loop testing on host and integration testing on target as in Figure 94.

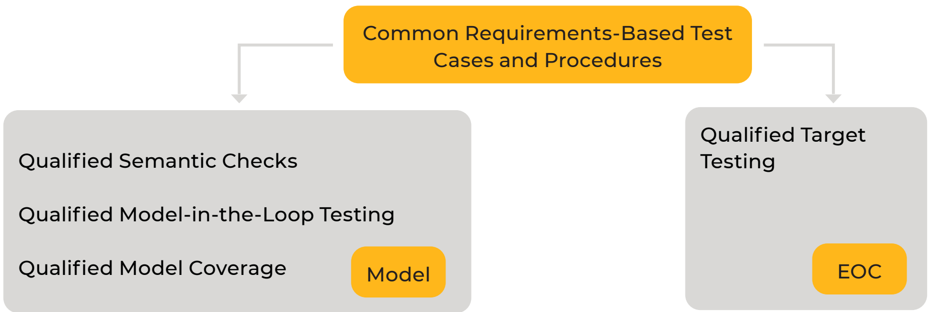


FIGURE 94: FACTORING MODEL-IN-THE-LOOP AND TARGET TESTING WITH SCADE TEST

An overview of the SCADE testing process is provided in Figure 95.

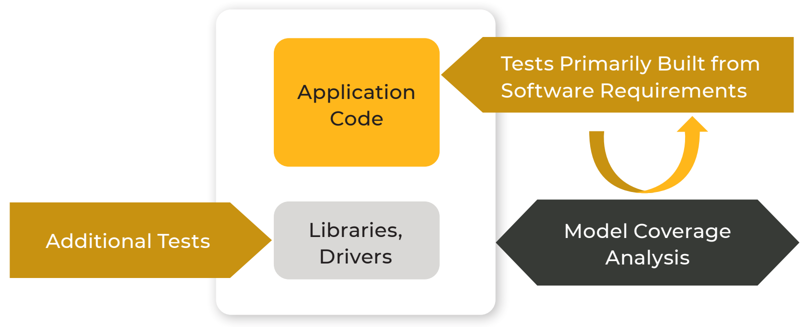


FIGURE 95: OVERVIEW OF THE SCADE TESTING PROCESS

The testing effort is mainly focused on the software requirements-based testing for the application code. This is the software part that is undergoing the most modifications during the software life cycle. On the other hand, library components and drivers are usually developed, using either SCADE modeling or manual coding and additional tests must be considered in this context. Because the corresponding code is quite stable during the software life cycle, the additional testing effort is usually not significant for this software part.

9.4.1 Compliance and robustness of the Executable Object Code (EOC) with the software requirements

Test cases and procedures are developed firstly based on the software requirements and executed in the target environment. They should include normal range test cases and robustness test cases.

In the context of SCADE Suite, users can reuse existing test cases and procedures developed for Model-in-the-Loop host testing (see Section 8.3.2). SCADE Target Test Harness Generator allows automatic translation of Host test cases to Target test cases for RTRT, LDRA, VectorCAST, and generic testing environments as shown in Figure 96.

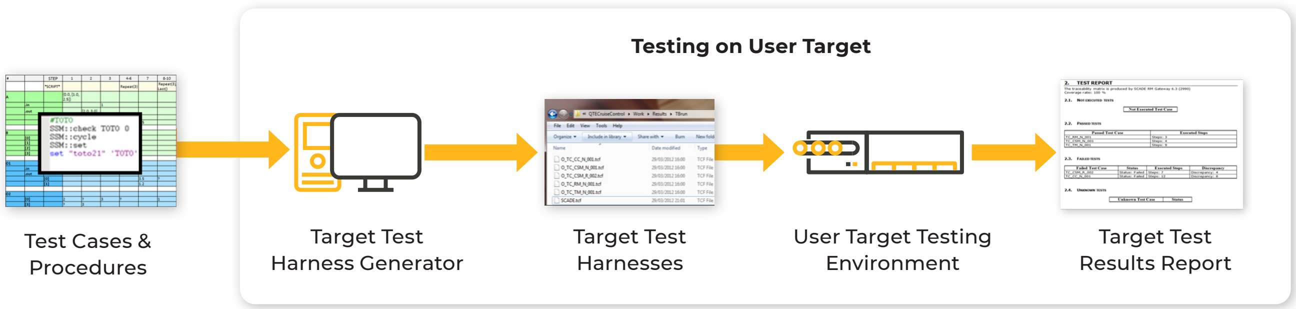


FIGURE 96: RE-RUNNING TEST CASES AND PROCEDURES WITH SCADE TEST TARGET EXECUTION

The position of SCADE Test Target Execution within the software development and verification flow is described in the Figure below.

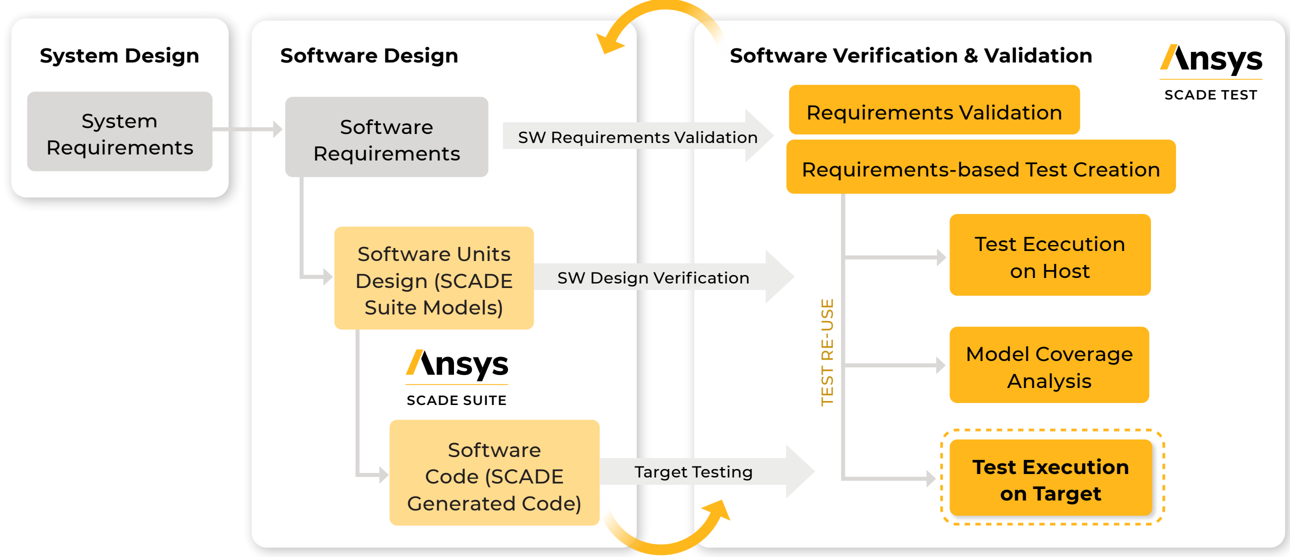


FIGURE 97: POSITIONING OF SCADE TEST TARGET EXECUTION WITHIN THE VERIFICATION FLOW

SCADE Test Target Execution has been qualified for [ISO 26262:2018] at TCL3. The categorization of the tool as TCL3, the qualification method and compliance to Clause 11 of [ISO 26262-8] are described in the compliance document [SCS-STE-COMPL]. For further details on SCADE Test Target Execution, see Appendix E.4.

9.4.2 Compliance and robustness of the Executable Object Code (EOC) when using library operators

When library operators are used, their implementation must be tested from **additional software requirements** established for these operators, as described in Section7.3.3, and additionally, the integration of these operators within upper-level operators must be tested.

This integration and verification activity is complete when full model coverage is achieved with SCADE Test Model Coverage as defined in Section 8.4.2 :

- for structural model-level coverage criteria, according to the requirements of the ASIL (Influence, ODC, or OMC/DC)
- for additional user-defined coverage objectives related to support equivalence classes testing

9.5 Takeaway from Using SCADE Suite and SCADE Test for Software Integration and Verification

This Chapter has established how SCADE Architect, SCADE Suite, SCADE Test Model Coverage, and SCADE Test Target Execution support software integration and verification activities, for the components that have been developed in Scade.

This can be seen in four ways, depending on the tool that is used:

1. SCADE Architect
   - SCADE Architect, together with SADE Suite (see Next), supports the integration of Scade generated code with AUTOSAR software
2. SCADE Suite
   - verifies that interfaces between components are correctly typed and that additional design rules are obeyed
   - supports the automatic integration of multiple software units while generating code from an integration model (see Figure 55 in Section 7.4)
   - supports the integration effort with the execution environment (e.g., AUTOSAR RTE)
   - supports the evaluation of WCET and memory consumption
3. SCADE Test Model Coverage
   - ensures full coverage of data and control flows across software components interfaces
4. SCADE Test Target Execution
   - automates the translation of requirements-based test cases specified for host testing to target compatible requirement-based test cases

A detailed analysis of the level of support of SCADE Suite, SCADE Test Model Coverage, and SCADE Test Target Execution for software verification and integration is provided in Appendix C.6.

10
TESTING OF THE EMBEDDED SOFTWARE

10.1 Objectives and Work Products

The objectives of this sub-phase (Clause 11 of [ISO 26262-6:2018]) are to provide evidence that the embedded software:

- fulfils the safety-related requirements when executed in the target environment
- contains neither undesired functionalities nor undesired properties regarding functional safety

The inputs to the embedded software testing sub-phase are:

- software architectural specification
- hardware-software interfaces specification
- software requirements specification
- configuration data and calibration data, if any
- software units design specification
- software units implementation
- software verification specification
- software verification report
- embedded software

Work products are:

- software verification specification (final)
- software verification report (final)

10.2 Requirements and recommendations

Section 11.4.1 of [ISO 26262-6] specifies the test environment on which testing shall be conducted and the testing methods to be used. This is described in the Tables below.

TABLE 27: TEST ENVIRONMENTS FOR CONDUCTING THE SOFTWARE TESTING

Source: Table 13 in ISO 26262-6:2018

Methods		ASIL			
		A	B	C	D
1a	Hardware-in-the-loop	++	++	++	++
1b	Electronic control unit network environments[a]	++	++	++	++
1c	Vehicles	+	+	++	++

[a] *Examples include test benches partially or fully integrating the electrical systems of a vehicle, "lab-cars" or "mule" vehicles, and "rest of the bus" simulations.*

TABLE 28: METHODS FOR TESTS OF THE EMBEDDED SOFTWARE

Source: Table 14 in ISO 26262-6:2018

Methods		ASIL			
		A	B	C	D
1a	Requirements-based test	++	++	++	++
1b	Fault injection test[a]	++	++	++	++

[a] *In the context of software testing, fault injection test means to introduce faults into the software by means of e.g. corrupting calibration parameters.*

TABLE 29: METHODS FOR DERIVING TEST CASES FOR THE TEST OF THE EMBEDDED SOFTWARE

Source: Table 15 in ISO 26262-6:2018

Methods		ASIL			
		A	B	C	D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes	+	++	++	++
1c	Analysis of boundary values	+	+	++	++
1d	Error guessing based on knowledge or experience	+	+	++	++
1e	Analysis of functional dependencies	+		++	++
1f	Analysis of operational use cases[a]	+	++	++	++

[a] *Examples for operational use cases for software can include software update in the field, starting the nominal application only if the integrity of the software is ensured by bootloader, safety-related behaviour of the embedded software in different operational modes such as start-up, diagnosis, degraded, power-down (going to sleep), power-up (waking up), calibration, functions for mode synchronization between different ECUs or end-of-line-specific test bench mode for safeguarding production personnel.*

10.3 Testing the Embedded Software with SCADE Suite and SCADE test

Let us come back to our original AEB example of Figure 33.

We can now exercise the final integration of the application software (the AEB function) through the connection to VRXPERIENCE Driving Simulator, which is used to create and simulate driving scenarios that are representative of the Operational Design Domain (ODD). The Scade AEB software model integrates with sensors and actuators for the Driving Simulator, as shown in Figure 98.

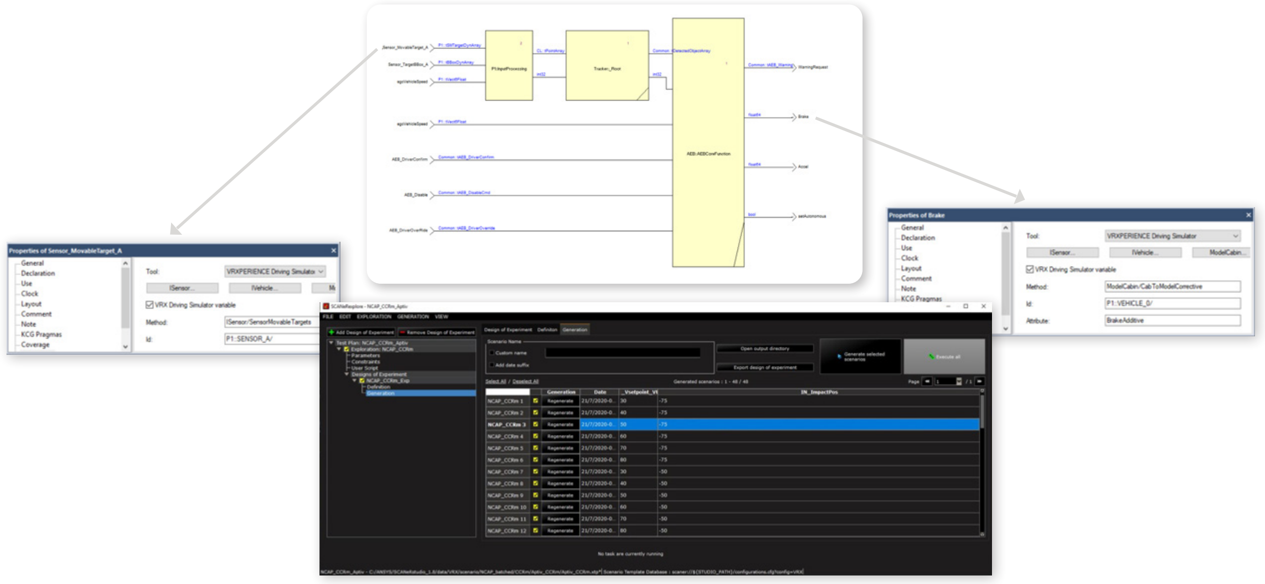


FIGURE 98: FINAL MODEL-BASED INTEGRATION TESTING OF THE AEB APPLICATION SOFTWARE

At this level, when a scenario is not giving expected results, the SCADE Suite debugger can be used to further understand the behavior of the Scade model of the full application. In the example below, a breakpoint set in the AEB model has been triggered; driving physics, radar tracking and AEB decision logic are all paused for examination. Step-by-step Co-simulation of driving with AEB function can be exercised.

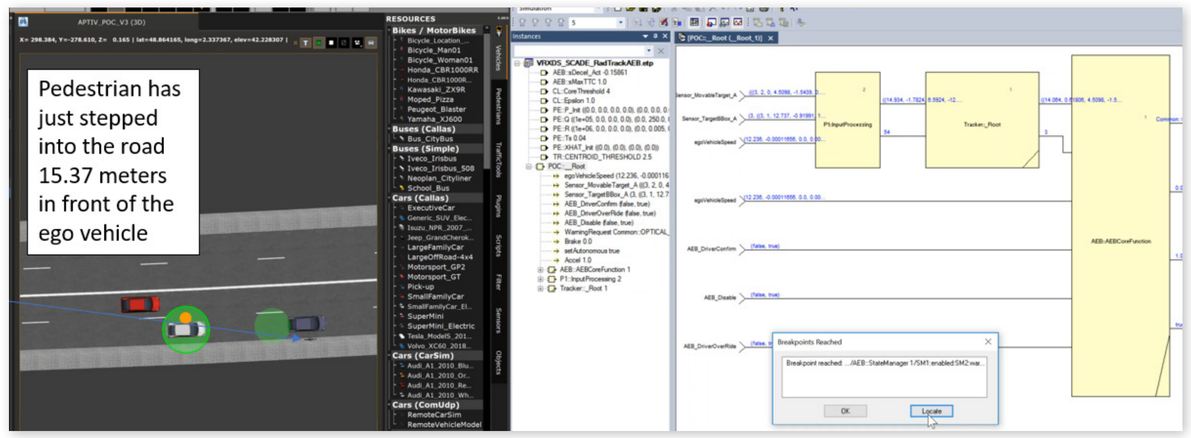
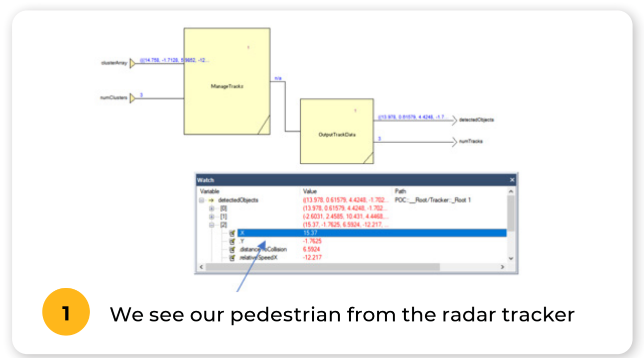
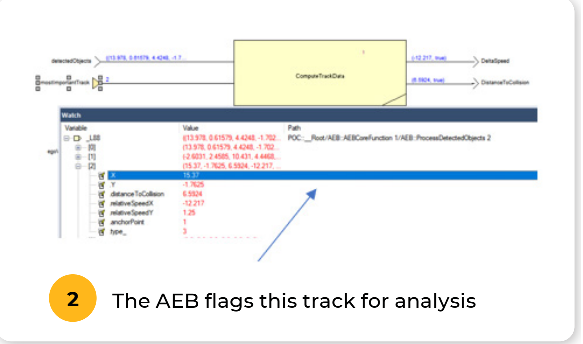


FIGURE 99: SETTING UP A BREAKPOINT IN THE AEB FUNCTION MODEL

As shown in the Figure below, a detailed analysis of the situation, at model-level, is now possible:





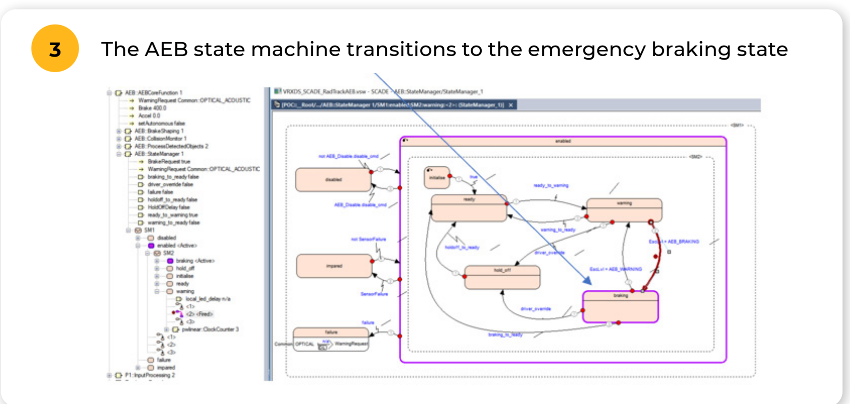


FIGURE 100: DETAILED ANALYSIS USING THE SCADE SUITE SIMULATION

To better assess how the AEB controller performs, it is also possible to build customized visualization with the Rapid Prototyper module of SCADE Test enabling model stimulation with easy-to-design interactive graphical panels. A library of predefined widgets is included; widgets can be customized, and this library is extensible with custom widgets. Based on these graphical panels, Rapid Prototyper also features automatic generation of Windows/PC standalone executables. This allows developers/testers to view how algorithms perform without even digging through the software models. This is illustrated by Figure 101.

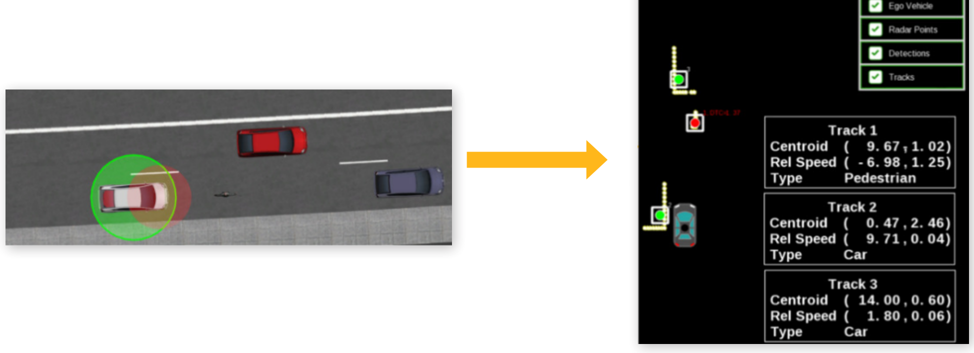


FIGURE 101: RAPID PROTOTYPING FOR AEB RADAR TRACKING

Model-in-the-Loop testing of the software application can be used to simulate full operational scenarios. The example below illustrates MiL testing of a AEB standard Car-to-Car Rear Moving (CCRm 7) NCAP scenario where a bug leading to a collision could be detected and later fixed.

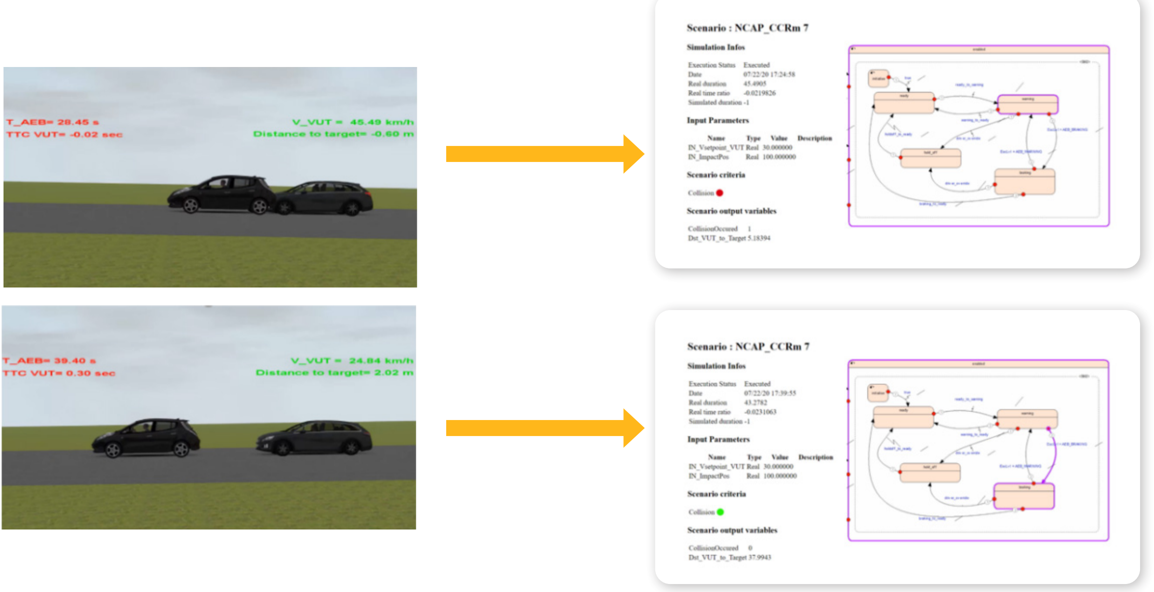


FIGURE 102: MIL TESTING OF NCAP AEB CCRM SCENARIO

Once, MiL testing has been done, complete scenarios can be saved in SCADE Test and re-used for HiL, ECU and Vehicle testing, as recommended in Table 24 above.

10.4 Takeaway from Using SCADE Suite and SCADE Test Target Execution for TESTING the Embedded Software

The support of SCADE Suite and SCADE Test, in combination with physics simulation (*e.g.*, Ansys VRXPERIENCE) and HiL testing (*e.g.*, National Instruments Veristand), can be seen as follows:

- SCADE Suite and SCADE Test are connected to HiL testing environments, including National Instruments Veristand for Processor-in-the-Loop testing
- SCADE Suite and SCADE Test are connected to physics simulation environments, including Ansys VRXPRIENCE and Ansys Twin Builder for simulation of operational scenarios
- SCADE Test ensures continuity between Model-in-the-Loop testing on host and final testing on target, through re-using the requirements-based test cases
- SCADE Test Model Coverage supports the analysis of how well functional dependencies and equivalence classes are covered by test cases

  – A detailed analysis of the level of support of SCADE Suite and SCADE Test for testing the embedded software is provided in Appendix C.7.

11
SUMMARY

In this handbook, we have presented a model-based approach for product development at the software level that provides efficient support for satisfying the requirements and objectives of [ISO 26262-6:2018].

This model-based approach is based on using the SCADE toolchain as the software development environment and it covers the complete flow of [ISO 26262-6:2018]:

- Specification of the software requirements
- Software architectural design
- Software unit design and implementation
- Software unit verification
- Software integration and verification
- Testing of the embedded software

If we revisit the generic model-based development workflow that was presented in Figure 5, we have seen that, while using the SCADE toolchain, we can remove or optimize several steps that had been previously identified in the generic workflow. This is depicted in Figure 103.

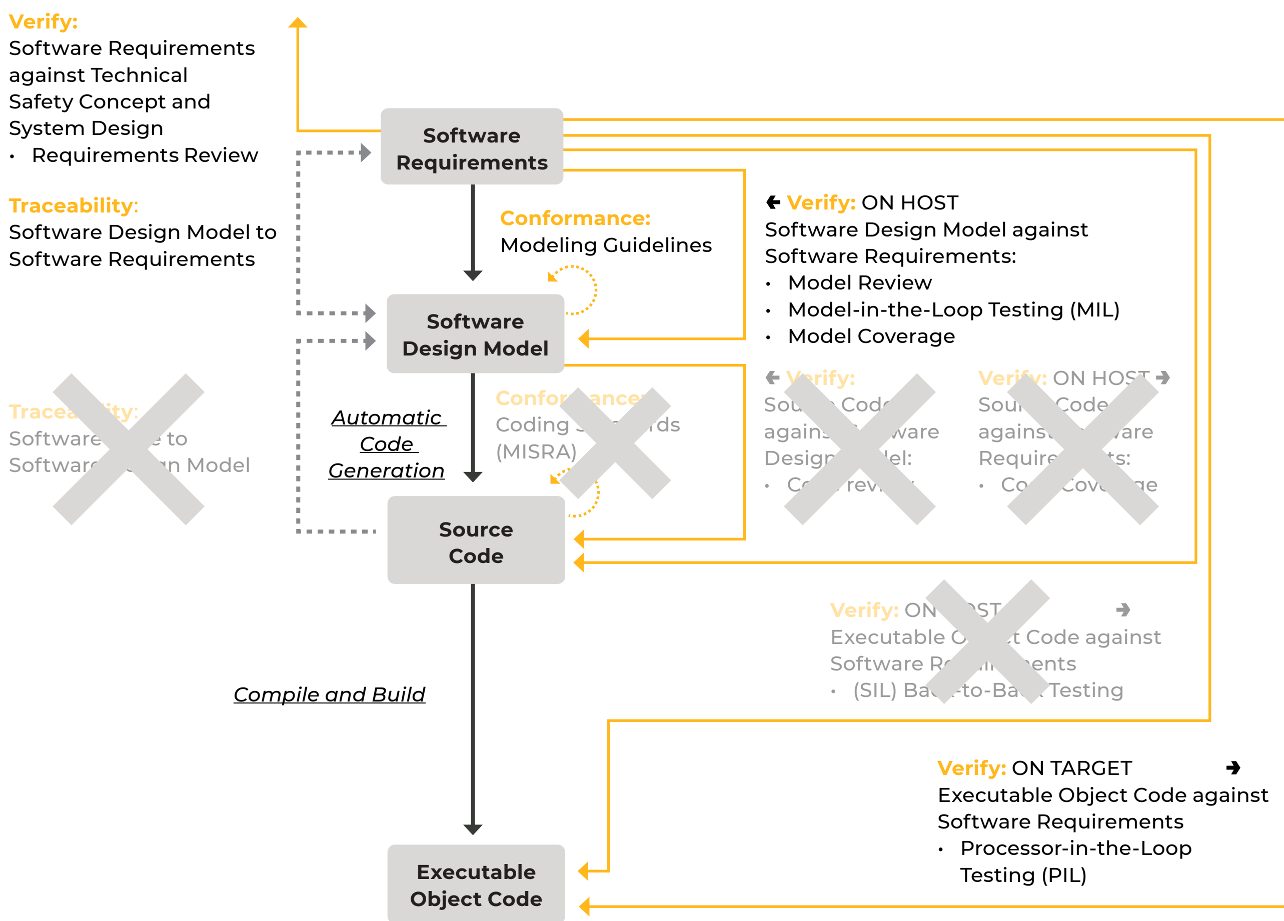


FIGURE 103: OPTIMIZATION OF THE GENERIC MODEL-BASED DEVELOPMENT WORKFLOW

The highlights of this optimized workflow are the following:

- Scade design models, covering both software architectural design and software unit design, are positioned as the cornerstone of the software development and verification workflow:
  - They are reviewed for conformance with the software requirements and the design modeling guidelines.

- They are used as the basis for software Model-in-the-Loop testing on host; test cases are based on the software requirements.
- They are used as the basis for structural coverage measurement at the model-level, in such a way that coverage analysis at code-level becomes unnecessary.

- Qualified code generation of the source code from Scade models automates the following activities:
  - Traceability between source code and software design models is automatically established by the code generator.
  - Code reviews are unnecessary because the conformance of the source code to the design model and the conformance of the source code to the coding guidelines are guaranteed.
- Finally, the requirements-based test cases that had been initially created for model-level testing are replayed in the target environment.

We therefore obtain a streamlined workflow, as shown in Figure 104.

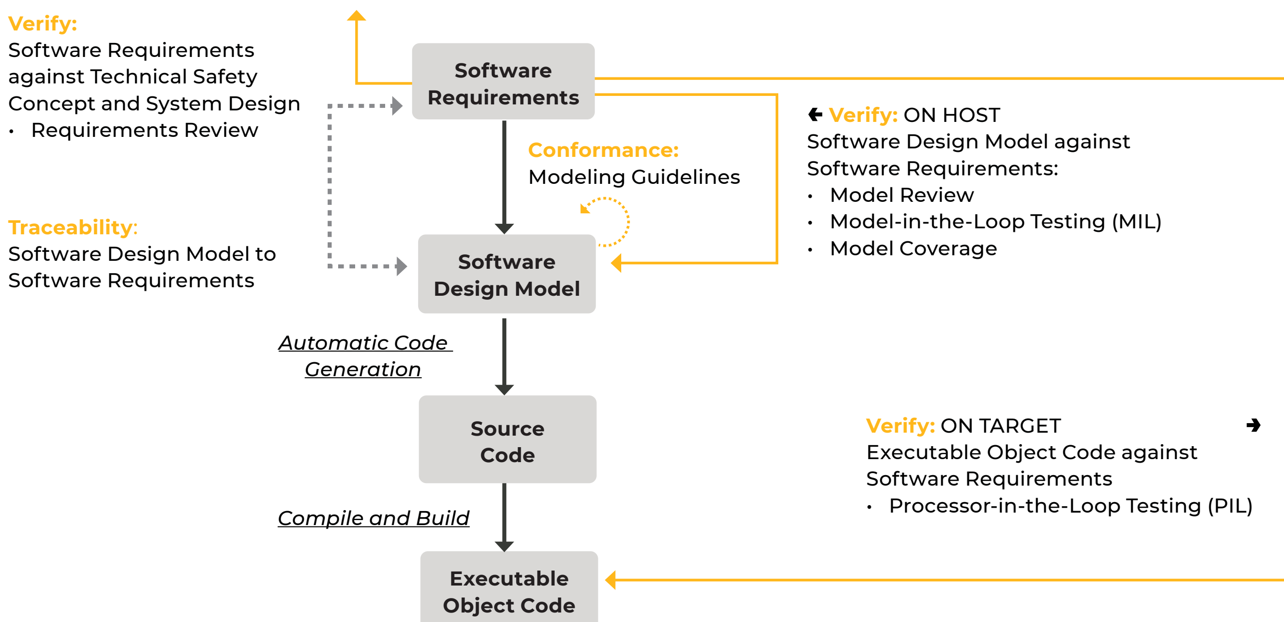


FIGURE 104: THE OPTIMIZED SCADE MODEL-BASED WORKFLOW

The key enablers for the above SCADE streamlined model-based workflow are:

- The formal nature of the Scade language and its fundamental properties (declarative, hierarchical, modular, concurrent, etc.)
- The ability of the Scade language to handle the overall design of the safety-related application software, including architectural and detailed design (controls and decision logic), thus supporting the integration and verification of the complete embedded application software
- The qualification of the appropriate tools in the SCADE toolchain:
  - Model reporter
  - Code generator
  - Coverage analyzer
  - Testing environment (host and target)

12 APPENDICES

APPENDIX A
ACRONYMS AND GLOSSARY

A.1 Acronyms

ACC	Adaptive Cruise Control
ACG	Automotive Code Generator for AUTOSAR
AEB	Automatic Emergency Braking
ALM	Application Lifecycle Management
API	Application Programming Interface
ARXML	AUTOSAR Extensible Markup Language
ASPICE	Automotive Software Performance Improvement and Capability dEtermination
ASIL	Automotive Safety Integrity Level
AUTOSAR	AUTomotive Open System ARchitecture
AV	Autonomous Vehicles
BMS	Battery Management System
BNF	Backus-Naur Form
BP	Base Practice
BSW	Basic Software
CCRm	Car-to-Car Rear Moving
COTS	Commercial Off-The-Shelf
CMS	Configuration Management System
CPU	Central Processing Unit
CVK	Compiler Verification Kit
DAL	Design Assurance Level
DC	Decision Coverage
DSM	Digital Safety Manager
e.g.	exampli gratia
ECU	Electronic Control Unit
EOC	Executable Object Code
EPS	Electric Power Steering
E/E	Electrical and/or Electronic
FHA	Functional Hazard Analysis
FIR	Finite Impulse Response
FMEA	Failure Modes and Effects Analysis
FMI	Functional Mock-up Interface

FMU	Functional Mock-up Unit
FSC	Functional Safety Concept
FSR	Functional Safety Requirements
FTA	Fault Tree Analysis
HAZOP	Hazard and Operability Analysis
HiL	Hardware-in-the-Loop
HIS	Hardware-software Interface
HMI	Human-Machine Interface
HTML	Hypertext Markup Language
HW	Hardware
Hz	Hertz
ICD	Interface Control Document
IDE	Integrated Development Environment
i.e.	id est
I/O	Input/Output
IP	Intellectual Property
IIR	Infinite Impulse Response
incl.	including
KCG	Qualified Code Generator
MC/DC	Modified Condition/Decision Coverage
MiL	Model-in-the-Loop
NvM	Non-volatile Memory
MB	Model-Based
MBD	Model-Based Development
MBSE	Model-Based System Engineering
N/A	Not Applicable
NaN	Not a Number
NCAP	New Car Assessment Program
N.B.	Nota Bene
PA	Process Attribute
PiL	Process-in-the-Loop
PIM	Per-Instance Memory
ODC	Observable Decision Coverage
ODD	Operational Design Domain
OMC/DC	Observable Modified Condition/Decision Coverage
OS	Operating System
QM	Quality Management
RM	Requirements Management

ROI	Return on Investment
RTOS	Real Time Operating System
RTE	Run-Time Environment
SAE	Society of Automotive Engineers
SCADE	Safety Critical Application Development Environment
SG	Safety Goal
SIL	Safety Integrity Level
SIP	Software Installation Procedure
SR	Software Requirement
SysML	Systems Modeling Language
SW	Software
SWC	Software Component
TAS	Tool Accomplishment Summary
TECI	Tool Life Cycle Environment Configuration Index
TCI	Tool Configuration Index
TD	Tool error Detection
TI	Tool Impact
TOR	Tool Operational Requirements
TORD	Tool Operational Requirements Data
TQL	Tool Qualification Level
TQP	Tool Qualification Plan
TSO	Timing and Stack Optimizer
TSC	Technical Safety Concept
TSV	Timing and Stack Verifier
UML	Unified Modeling Language
VFB	Virtual Functional Bus
vs.	versus
WCET	Worst Case Execution Time
w/o	without
wrt.	with respect to

A.2 Glossary

Architecture
Representation of the structure of an item that allows identification of building blocks, their boundaries and interfaces, and includes the allocation of requirements to these building blocks.

Automotive Safety Integrity Level (ASIL)
One of four levels to specify an item's necessary ISO 26262:2018 requirements and safety measures to apply for avoiding an unreasonable risk, with D representing the most stringent and A the least stringent level.

Branch coverage
Percentage of branches of the control flow of a computer program executed during a test.

Calibration data
Data that will be applied as software parameter values after the software build in the development process.

Condition
A Boolean expression containing no Boolean operators except for the unary operator (NOT).

Coverage analysis
The process of determining the degree to which a proposed software verification process activity satisfies its objective.

Deactivated code
Executable object code (or data) that, by design, is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component; or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options.

Dead code
Executable object code (or data) which exists as the result of a software development error but cannot be executed (code) or used (data) in an operational configuration of the target computer environment. It is not traceable to a system or software requirement.

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.

Embedded software
Fully integrated software to be executed on a processing element.

Error
Discrepancy between a computed, observed or measured value or condition, and the true, specified or theoretically correct value or condition.

Failure
Termination of an intended behavior of an item due to a fault manifestation.

Fault
Abnormal condition that can cause an item to fail.

Fault tolerance
Ability to deliver a specified functionality in the presence of one or more specified faults.

Formal methods
Descriptive notations and analytical methods used to construct, develop, and reason about mathematical models of system behavior. A formal method is a formal analysis carried out on a formal model.

Functional safety
Absence of unreasonable risk due to hazards caused by malfunctioning behavior of E/E systems.

Functional safety concept (FSC)
Specification of the functional safety requirements, with associated information, their allocation to elements within the architecture, and their interaction necessary to achieve the safety goals.

Hardware/software integration
The process of combining the software into the target computer.

Harm
Physical injury or damage to the health of persons.

Hazard
Potential source of harm caused by malfunctioning behavior of an item.

Hazard analysis and risk assessment (HARA)
Method to identify and categorize hazardous events of items and to specify safety goals and ASILs related to the prevention or mitigation of the associated hazards (3.75) in order to avoid unreasonable risk.

Hazardous event
Combination of a hazard and an operational situation.

Item
A system or a combination of systems, to which ISO 26262:2018 is applied, that implements a function or part of a function at the vehicle level.

Malfunction
Failure or unintended behavior of an item with respect to its design intent.

Model-based Development (MBD)
Development that uses models to describe the behavior or properties of an element to be developed.

Modified Condition/Decision Coverage (MC/DC)
Every point of entry and exit in the program was invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision was shown to independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.

Review
Examination of a work product, for achievement of its intended work product goal, according to the purpose of the review.

Risk
Combination of the probability of occurrence of harm and the severity of that harm.

Robustness
The extent to which software can continue to operate correctly despite abnormal inputs and conditions.

Standard
A rule or basis of comparison used to provide both guidance in and assessment of the performance of a given activity or the content of a specified data item.

Technical safety concept (TSC)
Specification of the technical safety requirements and their allocation to system elements with associated information providing a rationale for functional safety at the system level.

Systematic fault
Fault whose failure is manifested in a deterministic way that can only be prevented by applying process or design measures.

Test case
A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Test Procedure
Detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases.

Tool qualification

The process necessary to obtain the evidence that the software tool is suitable to be used in a way that the user can rely on its correct functioning, at the required level of confidence.

Traceability
An association between elements, such as between process outputs, between an output and its originating process, or between a requirement and its implementation.

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.

APPENDIX B
REFERENCES

[Ansys medini] Ansys medini analyze 2021R1, January 2021

[Ansys SCADE] Ansys SCADE 2021R1, Ansys, January 2021

[Ansys Twin Builder] Ansys Twin Builder 2021 R1, January 2021

[Ansys VRXPERIENCE] Ansys VRXPERIENCE 2021 R1, February 2021

[ASPICE] Automotive SPICE Process Reference Model Process Assessment Model Version 3.1, VDA, November 2017

[AUTOSAR] AUTOSAR Classic Release 4.4.0 documentation, www.AUTOSAR.org/standards/classic-platform/classic-platform-440, AUTOSAR, October 2018

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

[Caspi] "Integrating model-based design and preemptive scheduling in mixed time and event-triggered systems", N. Scaife and P. Caspi, Verimag Report Nr. TR-2004-12, June 2004

[CVK-UM] SCADE Suite CVK User Manual, Ansys, January 2021

[CVK-RM] SCADE Suite CVK Reference Manual, Ansys, January 2021

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

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

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

[EN 50128] CENELEC – EN 510128, Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems, CENELEC, June 2020

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

[IEC 61508] IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, IEC, April 2010

[IEEE-754] IEEE Standard 754-2008, IEEE Standard for Floating-Point Arithmetic, IEEE Computer Society, August 2008

[ISO 26262:2018] ISO 26262:2018 Road vehicles — Functional Safety, ISO, December 2018

[ISO-IEC-33020] ISO/IEC 33020:2019 Information technology – Process assessment – Process measurement framework for assessment of process capability, ISO, November 2019

[ISO-IEC-9899] ISO/IEC 9899:2018 Information Technology – Programming Languages – C, ISO/IEC, June 2018

[KCG-Report to the Certificate] Report to the Certificate Z10 16 11 55460 008, Code Generator SCADE Suite KCG 6.6, TÜV SÜD, December 2016

[LR-Report to the Certificate] Report to the Certificate Z10 055460 0016 Rev. 00, SCADE LifeCycle Reporter for SCADE Suite, TÜV SÜD, April 2020

[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

[MCOV-FAQ11] Model Coverage for SCADE Suite – FAQ#11: Application Conditions and Property, MC-SRS-004, Rev D, Incr.1, Ansys, January 2019

[MCOV-FAQ11-Ext] Extended FAQ#11 Applications Conditions with SCADE Suite KCG 6.6 and MC 6.7, MC 2019, MC 2020, Engineering Note SAF-EN-115_FAQ_11, March 2021

[MCOV-Report to the Certificate] Report to the Certificate Z10 055460 0014 Rev. 00, Model Coverage for SCADE Suite, TÜV SÜD, April 2020

[MISRA C:2012] MISRA C:2012, Guidelines for the use of the C language in Critical Systems, Misra Limited, March 2013

[MISRA C:2012/AMD1] MISRA C:2012, Amendment 1, Additional Security Guidelines for MISRA C:2012, Horiba Mira, April 2016

[NASA-MCDC] "A Practical Tutorial on Modified Condition/Decision Coverage", K. J. Hayhurst (NASA), D. Veerhusen (Rockwell Collins), J. J. Chilenski (Boeing), L. K. Rierson (FAA), NASA, May 2001

[Scade 6] Scade 6: A Formal Language for Embedded Critical Software Development. J-L. Colaco, B. Pagano, and M. Pouzet, in Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE), September 2017

[SCS-ACG-ISO26262-TCL3-COMPL]: SCADE Automotive Code Generator for AUTOSAR – V2.1 – ISO 26262 Compliance Analysis document, SC-TR-559, Ansys, May 2021

[SCS-ACG-RN]: SCADE Automotive Code Generator for AUTOSAR v2.1 – Release Note, SC-TR-588, Ansys, May 2021.

[SCS-ACG-TCI]: SCADE Automotive Code Generator for AUTOSAR v2.1 – Tool Configuration Index, SC-TR-222, Ansys, May 2021

[SCS-ACG-TOR]: SCADE Automotive Code Generator for AUTOSAR – Tool Operational Requirements, SC-SRS-161, Ansys, March 2021

[SCS-ACG-Report to the Certificate]: Report to the Certificate Z10 055460 0020 Rev. 00, SCADE Automotive Code Generator for AUTOSAR ACG, TÜV SÜD, May 2021

[SCS-ACG-Safety Analysis] Safety Analysis of SCADE Suite 2021 R2 AUTOSAR Features, SAF-EN-003, Ansys, May 2021

[SCS-KCG-DO330-TQL1-COMPL] SCADE Suite KCG 6.6.2 – Compliance analysis with DO-330 level 1 and 2, KCG-TR-144, Ansys, June 2021

[SCS-KCG-EN50128-SIL3/4-COMPL] SCADE Suite KCG 6.6.2 – Compliance Analysis with EN 50128 SIL3-4, KCG-TR-140, Ansys, June 2021

[SCS-KCG-IEC61508-SIL3-COMPL] SCADE Suite KCG 6.6.2 – Compliance Analysis with IEC 61508 SIL 3, KCG-TR-138, Ansys, June 2021

[SCS-KCG-LRM] The Scade 6 Language, KCG-SRS-007, Ansys, March 2016

[SCS-KCG-MISRA-C-COMPL] SCADE Suite KCG 6.6.2 – KCG generated C code MISRA Compliance Report, KCG-TR-136, Ansys, June 2021

[SCS-KCG-Safety Case] SCADE Suite KCG 6.6.2 – Safety Case, KCG-TR-137, Ansys, June 2021

[SCS-KCG-SIP] SCADE Suite KCG 6.6 – Software Installation Procedure, KCG-SP-017, Ansys, March 2021

[SCS-KCG-TAS] SCADE Suite KCG 6.6.2 – Tool Accomplishment Summary, KCG-TR-143, Ansys, June 2021

[SCS-KCG-TCI] SCADE Suite KCG 6.6.2 – Tool Configuration Index, KCG-SCI-019, Ansys, June 2021

[SCS-KCG-TECI] SCADE Suite KCG 6.6.2 – Tool Life Cycle Environment Configuration Index, KCG-SCI-020, Ansys, June 2021

[SCS-KCG-TOR] SCADE Suite KCG Tool Operational Requirements, KCG-SRS-011, Ansys, March 2016

[SCS-KCG-TQP] SCADE Suite KCG 6.6 – Tool Qualification Plan, KCG-PL-028, Ansys, February 2015

[SCS-MCH-COMPL] SCADE LifeCycle Model Change for SCADE Suite 2021 R2 – ISO 26262 Compliance Analysis document, SC-TR-531, Ansys, June 2021

[SCS-MCOV-COMPL] Model Coverage for SCADE Suite 2020 – Compliance Analysis with ISO 26262 TCL3 Classified Tool, MC-TR-022, Ansys, November 2020

[SCS-MR-COMPL] Model Reporter for SCADE Suite 2020 – Compliance Analysis with ISO 26262, Ansys, November 2020

[SCS-SDVST] SCADE Suite Development Standards, Ansys, September 2017

[SCS-STE-COMPL] SCADE Test Environment 2020 R2 – Compliance Analysis with ISO 26262 ASIL C-D, SC-TR-485, November 2020

[SPICE] ISO/IEC 15504 – Software Process Improvement and Capability Determination, SPICE, ISO/IEC

[Statecharts] “Statecharts for Unified Model-Based Design – As simple as possible, as rich as needed”, J. L. Dufour, ERTS, January 2018

[TE-Report to the Certificate] Report to the Certificate Z10 055460 0015 Rev. 00, SCADE Test Environment, TÜV SÜD, April 2020

APPENDIX C
COMPLIANCE MATRIX OF SCADE WITH ISO 26262-6:2018

This Appendix provides a description of the compliance of the SCADE toolchain and methodology with the requirements and recommendations of [ISO 262262-6:2018], as they appear in Clauses 5 to 11 of the standard, including Tables 1 to 15. For the sake of readability, we only reproduced in these Tables (1 to 15) the ASIL D recommendations for these requirements.

For each requirement and recommendation in the standard, the Level of Support of the SCADE toolchain is rated as follows:

- **Automate**: tool automates the activity
- **Reduce**: activity still must be performed by user, but tool supports it
- **No Support**

Note: In the table below, "Automate" from a tool describes the support from the tool's perspective. It does not mean that the tool user has no activity to perform. For example, SCADE Suite KCG achieves automatic source code generation from a design model. However, the KCG user must check the KCG log file to ensure that code generation completed normally, as specified in the conditions of use of the tool.

C.1 General topics for the product development at the software level (Clause 5)

C.1.1 Requirements regarding the software development processes and environment

TABLE 30: COMPLIANCE WITH REQUIREMENTS REGARDING THE SOFTWARE DEVELOPMENT ENVIRONMENT AND PROCESSES

Source: Extract from Clause 5.4.1 in ISO 26262-6:2018

Requirements		Level of Support	SCADE toolchain and methodology compliance
5.4.1	When developing the software of an item, software development processes and software development environments shall be used which:		
a	are suitable for developing safety-related embedded software, including methods, guidelines, languages, and tools	Reduce	The SCADE language, toolchain and methodology have been created specifically for developing safety-related embedded software
b	support consistency across the sub-phases of the software development lifecycle and the respective work products	Reduce	SCADE is an integrated toolchain covering the full product development cycle at the software level
c	support consistency of exchange of information	Reduce	The SCADE data formats support consistency of exchange of information throughout the full product development cycle at the software level

C.1.2 Requirements for selecting a design, modeling, or programming language

TABLE 31: COMPLIANCE WITH REQUIREMENTS REGARDING THE SOFTWARE DEVELOPMENT ENVIRONMENT AND PROCESSES

Source: Extract from Clause 5.4.2 in ISO 26262-6:2018

Requirements		Level of Support	SCADE toolchain and methodology compliance
5.4.2	The criteria that shall be considered when selecting a design, modeling or programming language are:		
a	an unambiguous and comprehensible definition	Automate	The Scade language is unambiguous and comprehensible
b	suitability for specifying and managing safety requirements according to ISO26262-8:2018 if modelling is used for requirements engineering and management	No Support	
c	support the achievement of modularity, abstraction and encapsulation	Reduce	The Scade language is modular; it fosters abstraction and encapsulation
d	support the use of structured constructs	Automate	The Scade language is structured

C.1.3 Requirements regarding modeling and coding guidelines

TABLE 32: COMPLIANCE WITH TOPICS TO BE COVERED BY MODELING AND CODING GUIDELINES

Source: Extract from Table 1 in ISO 26262-6:2018

Topics		ASIL D	Level of Support	SCADE Suite compliance
1a	Enforcement of low complexity	++	Reduce	SCADE Suite Rules Checker provides scripting capabilities to check Scade models complexity
1b	Use of language subsets	++	Automate	Scade is a domain specific language for the development of safety-related applications; it does not need to be restricted by coding rules. SCADE Suite KCG generates a small and safe subset of the C language that is MISRA C:2012 compliant
1c	Enforcement of strong typing	++	Automate	Scade is a strongly typed language
1d	Use of defensive implementation techniques	++	Reduce	SCADE Suite promotes the development of robust libraries to implement a defensive programming strategy at model-level
1e	Use of well-trusted design principles	++	Automate	The Scade language is domain specific language for the development of safety-related applications, and it is based on well-trusted design principles such as modularity, composability, hierarchy, and concurrency
1f	Use of unambiguous graphical representation	++	Automate	SCADE Suite provides an unambiguous graphical notation
1g	Use of style guides	++	Reduce	SCADE Suite Rules Checker provides scripting capabilities to enforce user-specific modeling style guides. Modeling guidelines are proposed in [SCS-SDVST]
1h	Use of naming conventions	++	Reduce	SCADE Suite Rules Checker provides scripting capabilities to enforce user-specific naming conventions
1i	Concurrency aspects	+	Automate	The Scade language is a concurrent language

C.2 Specification of software safety requirements (Clause 6)

TABLE 33: COMPLIANCE WITH REQUIREMENTS REGARDING THE SOFTWARE SAFETY REQUIREMENTS

Source: Extract from Clause 6.4.1/2/3/4/6/6/7 in ISO 26262-6:2018

	Requirements	Level of Support	SCADE toolchain compliance
6.4.1	The software safety requirements are either derived directly from the technical safety requirements allocated to software or are requirements for software functions and properties that, if not fulfilled, could lead to a violation of the technical safety requirements allocated to software	No Support	
6.4.2	Specification of the software safety requirements derived from the technical safety requirements, the technical safety concept and the system architectural design ... shall consider: a) the specification and management of safety requirements in accordance with ISO 26262-8:2018, Clause 6; b) the specified system and hardware configurations; c) the hardware-software interface specification; d) the relevant requirements of the hardware design specification; e) the timing constraints; f) the external interfaces; and g) each operating mode and each transition between the operating modes of the vehicle, the system, or the hardware, having an impact on the software	No Support	
6.4.3	If ASIL decomposition is applied to the software safety requirements, ISO 26262-9:2018, Clause 5, shall be complied with	No Support	
6.4.4	The hardware-software interface specification initiated in ISO 26262-4:2018, Clause 6, shall be refined sufficiently to allow for the correct control and usage of the hardware by the software, and shall describe each safety-related dependency between hardware and software	No Support	
6.4.5	If other functions in addition to those functions for which safety requirements are specified in 6.4.1 are carried out by the embedded software, a specification of these functions and their properties in accordance with the applied quality management system shall be available	No Support	
6.4.6	The refined hardware-software interface specification shall be verified jointly by the persons responsible for the system, hardware and software development	No Support	
6.4.7	The software safety requirements and the refined requirements of the hardware-software interface specification shall be verified in accordance with ISO 26262-8:2018, Clauses 6 and 9, to provide evidence for their: a) suitability for software development; b) compliance and consistency with the technical safety requirements; c) compliance with the system design; and d) consistency with the hardware-software interface	No Support	

C.3 Software architectural design (Clause 7)

C.3.1 Requirements regarding the notation for software architectural design

TABLE 34: COMPLIANCE WITH REQUIREMENT REGARDING THE NOTATION FOR SOFTWARE ARCHITECTURAL DESIGN

Source: Extract from Clause 7.4.1 in ISO 26262-6:2018

Requirement		Level of Support	SCADE Architect and SCADE Suite compliance
7.4.1	To avoid systematic faults in the software architectural design and in the subsequent development activities, the description of the software architectural design shall address the following characteristics supported by notations for software architectural design: a) comprehensibility b) consistency c) simplicity d) verifiability e) modularity f) abstraction g) encapsulation h) maintainability	Reduce	Both the SysML notation of SCADE Architect and the Scade language of SCADE Suite support the description of software architectural design through the following characteristics: a) comprehensibility; b) consistency; c) simplicity; d) verifiability; e) modularity; f) abstraction; g) encapsulation; and h) maintainability

TABLE 35: COMPLIANCE WITH NOTATION FOR SOFTWARE ARCHITECTURAL DESIGN

Source: Extract from Table 2 in ISO 26262-6:2018

Notations		ASIL D	Level of Support	SCADE Architect and SCADE Suite compliance
1a	Natural language	++	No Support	
1b	Informal Notations	+	No Support	
1c	Semi-formal notations	++	Automate	SCADE Architect is based on the SysML semi-formal notation
1d	Formal notations	+	Automate	SCADE Suite is based on the Scade language which is a formal notation

C.3.2 Requirements regarding the principles for software architectural design

TABLE 36: COMPLIANCE WITH REQUIREMENTS REGARDING THE PRINCIPLES FOR SOFTWARE ARCHITECTURAL DESIGN

Source: Extract from Clause 7.4.2/3 in ISO 26262-6:2018

Requirement		Level of Support	SCADE Architect, SCADE Suite, SCADE Test, and SCADE LifeCycle compliance
7.4.2	During the development of the software architectural design, the following shall be considered:		
a	verifiability of the software architectural design	Reduce	Software architectural design described in a combination of SCADE Architect and SCADE Suite eases verifiability
b	suitability for configurable software	Reduce	Both SCADE Architect and SCADE Suite are connected to Configuration Management tools through the SCADE LifeCycle ALM Gateway
c	feasibility for the design and implementation of the software units	Reduce	The SCADE Architect to SCADE Suite synchronization facilitates the assessment of the feasibility for the design and implementation of the software units
d	testability of the software architecture during software integration testing	Reduce	The Scade language, coupled with the capabilities of SCADE Test, foster the creation of a testable software architecture during software integration testing
e	maintainability of the software architectural design	Reduce	The notations of SCADE Architect and SCADE Suite foster the creation of maintainable software architectural design
7.4.3	In order to avoid systematic faults, the software architectural design shall exhibit the following characteristics by use of the principles: a) comprehensibility b) consistency c) simplicity d) verifiability e) modularity f) encapsulation g) maintainability	Reduce	The combination of SCADE Architect and SCADE Suite fosters the creation of software architectural design that exhibits: a) comprehensibility b) consistency c) simplicity d) verifiability e) modularity f) encapsulation g) maintainability

TABLE 37: COMPLIANCE WITH PRINCIPLES FOR SOFTWARE ARCHITECTURAL DESIGN

Source: Extract from Table 3 in ISO 26262-6:2018

Principles		ASIL D	Level of Support	SCADE Architect and SCADE Suite compliance
1a	Appropriate hierarchical structure of the software components	++	Automate	Both SysML and Scade are modular languages with a hierarchical structure
1b	Restricted size and complexity of software components	++	Reduce	Dedicated rules can be established by the SCADE Architect and SCADE Suite users
1c	Restricted size of interfaces	++	Reduce	Dedicated rule can be established by the SCADE Architect and SCADE Suite users
1d	Strong cohesion within each software component	++	Reduce	Enforced by modularity and hierarchy
1e	Loose coupling between software components	++	Reduce	Enforced by modularity and hierarchy
1f	Appropriate scheduling properties	++	Automate	The Scade language ensures explicit and deterministic activation within the SCADE model
1g	Restricted use of interrupts	++	Automate	Use of interrupts is restricted to the outside of SCADE-generated code. The Scade language does not allow the use of interrupts
1h	Appropriate spatial isolation of the software components	++	Out of scope	
1i	Appropriate management of shared resources	++	Out of scope	

C.3.3 Requirements regarding the scope of the software architectural design

TABLE 38: COMPLIANCE WITH REQUIREMENTS REGARDING THE SCOPE OF THE SOFTWARE ARCHITECTURAL DESIGN

Source: Extract from Clause 7.4.4 to 7.4.13 in ISO 26262-6:2018

Requirements		Level of Support	SCADE Architect, SCADE Suite, and SCADE LifeCycle compliance
7.4.4	The software architectural design shall be developed down to the level where the software units are identified	Reduce	Synchronization between SCADE Architect and SCADE Suite ensures that a level of architectural description has been reached such that the software units can be designed in SCADE Suite
7.4.5	The software architectural design shall describe:		
a	the static design aspects of the software architectural elements	Reduce	Both SCADE Architect and SCADE Suite support the static design aspects of the software architecture elements
b	the dynamic design aspects of the software architectural elements	Reduce	Both SCADE Architect and SCADE Suite support the dynamic design aspects of the software architecture elements
7.4.6	The software safety requirements shall be hierarchically allocated to the software components down to software units. As a result, each software component shall be developed in compliance with the highest ASIL of any of the requirements allocated to it	Reduce	The SCADE LifeCycle ALM Gateway supports the allocation of the software requirements to the architectural elements, down to the software units level
7.4.7	If a pre-existing software architectural element is used without modifications in order to meet the assigned safety requirements without being developed according to the ISO 26262 series of standards, then it shall be qualified in accordance with ISO 26262-8:2018, Clause 12.	No Support	
7.4.8	If the embedded software has to implement software components of different ASILs, or safety-related and non-safety-related software components, then all of the embedded software shall be treated in accordance with the highest ASIL, unless the software components meet the criteria for coexistence in accordance with ISO 26262-9:2018, Clause 6	No Support	

Requirements		Level of Support	SCADE Architect, SCADE Suite, and SCADE LifeCycle compliance
7.4.9	If software partitioning (see Annex D) is used to implement freedom from interference between software components it shall be ensured that: a) the shared resources are used in such a way that freedom from interference of software partitions is ensured b) the software partitioning is supported by dedicated hardware features or equivalent means (this requirement applies to ASIL D) c) the element of the software that implements the software partitioning is developed in compliance with the highest ASIL assigned to any requirement of the software partitions d) evidence for the effectiveness of the software partitioning is generated during software integration and verification	No Support	
7.4.10	Safety-oriented analysis shall be carried out at the software architectural level in accordance with ISO 26262-9:2018, Clause 8, in order to: – provide evidence for the suitability of the software to provide the specified safety-related functions and properties as required by the respective ASIL – identify or confirm the safety-related parts of the software – support the specification and verify the effectiveness of the safety measures.	Reduce	The combination of medini and SCADE Architect facilitate this analysis
7.4.11	If the implementation of software safety requirements relies on freedom from interference or sufficient independence between software components, dependent failures and their effects shall be analysed in accordance with ISO 26262-9:2018, Clause 7.	No Support	
7.4.12	Depending on the results of the safety-oriented analyses at the software architectural level, safety mechanisms for error detection and error handling shall be applied.	No Support	
7.4.13	An upper estimation of required resources for the embedded software shall be made, including: a) the execution time b) the storage space c) the communication resources	No Support	

C.3.4 Requirements for the verification of the software architectural design

TABLE 39: COMPLIANCE WITH REQUIREMENT FOR THE VERIFICATION OF THE SOFTWARE ARCHITECTURAL DESIGN

Source: Extract from Clause 7.4.14 in ISO 26262-6:2018

Requirement		Level of Support	SCADE Architect, SCADE Suite, and SCADE LifeCycle compliance
7.4.14	The software architectural design shall be verified in accordance with ISO 26262-8:2018, Clause 9 and by using the software architectural design verification methods listed in Table 4 [of ISO 26262-6] to provide evidence that the following objectives are achieved:		
a	the software architectural design is suitable to satisfy the software requirements with the required ASIL	No Support	
b	the review or investigation of the software architectural design provides evidence for the suitability of the design to satisfy the software requirements with the required ASIL	Reduce	SCADE LifeCycle Reporter automatically produces the software architectural design document to be reviewed
c	compatibility with the target environment	Reduce	SCADE Suite TSO and TSV, and SCADE Suite CVK support the assessment of compatibility of the software architectural design with the target environment
d	adherence to design guidelines	Reduce	In addition to the verifications provided by SCADE Architect and SCADE Suite modeling rules, the user can create specific rules to implement further design guidelines with SCADE Architect and SCADE Suite Rule Checker

TABLE 40: COMPLIANCE WITH METHODS FOR THE VERIFICATION OF THE SOFTWARE ARCHITECTURAL DESIGN

Source: Extract from Table 4 in ISO 226262-6:2018

Methods		ASIL D	Level of Support	SCADE Architect and SCADE Suite compliance
1a	Walk-through of the design	o	Reduce	SCADE LifeCycle Reporter supports the activity
1b	Inspection of the design	++	Reduce	SCADE LifeCycle Reporter supports the activity
1c	Simulation of dynamic behaviour of the design	++	No Support	
1d	Prototype generation	++	No Support	
1e	Formal verification	+	No Support	
1f	Control flow analysis	++	Automate	SCADE Suite Semantics Checker performs a static analysis of control and data flows
1g	Data flow analysis	++		
1h	Scheduling analysis	++	Automate	For the part of the architecture that is described in SCADE Suite, schedulability is guaranteed by SCADE Suite Semantic Checker

C.4 Software unit design and implementation (Clause 8)

C.4.1 Generic requirements for software unit design and implementation

TABLE 41: COMPLIANCE WITH GENERIC REQUIREMENTS FOR SOFTWARE UNIT DESIGN AND IMPLEMENTATION

Source: Extract from Clause 8.4.2 in ISO 26262-6:2018

Requirements		Level of Support	SCADE Suite compliance
8.4.2	The software unit design and implementation shall be:		
a	suitable to satisfy the software requirements allocated to the software unit with the required ASIL	Reduce	SCADE Suite is based on a language that is specific to the design and implementation of safety-related applications at the highest ASIL and it provides a toolchain to perform all the expected activities regarding development, integration, and verification
b	consistent with the software architectural design specification	Reduce	Software unit design in SCADE Suite is synchronized with software architectural design in SCADE Architect
c	consistent with the hardware-software interface specification, if applicable	Reduce	SCADE Suite provides tools to assess compatibility between unit designs and the target architecture (SCADE Suite TSO and TSV)

C.4.2 Requirements for the software units design notation

TABLE 42: COMPLIANCE WITH NOTATION FOR SOFTWARE UNIT DESIGN

Source: Extract from Table 5 in ISO 26262-6:2018

Notations		ASIL D	Level of Support	SCADE Suite compliance
1a	Natural language	++	No Support	
1b	Informal notations	+	No Support	
1c	Semi-formal notations	++	No Support	
1d	Formal notations	+	Automate	SCADE Suite is based on the Scade language, which is a formal notation, thus ensuring: a) consistency b) comprehensibility c) maintainability d) verifiability

C.4.3 Requirements for software unit design and implementation principles

TABLE 43: COMPLIANCE WITH PROPERTIES OF SOFTWARE UNIT DESIGNS

Source: Extract from Clause 8.4.5 in ISO 26262-6:2018

Requirements		Level of Support	SCADE Suite compliance
8.4.5	Design principles for software unit design and implementation at the source code level shall be applied to achieve the following properties:		
a	correct order of execution of subprograms and functions within the software units, based on the software architectural design	Automate	SCADE Suite guarantees correct execution order
b	consistency of the interfaces between the software units	Automate	SCADE Suite Semantics Checker performs verification of the consistency of the interfaces between the software units
c	correctness of data flow and control flow between and within the software units	Automate	SCADE Suite Semantics Checker performs a static analysis of control and data flows between and within the software units
d	simplicity	Reduce	The Scade language fosters the creation of simple designs
e	readability and comprehensibility	Reduce	The Scade language fosters the creation of readable and comprehensible designs
f	robustness	Reduce	The Scade language precludes the use of error prone constructs; the SCADE Suite design and verification methodology promotes ways to produce robust designs; SCADE Suite Design Verifier allows to perform verification of design robustness
g	suitability for software modification	Reduce	Modularity of the Scade language makes it suitable for software modifications; SCADE Suite Model Diff and SCADE LifeCycle Model Change allow to better assess and manage software modifications
h	verifiability	Automate	The formal definition of the Scade language makes software units designs verifiable

TABLE 44: COMPLIANCE WITH PRINCIPLES FOR SOFTWARE UNIT DESIGN AND IMPLEMENTATION

Source: Extract from Table 6 in ISO 26262-6:2018

Principles		ASIL D	Level of Support	SCADE Suite compliance
1a	One entry and one exit point in subprograms and functions	++	Automate	A Scade operator has exactly one entry and one exit point
1b	No dynamic objects or variables, or else online test during their creation	++	Automate	There is no dynamic creation of objects or variables in the Scade language
1c	Initialization of variables	++	Automate	Every flow in a Scade model is checked for correct initialization by the SCADE Suite Semantics Checker
1d	No multiple use of variable names	++	Automate	Scade variable names are checked to be unique in their scope. Multiple definitions of variables are forbidden at language level
1e	Avoid global variables or else justify their usage	++	Automate	There are no global variables in Scade, except for Sensors which are read-only variables
1f	Restricted use of pointers	++	Automate	There are no pointers in the Scade language
1g	No implicit type conversions	++	Automate	Scade is a strongly typed language allowing only explicit type conversions
1h	No hidden data flow or control flow	++	Automate	The Scade language describes all data and control flows
1i	No unconditional jumps	++	Automate	There are no unconditional jumps in the Scade language
1j	No recursions	++	Automate	The Scade language does not allow recursion

C.5 Software unit verification (Clause 9)

C.5.1 Generic requirements for software unit verification

TABLE 45: COMPLIANCE WITH GENERIC REQUIREMENTS FOR SOFTWARE UNIT VERIFICATION

Source: Extract from Clause 9.4.2 in ISO 26262-6:2018

Requirements		Level of Support	SCADE LifeCycle, SCADE Suite, and SCADE Test compliance
9.4.2	The software unit design and the implemented software unit shall be verified by applying an appropriate combination of methods to provide evidence for:		
a	compliance with the requirements regarding the unit design and implementation	Reduce	SCADE LifeCycle Reporter and SCADE LifeCycle Model Change support (incremental) reviews of Scade models; SCADE Test facilitates creating and running requirements-based test cases on host and target
b	the compliance of the source code with its design specification	Automate	SCADE Suite KCG (and SCADE ACG) have been qualified at TCL3
c	compliance with the specification of the hardware-software interface	Reduce	SCADE Test Target Execution facilitates re-running requirements-based test cases on target, thus checking compliance of the software unit design with hardware-software interfaces
d	confidence in the absence of unintended functionality and properties	Reduce	SCADE Test Model Coverage detects unintended functionality at model-level. Coverage at model-level implies coverage at code level, *w/o* the need for verifying this at the coding phase
e	sufficient resources to support their functionality and properties	Reduce	SCADE Suite TSO and TSV support the evaluation of the resources needed to implement the software units
f	implementation of the safety measures resulting from the safety-oriented analyses	Reduce	Safety-oriented analyses may uncover new software safety requirements, which will be implemented and verified in the way described in this document

C.5.2 Requirements regarding methods for software unit verification

TABLE 46: COMPLIANCE WITH METHODS FOR SOFTWARE UNIT VERIFICATION

Source: Extract from Table 7 in ISO 26262-6:2018

Methods		ASIL D	Level of Support	SCADE Suite, SCADE Test Environment for Host, and SCADE LifeCycle compliance
1a	Walk-through	o	Reduce	SCADE LifeCycle Reporter and SCADE LifeCycle Model Change support (incremental) reviews of Scade models
1b	Pair-programming	+		
1c	Inspection	++	Reduce	SCADE LifeCycle Reporter and SCADE LifeCycle Model Change support (incremental) reviews of Scade models
1d	Semi-formal verification	++	Automate	SCADE Suite Semantics Checker formally verifies static properties (*e.g.*, proper initialization). SCADE Suite Design Verifier formally verifies safety properties based on proof objectives provided by the users
1e	Formal verification	+	Automate	
1f	Control flow analysis	++	Automate	SCADE Suite Semantics Checker performs a static analysis of control and data flows
1g	Data flow analysis	++	Automate	
1h	Static code analysis	++	Automate	SCADE Suite Rule Checker allows to verify modeling guidelines. SCADE Suite KCG guarantees that the generated code complies to MISRA-C:2012/AMD1
1i	Static analyses based on abstract interpretation	+	Automate	SCADE Suite KCG performs analyses based on abstract interpretation to generate correct and efficient source code
1j	Requirements-based test	++	Reduce	SCADE Test Environment for Host supports requirements-based test of Scade models and produces a qualified conformance test report. SCADE Test Target Execution supports automated re-use of those test on the target platform
1k	Interface test	++	Automate	SCADE Test Model Coverage ensures proper coverage of interfaces
1l	Fault injection test	++		
1m	Resource usage evaluation	++	Reduce	The SCADE Suite KCG generated code properties facilitate memory footprint evaluation. The worst-case execution time (WCET) and worst-case stack usage can be evaluated by SCADE Suite Time and Stack Verifier or by conventional means
1n	Back-to-back comparison test between model and code, if applicable	++	Automate	Compliance of the generated code to the model is ensured by SCADE Suite KCG qualification. Compliance of the model to the software requirements is verified at model-level with SCADE Test. Back-to-back testing on host is eliminated

TABLE 47: COMPLIANCE WITH METHODS FOR DERIVING TEST CASES FOR SOFTWARE UNIT TESTING

Source: Extract from Table 8 in ISO-26262-6:2018

Methods		ASIL D	Level of Support	SCADE Test Environment for Host compliance
1a	Analysis of requirements	++	Reduce	Test cases creation is under user responsibility, and they will be created and managed in SCADE Test Environment for Host at software unit level
1b	Generation and analysis of equivalence classes	++	Reduce	
1c	Analysis of boundary values	++	Reduce	
1d	Error guessing based on knowledge or experience	+	Reduce	

C.5.3 Requirements for structural coverage metrics at the software unit level

TABLE 48: COMPLIANCE WITH STRUCTURAL COVERAGE METRICS AT THE SOFTWARE UNIT LEVEL

Source: Extract from Table 9 in ISO 26262-6:2018

Methods		ASIL D	Level of Support	SCADE Test Model Coverage and SCADE Suite KCG compliance
1a	Statement coverage	+	Automate	SCADE Test Model Coverage performs coverage analysis at model-level and guarantees that coverage at model level implies code coverage at the proper level (statement coverage, branch coverage, and MC/DC), when SCADE Suite KCG is used to generate the source code
1b	Branch coverage	++	Automate	
1c	MC/DC (Modified Condition/ Decision Coverage)	++	Automate	

C.5.4 Requirements for the test environment for software unit testing

TABLE 49: COMPLIANCE WITH REQUIREMENTS FOR THE TEST ENVIRONMENT FOR SOFTWARE UNIT TESTING

Source: Extract from Clause 9.4.5 in ISO 26262-6:2018

Requirements		Level of Support	SCADE Test compliance
9.4.5	The test environment for software unit testing shall be suitable for achieving the objectives of the unit testing considering the target environment. If the software unit testing is not carried out in the target environment, the differences in the source and object code, as well as the differences between the test environment and the target environment, shall be analyzed in order to specify additional tests in the target environment during the subsequent test phases.	Reduce	SCADE Test Target Execution automates the creation of test harnesses for re-running the requirements-based test cases that have originally been created for testing on host with SCADE Test Environment for Host

C.6 Software integration and verification (Clause 10)

C.6.1 Generic requirements for software integration and verification

TABLE 50: COMPLIANCE WITH GENERIC REQUIREMENTS FOR SOFTWARE INTEGRATION AND VERIFICATION

Source: Extract from Clause 10.4.2 in ISO 26262-6:2018

Requirements		Level of Support	SCADE Architect, SCADE Suite, and SCADE Test compliance
10.4.2	The software integration shall be verified ... to provide evidence that the hierarchically integrated software units, the software components and the integrated embedded software achieve:		
a	compliance with the software architectural design	Reduce	SCADE Test Target Execution automates the creation of test harnesses for re-running the requirements-based test cases that have originally been created for testing on host with SCADE Test Environment for Host
b	compliance with the hardware-software interface specification	No Support	
c	the specified functionality	Reduce	SCADE Test facilitates creating and running requirements-based test cases on host and target to verify that the integrated embedded software provides the specified functionality; SCADE Test Model Coverage detects unintended functionality at model-level. Coverage at model-level implies coverage at code level, *w/o* the need for verifying this at the coding phase
d	the specified properties EXAMPLE Reliability due to absence of inaccessible software, robustness against erroneous inputs, dependability due to effective error detection and handling	Reduce	SCADE Test facilitates creating robustness test cases on host and target to verify that the integrated embedded software provides the specified functionality; SCADE Test Model Coverage detects inaccessible software at model-level
e	sufficient resources to support the functionality	Reduce	SCADE Suite TSO and TSV support the evaluation of the resources needed to implement the software units
f	effectiveness of the safety measures resulting from the safety-oriented analysis	Reduce	Safety-oriented analyses of SCADE Architect and SCADE Suite models may uncover new software safety requirements, which will be implemented and verified in the way described in this document

C.6.2 Requirements regarding methods for verification of software integration

TABLE 51: COMPLIANCE WITH METHODS FOR VERIFICATION OF SOFTWARE INTEGRATION

Source: Extract from Table 10 in ISO 26262-6:2018

Methods		ASIL D	Level of Support	SCADE Suite and SCADE Test compliance
1a	Requirements-based test	++	Reduce	SCADE Test Environment for Host supports requirements-based test of Scade models and produces a qualified conformance test report. SCADE Test Target Execution supports automated re-use of those tests on the target platform
1b	Interface test	++	Automate	SCADE Test Model Coverage ensures proper coverage of interfaces
1c	Fault injection test	++		
1d	Resource usage evaluation	++	Reduce	The SCADE Suite KCG generated code properties facilitate memory footprint evaluation. The worst-case execution time (WCET) and worst-case stack usage can be evaluated by SCADE Suite Time and Stack Verifier or by conventional means
1e	Back-to-back comparison test between model and code, if applicable	++	Automate	Compliance of the generated code to the model is ensured by SCADE Suite KCG qualification. Compliance of the model to the software requirements is verified at model-level with SCADE Test. Back-to-back testing on host is eliminated
1f	Verification of the control flow and data flow	++	Automate	SCADE Suite Semantics Checker performs verification of the control and data flows
1g	Static code analysis	++	Automate	SCADE Suite Rule Checker allows to verify modeling guidelines. SCADE Suite KCG guarantees that the generated code complies to MISRA-C:2012/AMD1
1h	Static analyses based on abstract interpretation	+	Automated	SCADE Suite KCG performs analyses based on abstract interpretation to generate correct and efficient source code

TABLE 52: COMPLIANCE WITH METHODS FOR DERIVING TEST CASES FOR SOFTWARE INTEGRATION TESTING

Source: Extract from Table 11 in ISO 26262-6:2018

Methods		ASIL D	Level of Support	SCADE Test compliance
1a	Analysis of requirements	++	No Support	Test cases creation is under user responsibility, and they must be written in SCADE Test Environment for Host at each step of integration
1b	Generation and analysis of equivalence classes	++	No Support	
1c	Analysis of boundary values	++	No Support	
1d	Error guessing based on knowledge or experience	+	No Support	

C.6.3 Requirements regarding methods for structural coverage at the software architectural level

TABLE 53: COMPLIANCE WITH STRUCTURAL COVERAGE AT THE SOFTWARE ARCHITECTURE LEVEL

Source: Extract from Table 12 in ISO 26262-6:2018

Methods		ASIL D	Level of Support	SCADE Test Model Coverage compliance
1a	Function coverage	++	Automate	SCADE Test Model Coverage measures these metrics
1b	Call coverage	++	Automate	

C.6.4 Requirement regarding unspecified functions as part of the embedded software

TABLE 54: COMPLIANCE WITH REQUIREMENT REGARDING UNSPECIFIED FUNCTIONS AS PART OF THE EMBEDDED SOFTWARE

Source: Extract from Clause 10.4.6 in ISO 26262-6:2018

Requirement		Level of Support	SCADE Test compliance
10.4.6	It shall be verified that the embedded software that is to be included as part of a production release … contains all the specified functions and properties and only contains other unspecified functions if these functions do not impair the compliance with the software safety requirements	Reduce	SCADE Test facilitates creating and running requirements-based test cases on host and target. SCADE Test Model coverage will uncover unspecified functions

C.6.5 Requirements regarding the test environment for software integration testing

TABLE 55: COMPLIANCE WITH REQUIREMENT REGARDING THE TEST ENVIRONMENT FOR SOFTWARE INTEGRATION TESTING

Source: Extract from Clause 10.4.7 in ISO 26262-6:2018

Requirement		Level of Support	SCADE Test compliance
10.4.7	The test environment for software integration testing shall be suitable for achieving the objectives of the integration testing considering the target environment. If the software integration testing is not carried out in the target environment, the differences in the source and object code and the differences between the test environment and the target environment shall be analyzed in order to specify additional tests in the target environment during the subsequent test phases	Reduce	SCADE Test Target Execution automates the creation of test harnesses for re-running the requirements-based test cases that have originally been created for testing on host with SCADE Test Environment for Host, thus facilitating the integration testing phase

C.7 Testing of the embedded software (Clause 11)

TABLE 56: COMPLIANCE WITH TEST ENVIRONMENTS FOR CONDUCTING THE SOFTWARE TESTING

Source: Extract from Table 13 in ISO 26262-6:2018

Methods		ASIL D	Level of Support	SCADE Test compliance
1a	Hardware-in-the-loop	++	Reduce	SCADE Test Target Execution supports HiL testing when connected to HiL environments. Scenarios that have been created using SCADE Test Environment for Host can be replayed in HiL testing.
1b	Electronic control unit network environments	++	Reduce	Scenarios that have been created using SCADE Test Environment for Host can be replayed in HiL testing.
1c	Vehicles	++	Reduce	Scenarios that have been created using SCADE Test Environment for Host can be replayed in HiL testing.

TABLE 57: COMPLIANCE WITH METHODS FOR TESTS OF THE EMBEDDED SOFTWARE

Source: Extract from Table 14 in ISO 26262-6:2018

Methods		ASIL D	Level of Support	SCADE Test compliance
1a	Requirements-based test	++	Reduce	SCADE Test reduces the effort that is needed to test the embedded software, as requirements-based tests created for Model-in-the-Loop testing can be reused in this sub-phase.
1b	Fault injection test	++	No Support	

TABLE 58: COMPLIANCE WITH METHODS FOR DERIVING TEST CASES FOR THE TEST OF THE EMBEDDED SOFTWARE

Source: Extract from Table 15 in ISO 26262-6:2018

Methods		ASIL D	Level of Support	SCADE Suite and SCADE Test Model Coverage, compliance
1a	Analysis of requirements	++	No Support	
1b	Generation and analysis of equivalence classes	++	Reduce	SCADE Test Model Coverage supports the analyses of equivalence classes through the creation of user defined coverage criteria
1c	Analysis of boundary values	++	No Support	
1d	Error guessing based on knowledge or experience	++	No Support	
1e	Analysis of functional dependencies	++	Reduce	SCADE Suite supports the analysis of functional dependencies in-between software units. SCADE Test Model Coverage performs an analysis of how completely these functional dependencies are exercised by requirements-base tests.
1f	Analysis of operational use cases	++	No Support	

APPENDIX D
SCADE SUPPORT OF ASPICE

D.1 ASPICE overview

Automotive Software Performance Improvement and Capability dEtermination (Automotive SPICE®) [ASPICE] is a **process reference and assessment model** that provides a framework for defining, implementing, and evaluating the process required for system development focused on software and system parts in the automotive industry. It is derived from ISO/IEC 15504 Information technology – Process assessment, also termed Software Process Improvement and Capability dEtermination (SPICE) [SPICE].

D.2 The ASPICE process reference model

Processes are grouped by process category (Acquisition, System Engineering, Software Engineering, etc.) and at a second level into process groups according to the type of activity they address. Each process is described in terms of a purpose statement. For the process dimension, the Automotive SPICE process reference model provides the set of processes shown in Figure 105.

Source: Figure 2 of Automotive SPICE Version 3.1

FIGURE 105: OVERVIEW OF THE AUTOMOTIVE SPICE PROCESS REFERENCE MODEL

The system and software engineering processes in ASPICE have been organized according to the V model of Figure 106. The information flow on the left side of the "V" is ensured by a Base Practice (BP) "Communicate agreed work product x" and on the information flow on the right side is ensured through a Base Practice "Summarize and communicate results".

Source: Figure D.2 of Automotive SPICE Version 3.1

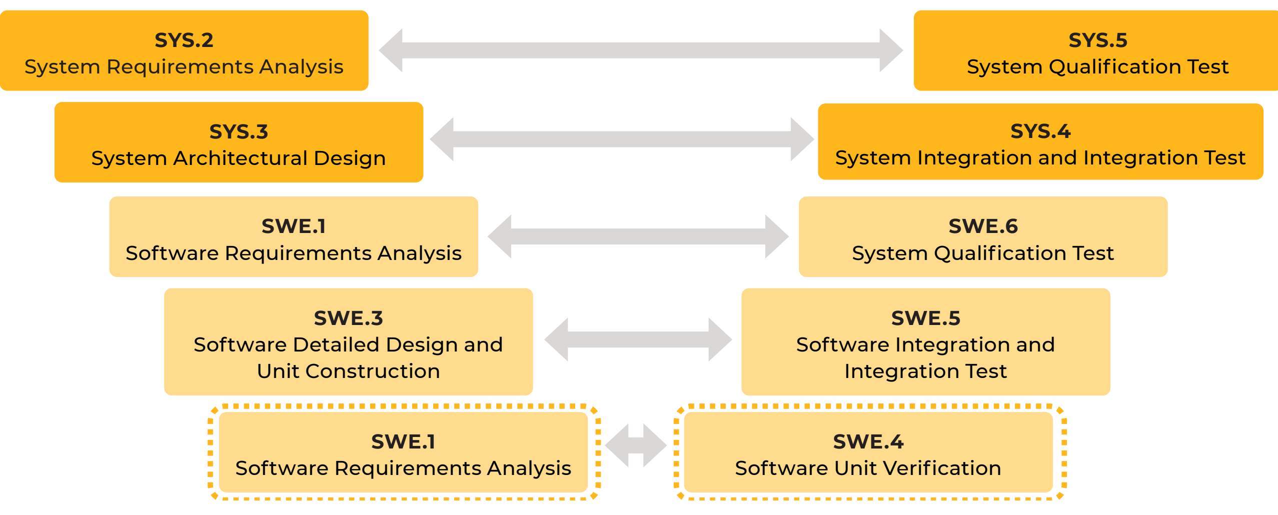


FIGURE 106: THE ASPICE V MODEL FOR ENGINEERING PROCESSES

D.3 Traceability and consistency in ASPICE

Traceability and consistency are addressed by two separate Best Practices in ASPICE. Traceability refers to the existence of links between work products and consistency addresses the content and semantics of the work products. This is depicted in Figure 107.

Source: Figure D.4 of Automotive SPICE Version 3.1

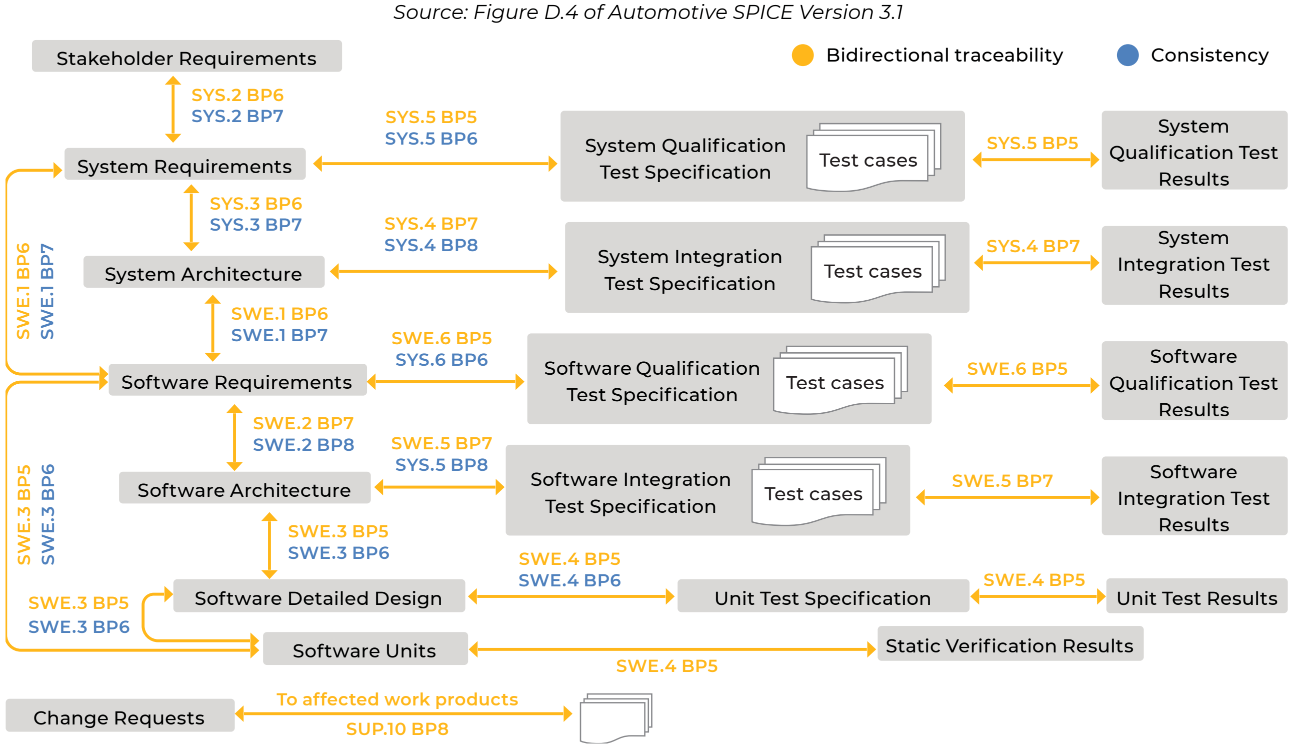


FIGURE 107: BIDIRECTIONAL TRACEABILITY AND CONSISTENCY IN ASPICE

D.4 The ASPICE capability assessment model

Process capability levels (6) and process attributes (9), which are features of a process that can be evaluated on a scale of achievement providing a measure of the capability of the process, have been defined in ASPICE according to [ISO/IEC 33020] and as described in Table 56 and Table 57.

TABLE 59: PROCESS CAPABILITY LEVELS ACCORDING TO ISO/IEC 33020

Source: Table 10 in ASPICE Version 3.1

Level 0: Incomplete process	The process is not implemented or fails to achieve its process purpose
Level 1: Performed process	The implemented process achieves its process purpose
Level 2: Managed process	The previously described performed process is now implemented in a managed fashion (planned, monitored, and adjusted) and its work products are appropriately established, controlled, and maintained
Level 3: Established process	The previously described managed process is now implemented using a defined process that is capable of achieving its process outcomes
Level 4: Predictable process	The previously described established process now operates predictively within defined limits to achieve its process outcomes. Quantitative management needs are identified, measurement data are collected and analyzed to identify assignable causes of variation. Corrective action is taken to address assignable causes of variation
Level 5: Innovating process	The previously described predictable process is now continually improved to respond to organizational change.

TABLE 60: PROCESS ATTRIBUTES ACCORDING TO ISO/IEC 33020

Source: Table 11 in ASPICE Version 3.1

Level 0: Incomplete process		
Level 1: Performed process	**PA 1.1**	Process performance process attribute
Level 2: Managed process	**PA 2.1**	Performance management process attribute
	PA 2.2	Work product management process attribute
Level 3: Established process	**PA 3.1**	Process definition process attribute
	PA 3.2	Process deployment process attribute
Level 4: Predictable process	**PA 4.1**	Quantitative analysis process attribute
	PA 4.2	Quantitative control process attribute
Level 5: Innovating process	**PA 5.1**	Process innovation process attribute
	PA 5.2	Process innovation implementation process attribute

D.5 SCADE support of ASPICE

This Section focuses on the Software Engineering Process Group (SWE) of ASPICE [ASPICE], described as part of Figure 105. It describes the support that the SCADE toolchain, used together with the ISO 26262-6:2018 compliant process described in this handbook, provides in terms of assisting a company using the toolchain while seeking compliance with ASPICE. Table 58 provides the Level of Support of SCADE (Full Support/ Partial Support/ No Support) for each relevant Base Practice in the SYS and SWE process categories.

Note: In the table below, "Full Support" from a tool describes the support from the tool's perspective. It does not mean that the tool user has no activity to perform. For example, SCADE Suite KCG achieves automatic source code generation from a design model. However, the KCG user must check the KCG log file to ensure that code generation completed normally, as specified in the conditions of use of the tool.

TABLE 61: SCADE SUPPORT OF ASPICE

Source: Extract from ASPICE Version 3.1, System Engineering, Software Engineering, and Supporting process groups

Base Practices	Level of Support	SCADE Support
	SYS.2 System Requirements Analysis	
		Can be supported by a SysML tool as SCADE Architect. medini Analyze can be used to capture and manage functional and technical safety requirements with links to HARA, FHA, or PHA.
	SYS.3 System Architectural Design	
		Can be supported by a SysML tool such as SCADE Architect
	SWE.1 Software Requirements Analysis	
SWE.1.BP1: Specify software requirements – Use the system requirements and the system architecture and changes to system requirements and architecture to identify the required functions and capabilities of the software. Specify functional and non-functional software requirements in a software requirements specification.	Partial Support	Can be supported by a SysML tool such as SCADE Architect that allows to describe the behavioral part of software requirements. medini Analyze can also be used to identify and create new software safety requirement point out during the safety analysis.

Base Practices	Level of Support	SCADE Support
SWE.1.BP2: Structure software requirements – Structure the software requirements in the software requirements specification by *e.g.* – grouping to project relevant clusters, – sorting in a logical order for the project, – categorizing based on relevant criteria for the project, – prioritizing according to stakeholder needs.	No Support	
SWE.1.BP3: Analyze software requirements – Analyze the specified software requirements including their interdependencies to ensure correctness, technical feasibility and verifiability, and to support risk identification. Analyze the impact on cost, schedule and the technical impact.	No Support	
SWE.1.BP4: Analyze the impact on the operating environment – Analyze the impact that the software requirements will have on interfaces of system elements and the operating environment.	No Support	
SWE.1.BP5: Develop verification criteria – Develop the verification criteria for each software requirement that define the qualitative and quantitative measures for the verification of a requirement	Partial Support	When defining verification criteria and test strategy, SCADE Test capabilities (*incl.* creation and management of test cases, Model-in-the-Loop testing, and model coverage analysis) and SCADE Suite Design Verifier capabilities must be considered. SCADE Test allows the user to describe qualitative and quantitative verification criteria.
SWE.1.BP6: Establish bidirectional traceability – Establish bidirectional traceability between system requirements and software requirements. Establish bidirectional traceability between the system architecture and software requirements.	Partial Support	If SCADE Architect is used for the system architecture, system architecture and software requirements traceability can be performed using the SCADE LifeCycle ALM Gateway
SWE.1.BP7: Ensure consistency – Ensure consistency between system requirements and software requirements. Ensure consistency between the system architecture and software requirements.	No Support	

Base Practices	Level of Support	SCADE Support
SWE.1.BP8: Communicate agreed software requirements – Communicate the agreed software requirements and updates to software requirements to all relevant parties	No Support	*Note: Software requirements allocated to SCADE components must be provided to the SCADE development team*
	SWE.2 Software Architectural Design	
SWE.2.BP1: Develop software architectural design – Develop and document the software architectural design that specifies the elements of the software with respect to functional and non-functional software requirements. *NOTE 1: The software is decomposed into elements across appropriate hierarchical levels down to the software components (the lowest level elements of the software architectural design) that are described in the detailed design.*	Partial Support	Two options are available: **1.** Develop the software architecture model in SCADE Architect from the allocated software requirements. Documentation (software architectural report) can be automatically generated from the SCADE Architect model with the SCADE LifeCycle Reporter. For the software elements that will be implemented in SCADE Suite, direct synchronization between SCADE Architect and SCADE Suite allows to automatically define the software element interfaces at detailed design level. **2.** Design directly the software architecture in SCADE Suite. A SCADE Suite architecture is an architecture diagram, with no behavioral description, that only describes the decomposition of the software elements into high-level operators, their interfaces, and the flows between these operators. This approach is applicable when the software architecture is composed of SCADE elements. Documentation (software architectural report) can be automatically generated from the SCADE Suite model with the SCADE LifeCycle Reporter. SCADE LifeCycle Model Change supports incremental reviews of these models
SWE.2.BP2: Allocate software requirements – Allocate the software requirements to the elements of the software architectural design.	Partial Support	Capture the traceability between the allocated software requirements and the software architectural design (SCADE Architect or SCADE Suite) with SCADE LifeCycle ALM Gateway
WE.2.BP3: Define interfaces of software elements – Identify, develop and document the interfaces of each software element.	Full Support	Define interfaces of the SCADE Architecture elements (in SCADE Architect or SCADE Suite). Documentation can be automatically generated with the SCADE LifeCycle Reporter. SCADE LifeCycle Model Change supports incremental reviews of these models

Base Practices	Level of Support	SCADE Support
SWE.2.BP4: Describe dynamic behavior – Evaluate and document the timing and dynamic interaction of software elements to meet the required dynamic behavior of the system. *NOTE 2: Dynamic behavior is determined by operating modes (e.g. start-up, shutdown, normal mode, calibration, diagnosis, etc.), processes and process intercommunication, tasks, threads, time slices, interrupts, etc.* *NOTE 3: During evaluation of the dynamic behavior the target platform and potential loads on the target should be considered.*	Full Support	The dynamic behavior can be captured in SCADE Architect using the behavioral diagrams (Activity diagram, Sequence diagram, State Machine diagram and Use Case diagram). Specific information (*e.g.*, Timing, scheduling, ...) can be captured in the model using annotations. Specific analysis can be implemented directly via scripting using the model API. Data can also be formatted and exported for processing by dedicated tools. Documentation can be automatically generated with the SCADE LifeCycle Reporter. SCADE LifeCycle Model Change supports incremental reviews of these models
SWE.2.BP5: Define resource consumption objectives – Determine and document the resource consumption objectives for all relevant elements of the software architectural design on the appropriate hierarchical level *NOTE 4: Resource consumption is typically determined for resources like Memory (ROM, RAM, external / internal EEPROM or Data Flash), CPU load, etc.*	Partial Support	Resource consumption information can be captured in the SCADE Architect model using annotations. Specific analysis can be implemented directly via scripting using the model API. Data can also be formatted and exported for processing by dedicated tools. Documentation can be automatically generated with the SCADE LifeCycle Reporter
SWE.2.BP6: Evaluate alternative software architectures – Define evaluation criteria for the architecture. Evaluate alternative software architectures according to the defined criteria. Record the rationale for the chosen software architecture. *NOTE 5: Evaluation criteria may include quality characteristics (modularity, maintainability, expandability, scalability, reliability, security realization and usability) and results of make-buy-reuse analysis.*	Partial Support	User level activity using different versions of the architecture model
SWE.2.BP7: Establish bidirectional traceability between software requirements and elements of the software architectural design – *NOTE 6: Bidirectional traceability covers allocation of software requirements to the elements of the software architectural design.* *NOTE 7: Bidirectional traceability supports overage, consistency and impact analysis.*	Full Support	Traceability between software requirements and elements of the SCADE software architecture is performed with SCADE LifeCycle ALM Gateway. SCADE LifeCycle ALM Gateway allows the user to bridge Application Lifecycle Management tools to SCADE enabling to graphically manage links between model and requirements. The ALM Gateway imports assets using Requirement Management capabilities included in ALM tools and allows performing traceability between requirements and SCADE model items When traceability has been established, the traceability matrix can be generated from the ALM tool.

Base Practices	Level of Support	SCADE Support
SWE.2.BP8: Ensure consistency – Ensure consistency between software requirements and the software architectural design. *NOTE 8: Consistency is supported by bidirectional traceability and can be demonstrated by review records.*	Partial Support	For software architecture model captured in SCADE Architect, consistency check is supported by: – Review based on SCADE allocated software requirements and the SCADE Architect model traceability data – Project specific verification rules can be implemented in the Rule Checker For software architecture model captured in SCADE Suite, consistency check is supported by: – Automated SCADE Semantic Checker on the SCADE Suite Architecture model to verify consistency of both the interface and the connections – Review based on SCADE allocated software requirements and the SCADE Suite Architecture elements traceability data – Project specific verification rules can be implemented in the Rule Checker
SWE.2.BP9: Communicate agreed software architectural design. Communicate the agreed software architectural design and updates to software architectural design to all relevant parties.	Full Support	Communication can be done directly by sharing the models or by sharing the documentation automatically generated by the SCADE LifeCycle Reporter Updates can be precisely identified using model diff
SWE.3 Software Detailed Design and Unit Construction		
SWE.3.BP1: Develop software detailed design – Develop a detailed design for each software component defined in the software architectural design that specifies all software units with respect to functional and non-functional software requirements.	Full Support	Develop SCADE Suite Detailed models from SCADE allocated software requirements and from the software architecture. A best practice is to define a modeling standard and ensures its enforcement with the Rule Checker. Documentation (software detailed design) can be automatically generated from the SCADE Suite model with the SCADE LifeCycle Reporter. SCADE LifeCycle Model Change supports incremental reviews of these models
SWE.3.BP2: Define interfaces of software units – Identify, specify, and document the interfaces of each software unit.	Full Support	Interfaces of the software units are defined in the SCADE model. SCADE Suite modeling language natively supports the concept of interfaces and structured data types. Documentation (software interfaces and detailed design) can be automatically generated from the SCADE Suite model with the SCADE LifeCycle Reporter. SCADE LifeCycle Model Change supports incremental reviews of these models
SWE.3.BP3: Describe dynamic behavior – Evaluate and document the dynamic behavior of and the interaction between relevant software units. *NOTE 1: Not all software units have dynamic behavior to be described.*	Full Support	Develop the dynamic behavior into the SCADE Suite Design model. The formality of Scade language ensures a non-ambiguous implementation of the dynamic behavior using both high level data and control structures. Documentation (software detailed design) can be automatically generated from the SCADE Suite model with the SCADE LifeCycle Reporter. SCADE LifeCycle Model Change supports incremental reviews of these models

Base Practices	Level of Support	SCADE Support
SWE.3.BP4: Evaluate software detailed design – Evaluate the software detailed design in terms of interoperability, interaction, criticality, technical complexity, risks and testability. *NOTE 2: The results of the evaluation can be used as input for software unit verification.*	Full Support	The formal foundation of the Scade language and the SCADE Suite Semantic Checker greatly reduce the evaluation activities. Static analysis of the model by the Semantic checker (*i.e.*, checks that the detail design is consistent, data flows are properly typed, initializations are properly done) are achieved by the front-end of the SCADE Suite code generator and is therefore qualified. Specific project rules (*e.g.*, design complexity) can also be implemented in SCADE Suite Rule Checker allowing automatic verifications. Debugging and Model-in-the-Loop testing of the detailed design or any of its blocks allow early detection of design errors. SCADE Suite Design Verifier can be used to formally express and assess safety requirements. Once test cases are available, SCADE Test Model Coverage can be used to compute the model coverage by the test suite, detecting unintended functionality expressed by the detailed design
SWE.3.BP5: Establish bidirectional traceability – Establish bidirectional traceability between software requirements and software units. Establish bidirectional traceability between the software architectural design and the software detailed design. Establish bidirectional traceability between the software detailed design and software units. *NOTE 3: Redundancy should be avoided by establishing a combination of these approaches that covers the project and the organizational needs.* *NOTE 4: Bidirectional traceability supports coverage, consistency and impact analysis.*	Full Support	Traceability between software requirements and the SCADE model is established with SCADE LifeCycle ALM Gateway. SCADE LifeCycle ALM Gateway allows the user to bridge Application Lifecycle Management tools to SCADE enabling to graphically manage links between model and requirements. The ALM Gateway imports assets using Requirement Management capabilities included in ALM tools and allows performing traceability between requirements and SCADE model items. When traceability has been performed, the traceability matrix can be generated from the ALM tool. Traceability between SCADE models (detailed design) and the software units is ensured by KCG generated trace data
SWE.3.BP6: Ensure consistency. Ensure consistency between software requirements and software units. Ensure consistency between the software architectural design, the software detailed design and software units. *NOTE 5: Consistency is supported by bidirectional traceability and can be demonstrated by review records.*	Partial Support	Consistency check of the SCADE detailed design model is supported by: – Automated SCADE Suite Semantic Checker on the SCADE detailed design model to verify consistency thanks to Scade formal foundation – Review based on traceability between SCADE allocated software requirements and Scade model Consistency between Scade models detailed and architectural design is ensured by SCADE Suite Semantic Checker
SWE.3.BP7: Communicate agreed software detailed design – Communicate the agreed software detailed design and updates to the software detailed design to all relevant parties.	Full Support	Communication can be done directly by sharing the models or by sharing the documentation automatically generated by the SCADE LifeCycle Reporter. SCADE LifeCycle Model Change supports incremental reviews of these models

Base Practices	Level of Support	SCADE Support
	SWE.4 Software Unit Verification	
SWE.4.BP1: Develop software unit verification strategy including regression strategy – Develop a strategy for verification of the software units including regression strategy for re-verification if a software unit is changed. The verification strategy shall define how to provide evidence for compliance of the software units with the software detailed design and with the non-functional requirements. *NOTE 1: Possible techniques for unit verification include static/dynamic analysis, code reviews, unit testing etc.*	Full Support	Verification of software units is performed at the model level. Qualification of KCG compiler guarantees that model and code behavior are the same, activities at code level are eliminated. The verification of the compliance of the SCADE model is supported by: – the review of the model from the report generated with SCADE LifeCycle Reporter to verify compliance of the SCADE model with the SCADE allocated software requirements – the verification of compliance with a modeling guidelines or modeling standard with SCADE Rule Checker – the development of requirements-based test cases with SCADE Test – the regression strategy will be based on SCADE Test for test cases execution on host and on target – the verification of model coverage by the test cases with SCADE Model Coverage. The qualification of SCADE Test Model Coverage ensures that model coverage implies code coverage – the formal verification with SCADE Suite Design Verifier. Formal verification allows verification of safety properties The Timing and stack usage verification to verify compatibility with target CPU The verification of cross compiler combability with the SCADE Compiler Verification Kit combined with the analysis of Scade models complexity to ensure that generated code is in the range of the target compiler
SWE.4.BP2: Develop criteria for unit verification – Develop criteria for unit verification that are suitable to provide evidence for compliance of the software units, and their interactions within the component, with the software detailed design and with the non-functional requirements according to the verification strategy. For unit testing, criteria shall be defined in a unit test specification. *NOTE 2: Possible criteria for unit verification include unit test cases, unit test data, static verification, coverage goals and coding standards such as the MISRA rules.* *NOTE 3: The unit test specification may be implemented e.g. as a script in an automated test bench.*	Full Support	Develop verification cases in SCADE Test and review checklists to support the verification strategy. Verification criteria can use metrics computed by SCADE Test (*e.g.*, coverage ratio, test passed ratio, ...). Code generated by SCADE Suite KCG is MISRA compliant

Base Practices	Level of Support	SCADE Support
SWE.4.BP3: Perform static verification of software units – Verify software units for correctness using the defined criteria for verification. Record the results of the static verification. *NOTE 4: Static verification may include static analysis, code reviews, checks against coding standards and guide-lines, and other techniques.* *NOTE 5: see SUP.9 for handling of non-conformances.*	Full Support	Code verification activities eliminated with KCG qualification. All verification activities are performed on the Scade model SCADE Suite Design Verifier can be used to check properties, if relevant SCADE Suite TSO/TSV can be used to check stack and WCET
SWE.4.BP4: Test software units – Test software units using the unit test specification according to the software unit verification strategy. Record the test results and logs. *NOTE 6: see SUP.9 for handling of non-conformances.*	Full Support	Perform testing with SCADE Test. Test results report is automatically generated by SCADE Test. Model structural coverage is measured with SCADE Model Coverage which ensures that model coverage ensures code coverage
SWE.4.BP5: Establish bidirectional traceability – Establish bidirectional traceability between software units and static verification results. Establish bidirectional traceability between the software detailed design and the unit test specification. Establish bidirectional traceability between the unit test specification and unit test results. *NOTE 7: Bidirectional traceability sup-ports coverage, consistency and impact analysis.*	Full Support	Traceability between software units and static verification results is captured at model level, between the model and the static verification results (review results, rule checker results) Traceability between the detailed design and the unit test case is captured between the model elements and the SCADE Verification Cases Traceability between the SCADE verification cases and the SCADE verification results is provided by SCADE Test in the test execution results report Traceability between SCADE verification cases and software requirements is captured with SCADE LifeCycle ALM Gateway
WE.4.BP6: Ensure consistency – Ensure consistency between the software detailed design and the unit test specification. *NOTE 8: Consistency is supported by bidirectional traceability and can be demonstrated by review records.*	Partial Support	Consistency between the SCADE model and the verification cases is performed by review using the traceability between the model and the verification cases
SWE.4.BP7: Summarize and communicate results – Summarize the unit test results and static verification results and communicate them to all affected parties. *NOTE 9: Providing all necessary infor-mation from the test case execution in a summary enables other parties to judge the consequences.*	Full Support	Communication on SCADE Test results report (test execution report and model coverage report) and static verification results is supported by reports automatically generated from SCADE tool chain

Base Practices	Level of Support	SCADE Support
	SWE.5 Software Integration and Integration Test	
SWE.5.BP1: Develop software integration strategy. Develop a strategy for integrating software items consistent with the project plan and release plan. Identify software items based on the software architectural design and define a sequence for integrating them.	Partial Support	SCADE integration toolbox can help automate the code generation and part of the integration *Note: The formality of SCADE language and the qualification of KCG simplify the integration strategy when integrating SCADE components inside a top level SCADE*
SWE.5.BP2: Develop software integration test strategy including regression test strategy. Develop a strategy for testing the integrated software items following the integration strategy. This includes a regression test strategy for re-testing integrated software items if a software item is changed.	No Support	*Note: SCADE Test can be used for integration testing of SCADE components in a top level SCADE*
SWE.5.BP3: Develop specification for software integration test. Develop the test specification for software integration test including the test cases according to the software integration test strategy for each integrated software item. The test specification shall be suitable to provide evidence for compliance of the integrated software items with the software architectural design. *NOTE 1: Compliance to the architectural design means that the specified integration tests are suitable to prove that the interfaces between the software units and between the software items fulfill the specification given by the software architectural design.* NOTE 2: *The software integration test cases may focus on* • *the correct dataflow between software items* • *the timeliness and timing dependencies of dataflow between software items* • *the correct interpretation of data by all software items using an interface* • *the dynamic interaction between software items* • *the compliance to resource consumption objectives of interfaces*	No Support	*Note: SCADE Test can be used for integration testing of SCADE components in a top level SCADE*

Base Practices	Level of Support	SCADE Support
SWE.5.BP4: Integrate software units and software items. Integrate the software units to software items and software items to integrated software according to the software integration strategy.	Partial Support	SCADE integration toolbox can help automate the code generation and part of the integration *Note: The formality of SCADE language and the qualification of KCG simplify the integration strategy when integrating SCADE components inside a top level SCADE*
SWE.5.BP5: Select test cases. Select test cases from the software integration test specification. The selection of test cases shall have sufficient coverage according to the software integration test strategy and the release plan.	No Support	*Note: SCADE Test can be used for integration testing of SCADE components in a top level SCADE*
SWE.5.BP6: Perform software integration test. Perform the software integration test using the selected test cases. Record the integration test results and logs. *NOTE 4: see SUP.9 for handling of non-conformances.* *NOTE 5: The software integration test may be supported by using hardware debug interfaces*	No Support	*Note: SCADE Test can be used for integration testing of SCADE components in a top level SCADE*
SWE.5.BP7: Establish bidirectional traceability between elements of the software architectural design and test cases / between test cases included in the software integration test specification and software integration test results	No Support	*Note: If SCADE Architect or SCADE Suite are used to capture the software architecture, SCADE LifeCycle ALM Gateway can be used to capture traceability between SCADE verification cases and elements of the software architecture*
SWE.5.BP8: Ensure consistency. Ensure consistency between elements of the software architectural design and test cases included in the software integration test specification.	No Support	*Note: If SCADE Architect or SCADE Suite are used to capture the software architecture and if SCADE Test is used for integration testing of SCADE components in a top level SCADE, consistency between architectural design and the verification cases is performed by review using the traceability between the model and the verification cases*
SWE.5.BP9: Summarize and communicate results. Summarize the software integration test results and communicate them to all affected parties.	No Support	*Note: If SCADE Test is used for integration testing of SCADE components in a top level SCADE, communication shall be based on the SCADE Test results report (test execution report and model coverage report)*
	SUP.1 Quality Assurance	
		A specific quality assurance strategy shall be developed to audit the processes and work products associated to SCADE tool chain. Specific SCADE Quality Assurance checklists shall be developed to support the audit activities
	SUP.8 Configuration Management	
		The configuration management strategy shall ensure that the SCADE artifacts are controlled. List and format of SCADE artifacts are listed in SCADE user documentation

APPENDIX E
QUALIFICATION OF SCADE CODE GENERATION AND VERIFICATION TOOLS FOR ISO 26262:2018

E.1 Qualification of SCADE Suite KCG

The **SCADE Suite KCG code generator** is used for generating C source code from design models for application software up to ISO 26262:2018 ASIL D, without verification of its output.

According to the classification discussed in Section 2.5.1 of this handbook and in agreement with Table 3 of ISO 26262-8:2018, this use of KCG mandates qualification of the code generator at TCL3.

Regarding the choice of a qualification method described in Section 2.5.2 and in agreement with Table 4 of IS 26262-8:2018, SCADE Suite KCG has been developed in accordance with a safety standard (*i.e.*, using method 1d of Table 4).

More precisely, SCADE Suite KCG 6.6.2 has been developed in accordance with:

- IEC 61508 at SIL 3
- DO-330 at TQL-1 (usable for DO-178C applications at DAL A)
- EN 50128 at SIL 3/4

SCADE Suite KCG 6.6.2 development has been audited by TÜV. The corresponding Certificate is shown in Appendix G and the Report to the Certificate is available in [KCG-Report to the Certificate]. This Report states that “SCADE Suite KCG 6.6.2 complies with the testing criteria specified for ASIL D according to ISO 26262” and that “the applied plans, standards and guidelines listed in the Safety Case (see chapter 4.2) guarantee that Code Generator SCADE Suite KCG 6.6.2 is developed in a safe manner.”

E.1.1 Development of SCADE Suite KCG

The achievement of the above objectives for DO-330/TQL-1, IEC 61508/SIL 3 and EN 50128/SIL 3/4 is described in the following documents, audited by Certification Authorities on many past projects:

- **Compliance Analysis IEC 61508** [SCS-KCG-IEC61508-SIL3-COMPL] presents KCG compliance with IEC 61508 [IEC 61508] objectives at SIL 3
- **Compliance Analysis DO-330** [SCS-KCG-DO330-TQL1-COMPL] presents KCG compliance with DO-330 [DO-330] objectives at TQL-1
- **Compliance Analysis EN 50128** [SCS-KCG-EN50128-SIL3/4-COMPL] presents KCG compliance with EN 50128 [EN 51028] objectives at SIL 3/4
- **Tool Qualification Plan** [SCS-KCG-TQP] presents all provisions taken for KCG code generator qualification and references other project plans
- **Tool Operational Requirements** [SCS-KCG-TOR] describes KCG functionality and usage. It matches the Developer-TOR defined in [DO-330]
- **Scade Language Reference Manual** [SCS-KCG-LRM] contains the Scade language definition
- **Tool Accomplishment Summary** [SCS-KCG-TAS] shows compliance status with TQP, conditions of use, list of unresolved defects and tool limitations

- **Software Installation Procedure** [SCS-KCG-SIP] contains detailed instructions for installing KCG
- **Tool Configuration Index** [SCS-KCG-TCI] presents tool version and configuration
- **Tool Life Cycle Environment Configuration Index** [SCS-KCG-TECI] presents the software environment used for tool development and qualification

E.1.2 SCADE Suite KCG safety case

A hazard and risk assessment has been performed regarding the use of SCADE Suite KCG to identify the potential hazards of the tool and formulate safety goals related to the prevention or mitigation of these hazards.

This has been recorded in the **SCADE Suite KCG 6.6.2 – Safety Case** [SCS-KCG-Safety Case] which contains the following information:

- System and software definition
- Quality management report
- Safety management report
- Technical safety report

The Technical safety report contains a description of application conditions that have been established by performing hazard and risk assessment. These application conditions are listed in [SCS-KCG-Safety Case] along with additional application conditions gathered from [SCS-KCG-TOR] and [SCS-KCG-LRM].

The complete set of documents listed in Sections 16.1.1 and 16.1.2 is available to the SCADE users in the SCADE Suite KCG Certification Kit for ISO 26262:2018.

E.2 Qualification of SCADE Automotive Code Generator for AUTOSAR (ACG)

SCADE Suite AUTOSAR Code Generator (ACG) has been qualified for [ISO 26262:2018] at TCL3.

The categorization of the tool as TCL3, the qualification method and compliance to Clause 11 of ISO 26262-8:2018 are described in the compliance document [SCS-ACG-COMPL].

The achievement of the qualification objectives is described in the following documents:

- **Compliance Analysis ISO 26262** [SCS-ACG- COMPL] presents ACG compliance with ISO 26262 [ISO 26262] objectives at TCL3
- **Tool Operational Requirements** [SCS-ACG-TOR] describes ACG functionality and usage.
- **Release Note** [SCS-ACG-RN] contains ACG installation instructions, conditions of use, list of unresolved defects and tool limitations
- **Tool Configuration Index** [SCS-ACG-TCI] presents tool version and configuration

SCADE Automotive Code Generator for AUTOSAR (ACG) V2.1 development has been audited by TÜV. The corresponding Certificate is shown in Appendix G and the Report to the Certificate is available in [SCS-ACG- Report to the Certificate]. This Report states that “The tests and analyses performed by ANSYS France SAS have shown that SCADE Automotive Code Generator for AUTOSAR (ACG) complies with the testing criteria for tools according to ISO 26262”.

E.3 Qualification of SCADE LifeCycle Reporter and SCADE LifeCycle Model Change

SCADE LifeCycle Reporter for SCADE Suite is not designed as a tool to directly detect an error in SCADE Suite design models, but it is used to support the SCADE Suite design model review activity. Since this review activity is performed to detect errors in the model being developed, a malfunction of SCADE LifeCycle Reporter like for example failing to report some SCADE operators, may lead to the reviewer not reviewing part of the model and, therefore, failing to detect an error in the resulting software.

SCADE LifeCycle Reporter for SCADE Suite has been qualified for [ISO 26262:2018] at TCL3.

This qualification ensures completeness and consistency of the generated report according to the input model. The categorization of the tool as TCL3, the qualification method and compliance to Clause 11 of ISO 26262-8:2018 are described in the compliance document [SCS-MR-COMPL].

SCADE Suite LifeCycle Reporter for SCADE Suite has been audited by TÜV. The corresponding Certificate is shown in Appendix G and the Report to the Certificate is available in [MR-Report to the certificate]. This Report states that “SCADE LifeCycle Reporter for SCADE Suite complies with the testing criteria for tools according to ISO 26262”.

SCADE LifeCycle Model Change for SCADE Suite is not designed as a tool to directly detect an error in SCADE Suite design models, but it is used to support the SCADE Suite design model incremental review activity. Since this review activity is performed to detect errors in the model being developed, a malfunction of SCADE LifeCycle Model Change like for example failing to identify some modifications of SCADE operators, may lead to the reviewer not reviewing part of the model that has been modified and, therefore, failing to detect an error in the resulting software.

SCADE LifeCycle Model Change for SCADE Suite has been qualified for [ISO 26262:2018] at TCL3.

This qualification ensures completeness and consistency of the generated incremental report according to the input model in the previous and current iterations. The categorization of the tool as TCL3, the qualification method and compliance to Clause 11 of ISO 26262-8:2018 are described in the compliance document [SCS-MCH-COMPL].

E.4 Qualification of SCADE Test Environment for Host and SCADE Test Target Execution

SCADE Test Environment for Host (Model-in-the-Loop testing) is used to automate test execution and perform automatic checks to determine if tests are passed. An error in this tool may result in reporting a test as passed when it should not, which can result in a failure to detect an error in a Scade model.

SCADE Test Target Execution automates the translation of host test cases to specific target test cases**.** An error in this tool may result in creating erroneous target test cases which can result in a failure to detect an error in a Scade model/source code.

The categorization of SCADE Test Environment for Host and SCADE Test Target Execution tools as TCL3, the qualification method and compliance to Clause 11 of ISO 26262-8:2018 are described in the compliance document [SCS-STE-COMPL].

SCADE Test Environment has been audited by TÜV. The corresponding Certificate is shown in Appendix G and the Report to the Certificate is available in [TE-Report to the certificate]. This Report states that “SCADE Test Environment complies with the testing criteria for tools according to ISO 26262”.

E.5 Qualification of SCADE Test Model Coverage

SCADE Test Model Coverage allows to measure the coverage of a SCADE Suite model by test cases without the need to verify the tool outputs. Model coverage analysis allows to assess the thoroughness of Model-in-the-Loop testing of the software units design when used for verification of model compliance to the software requirements of the application.

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

SCADE Test Model Coverage has been qualified for [ISO 26262:2018] at TCL3. The categorization of the tool as TCL3, the qualification method and compliance to Clause 11 of ISO 26262-8:2018 are described in the compliance document [SCS-MCOV-COMPL].

SCADE Suite Test Model Coverage for SCADE Suite has been audited by TÜV. The corresponding Certificate is shown in Appendix G and the Report to the Certificate is available in [MCOV-Report to the certificate]. This Report states that "SCADE Test Model Coverage for SCADE Suite complies with the testing criteria for tools according to ISO 26262".

While the qualification credit of the Model Coverage tool covers the model coverage objective, it also extends to SCADE Suite KCG-generated code structural coverage objective, provided some conditions on models and code generation options are met (see [MCOV-FAQ11], extended by [MCOV-FAQ11-Ext] for a description of these conditions). This is worth explaining in more details.

As stated in Table 9 of ISO 26262-6:2018, in the case of model-based development, the analysis of structural coverage can be performed at model-level (see NOTE 3 of Table 9) and the analysis of structural coverage performed at model-level can replace the source code coverage provided it is shown to be equivalent (see EXAMPLE 4 of Table 9).

The coverage criteria of SCADE Suite Model Coverage (OMC/DC, ODC, Influence) are defined as a correspondence to code coverage criteria (MC/DC, Branch Coverage, Statement Coverage) in such a way that, when model coverage is achieved for a matching criterion, say OMC/DC, then structural coverage of the SCADE Suite KCG 6.6.2-generated code holds for the corresponding criterion, say MC/DC. In other words, SCADE Suite KCG preserves model coverage, meaning that achieving model coverage is enough to ensure that structural coverage of the generated code is also achieved for matching coverage criteria.

This enables SCADE Test Model Coverage and SCADE Suite KCG to meet the ISO 26262-6:2018 Table 9 condition to use Model coverage to also ensure structural coverage of the SCADE Suite KCG-generated code.

APPENDIX F
SCADE SUITE COMPILER VERIFICATION KIT (CVK)

F.1 SCADE Suite CVK overview

F.1.1 What SCADE Suite CVK is and is not

While SCADE Suite KCG qualification ensures that source code conforms to design model developed with SCADE Suite, CVK is a test suite that can be used to verify that the type of code generated by SCADE Suite KCG is correctly compiled/executed with a given cross-compiler for a given target.

CVK can be used for the following purposes:

- to support early verification of the correctness and consistency between the development toolchain and the target platform
- to address the verification of target

CVK is NOT:

- a validation suite of the C compiler. Such validation suites are generally available on the market. They rely on running large numbers of test cases covering all programming language constructs, the right number of combinations, and various compiling options. It is expected that the applicant requires evidence of this activity from the compiler provider (or other source)
- an executable software
- a hardware test suite

Since CVK is not a tool (it is a set of test cases and procedures), the concept of qualification is not relevant. Instead, CVK is verified with the same objectives as any other set of test cases and procedure, including review, requirements coverage analysis, and structural coverage analysis (MC/DC) (see [NASA-MCDC])

F.1.2 Role of SCADE Suite CVK

CVK is a test suite: it is part of verification means provided to SCADE Suite KCG users.

Figure 108 shows the complementary roles of KCG and CVK in the verification of the development environment of the users.

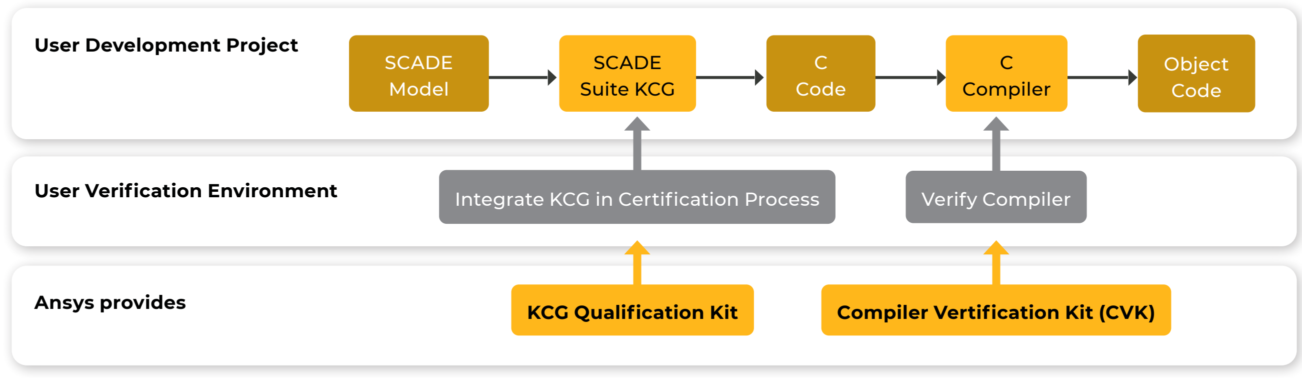


FIGURE 108: ROLE OF KCG AND CVK IN THE VERIFICATION OF USER DEVELOPMENT ENVIRONMENT

The role of CVK is twofold:

1. Compatibility analysis between the software architectural design and the cross-compiler for the target environment regarding:
    - Complexity of data structure nesting
    - Number of arguments in a function call
2. Compatibility analysis between the software unit design models and the cross-compiler for the target environment regarding:
    - Complexity of expressions
    - Complexity of control structures
    - Rounding to zero

F.1.3 SCADE Suite CVK contents

The CVK product is made of the following:

1. A CVK User's Manual [CVK-UM] and a Reference Manual [CVK-RM] containing:
    - Installation and user instructions
    - Description of the underlying methodology
    - Models' description
    - C sample description
    - Test cases and procedures description
    - Coverage matrices
    - C code complexity metrics description
2. The SCADE Suite-generated C sample to verify the C compiler.
3. A representative SCADE Suite Sample covering the set of Scade language primitive operators and enabling the generation of C sample with KCG in your own environment.
4. Requirements-based test cases to exercise the Scade C sample with 100 percent MC/DC coverage [NASA-MCDC] for all KCG settings.
5. Automated test procedures for the Windows platform.

F.1.4 C sample characteristics

The C sample is generated from a models database by SCADE Suite KCG and it exhibits the following characteristics:

- It contains an exhaustive set of elementary C constructs that can ever be generated from a model by the SCADE Suite KCG Code Generator.
- It contains a set of combinations of these elementary C constructs.

F.2 SCADE Suite CVK representativity

The source code generated by SCADE Suite KCG is a subset of C with several relevant safety properties in term of statements, data structures and control structures such as:

- No recursion or unbounded loop.
- No code with side effects (no a += b, no side effect in function calls).

- Communication between operators only goes through explicit data flows.
- No functions passed as arguments.
- No arithmetic on pointers.
- No pointer on function.
- No jump statement such as “goto” or “continue”
- Memory allocation is fully static (no dynamic memory allocation).
- Expressions are explicitly parenthesized.
- There are no implicit conversions.

CVK contains a representative sample of the generated code. This sample covers a subset of elementary C constructs as well as deeply nested constructs identified from C code complexity metrics.

The C code complexity metrics provided by CVK are relevant in the context of C compiler verification. These metrics, selected by analyzing compiler limits defined in C standards and cross-compilers documentation, address complexity both in depth and in width.

Each complexity metric has a limit defined by CVK to cover a certain degree of complexity. Therefore, CVK users must check that the complexity of the code generated by KCG from their SCADE Suite application fits in the limits covered by CVK. SCADE Suite KCG provides most values for these metrics in a dedicated generated file. Some other metrics are computed by scripts.

This approach addresses the concerns for compiler verification activities in the case of automatically generated code.

F.2.1 Strategy for developing SCADE Suite CVK

Figure 109 summarizes the strategy for developing and verifying CVK.

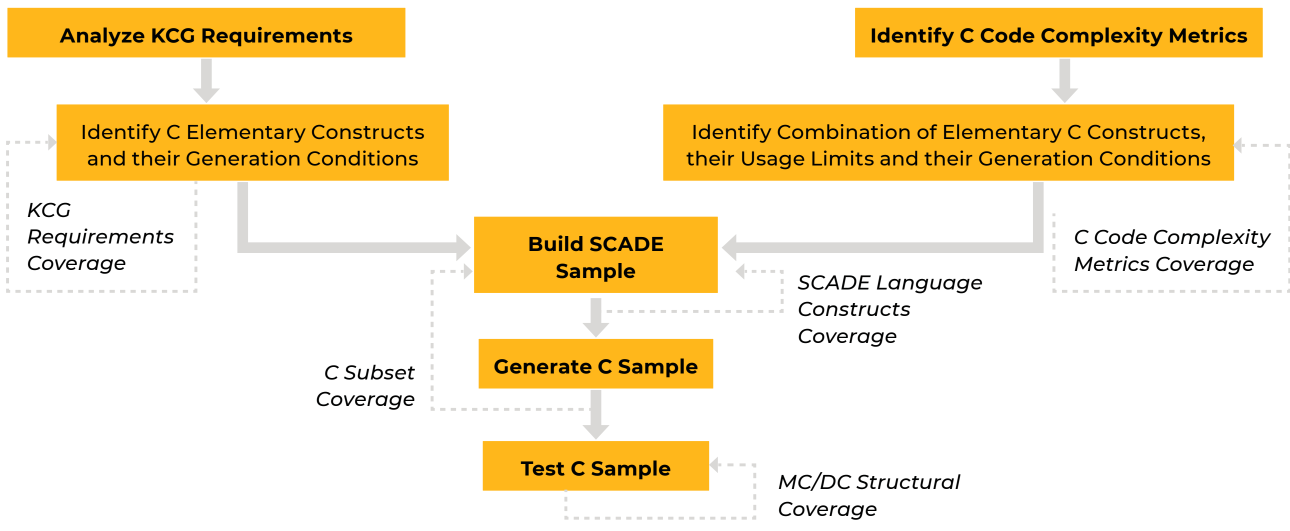


FIGURE 109: STRATEGY FOR DEVELOPING AND VERIFYING CVK

CVK is built in the following way:

1. Identify the C elementary constructs generated with KCG by analyzing the KCG software requirements. These C constructs are identified by a name and defined in terms of the C-ISO standard.
2. Define relevant complexity metrics for KCG-generated code by analyzing compilers limits defined in C standards and compilers documentation. These metrics address parameters

such as the number of levels of nested structures or the number of nesting levels of control structures.

3. Identify the combination of elementary C constructs generated by KCG that make sense in the compiler verification (in particular, focus on the risky events for a cross-compiler). These combinations are directly based on complexity metrics previously identified. Their usage limits and generation conditions are defined at this step.
4. Build the C sample:
   - **A)** A suite of Scade samples, covering all constructs, is built as material for code generation.
   - **B)** Each elementary C construct and their combination are generated from Scade samples root nodes with appropriate KCG options.
   - **C)** Coverage of the C subset (elementary C constructs and combination) by the C sample is required and verified.
5. Develop a test harness, exercising the C sample with a set of input vectors and verifying that the output vectors conform to the expected output vectors.
6. Tests execution on a host platform to verify:
   - **A)** Conformance of outputs to expected outputs.
   - **B)** MC/DC coverage at C code level.
7. Tests execution for each selected target platform to verify:
   - **A)** The adaptation to a specific cross environment capability of CVK (portability).
   - **B)** The correctness of effective output vectors on the platform.

F.2.2 Using SCADE Suite CVK

CVK is used as follows (see Figure 110):

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

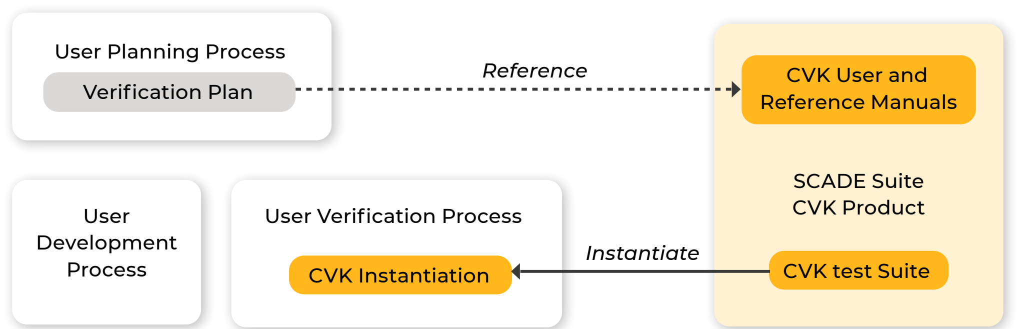


FIGURE 110: SCADE SUITE CVK IN USER PROCESSES

Figure 111 details the role of CVK (highlighted by shadowed boxes) in the verification of the compiler:

- The C sample is regenerated by KCG from the SCADE Suite sample, with specified KCG

options and is compared to the provided Reference C sample.

- From the C sample, the C compiler/linker generates an executable. Note that the C sample is always taken from the delivered reference sample, not from the regenerated C sample.
- The executable reads input vectors (from its static memory) and computes output vectors. It compares the actual output vectors to reference vectors (from its static memory). Comparison is performed directly in the C test harness. The C primitive "==" is used for boolean, integer and character data and a specific C function is used for floating point comparison with tolerance. Unit tests of these comparison C functions are provided in the CVK test suite to ensure that the C compiler correctly compiles these functions. The reference vectors were developed and verified when developing CVK, and are based on the requirements (*i.e.*, semantics of model).

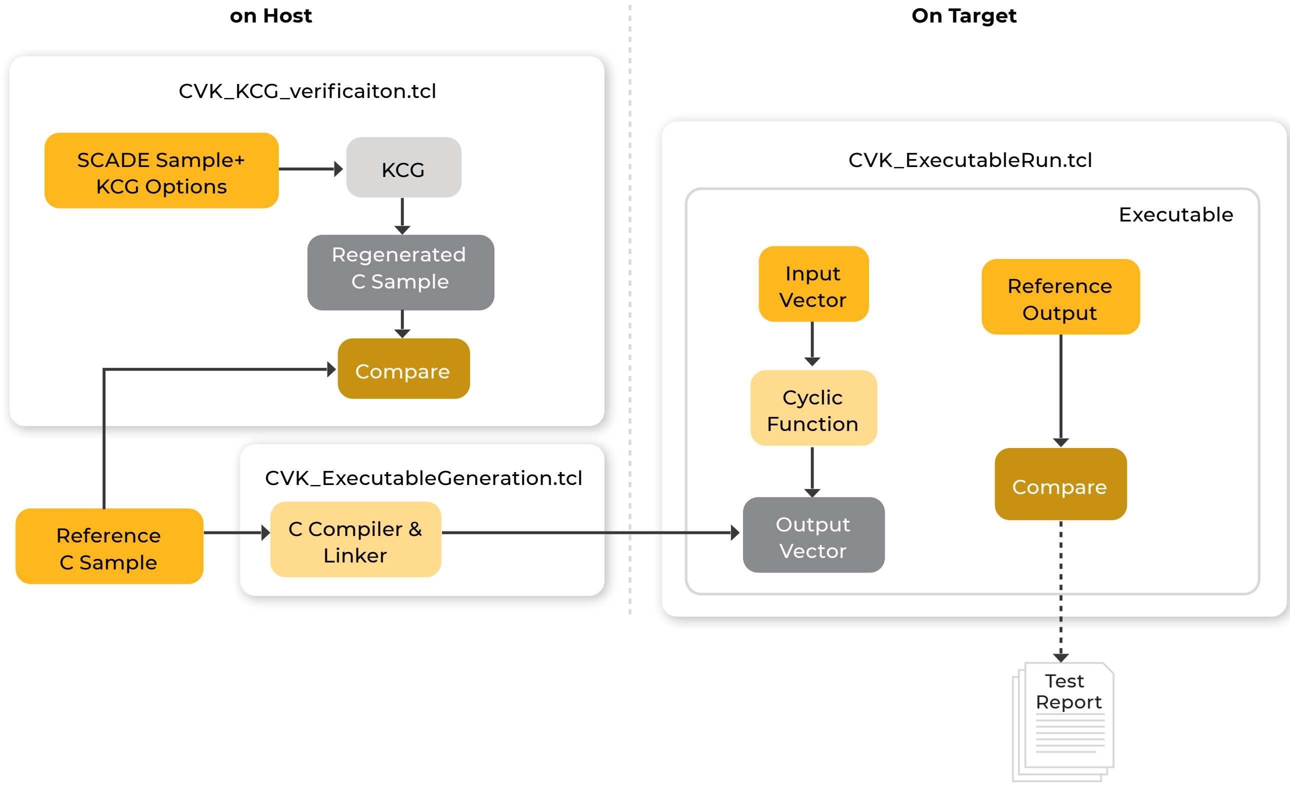


FIGURE 111: POSITION OF SCADE SUITE CVK IN THE COMPILER VERIFICATION PROCESS

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

APPENDIX G
TÜV SÜD SCADE CERTIFICATES

G.1 SCADE Suite KCG Certificate

ZERTIFIKAT ◆ CERTIFICATE ◆ 認證證書 ◆ СЕРТИФИКАТ ◆ CERTIFICADO ◆ CERTIFICAT

Product Service

C E R T I F I C A T E

No. Z10 055460 0022 Rev. 00

Holder of Certificate: ANSYS France SAS
15 Place Georges Pompidou
78180 Montigny-le-Bretonneux
FRANCE

Certification Mark:

Product: Software Tool for Safety Related Development

Model(s): Code Generator SCADE Suite KCG 6.6.2

Parameters: The code generator - classified as T3 offline support tool according to IEC 61508-4 / EN 50128 and TCL 3 according to ISO 26262-8 - is qualified for the use in safety-related software development according to IEC 61508, EN 50128 and ISO 26262.

The report AM97150C is a mandatory part of this certificate.

Tested according to:
IEC 61508-1:2010 (SIL 3)
IEC 61508-3:2010 (SIL 3)
EN 50128:2011/A2:2020 (SIL 3/4)
ISO 26262-8:2018 (ASIL D)

The product was tested on a voluntary basis and complies with the essential requirements. The certification mark shown above can be affixed on the product. It is not permitted to alter the certification mark in any way. In addition the certification holder must not transfer the certificate to third parties. This certificate is valid until the listed date, unless it is cancelled earlier. All applicable requirements of the testing and certification regulations of TÜV SÜD Group have to be complied. For details see: www.tuvsud.com/ps-cert

Test report no.: AM97150C
Valid until: 2026-07-13

Date, 2021-07-14

(Peter Weiß)

Page 1 of 1
TÜV SÜD Product Service GmbH • Certification Body • Ridlerstraße 65 • 80339 Munich • Germany

TÜV®

G.2 SCADE Automotive Code Generator for AUTOSAR (ACG) Certificate

ZERTIFIKAT ◆ CERTIFICATE ◆ 認證證書 ◆ СЕРТИФИКАТ ◆ CERTIFICADO ◆ CERTIFICAT

Product Service

CERTIFICATE

No. Z10 055460 0020 Rev. 00

Holder of Certificate: **ANSYS France SAS**
15 Place Georges Pompidou
78180 Montigny-le-Bretonneux
FRANCE

Certification Mark:

Product: **Software Tool for Safety Related Development**

Model(s): **SCADE Automotive Code Generator for AUTOSAR (ACG)**

Parameters:

The tool SCADE Automotive Code Generator for AUTOSAR (ACG), classified as TCL 3 according to ISO26262-8:2018, is qualified to be used in safety-related development for any ASIL.

The report listed below is a mandatory part of the certificate

Tested according to: ISO 26262-8:2018

The product was tested on a voluntary basis and complies with the essential requirements. The certification mark shown above can be affixed on the product. It is not permitted to alter the certification mark in any way. In addition the certification holder must not transfer the certificate to third parties. This certificate is valid until the listed date, unless it is cancelled earlier. All applicable requirements of the testing and certification regulations of TÜV SÜD Group have to be complied. For details see: www.tuvsud.com/ps-cert

Test report no.: AM96864C
Valid until: 2026-05-18

Date, 2021-05-20

(Guido Neumann)

Page 1 of 1
TÜV SÜD Product Service GmbH • Certification Body • Ridlerstraße 65 • 80339 Munich • Germany

TÜV®

G.3 SCADE Test Model Coverage Certificate

ZERTIFIKAT ◆ CERTIFICATE ◆ 認證證書 ◆ СЕРТИФИКАТ ◆ CERTIFICADO ◆ CERTIFICAT

Product Service

CERTIFICATE

No. Z10 055460 0014 Rev. 00

Holder of Certificate:	**ANSYS France SAS** 15 Place Georges Pompidou 78180 Montigny-le-Bretonneux FRANCE
Factory(ies):	055460
Certification Mark:	
Product:	**Software Tool for Safety Related Development**
Model(s):	**Model Coverage for SCADE Suite**
Parameters:	The tool is qualified to be used in safety-related development according to ISO26262:2018. The report listed below is a mandatory part of the certificate.
Tested according to:	ISO 26262-8:2018

The product was tested on a voluntary basis and complies with the essential requirements. The certification mark shown above can be affixed on the product. It is not permitted to alter the certification mark in any way. In addition the certification holder must not transfer the certificate to third parties. See also notes overleaf.

Test report no.: AV95045C
Valid until: 2025-04-27

Date, 2020-04-28

(Guido Neumann)

TÜV®

Page 1 of 1
TÜV SÜD Product Service GmbH • Certification Body • Ridlerstraße 65 • 80339 Munich • Germany

A4 / 07.17

G.4 SCADE LifeCycle Reporter Certificate

ZERTIFIKAT ◆ CERTIFICATE ◆ 認證證書 ◆ СЕРТИФИКАТ ◆ CERTIFICADO ◆ CERTIFICAT

Product Service

CERTIFICATE

No. Z10 055460 0016 Rev. 00

Holder of Certificate: **ANSYS France SAS**
15 Place Georges Pompidou
78180 Montigny-le-Bretonneux
FRANCE

Factory(ies): 055460

Certification Mark:

Product: **Software Tool for Safety Related Development**

Model(s): **SCADE Lifecycle Reporter for SCADE Suite**

Parameters: The tool is qualified to be used in safety-related development according to ISO26262:2018.

The report listed below is a mandatory part of the certificate.

Tested according to: ISO 26262-8:2018

The product was tested on a voluntary basis and complies with the essential requirements. The certification mark shown above can be affixed on the product. It is not permitted to alter the certification mark in any way. In addition the certification holder must not transfer the certificate to third parties. See also notes overleaf.

Test report no.: AV95049C
Valid until: 2025-04-27

Date, 2020-04-28 (Guido Neumann)

TÜV®

Page 1 of 1
TÜV SÜD Product Service GmbH • Certification Body • Ridlerstraße 65 • 80339 Munich • Germany

G.5 SCADE Test Environment Certificate

ZERTIFIKAT ◆ CERTIFICATE ◆ 認證證書 ◆ СЕРТИФИКАТ ◆ CERTIFICADO ◆ CERTIFICAT

Product Service

CERTIFICATE

Z10 055460 0015 Rev. 00

Holder of Certificate: **ANSYS France SAS**
15 Place Georges Pompidou
78180 Montigny-le-Bretonneux
FRANCE

Factory(ies): 055460

Certification Mark:

Product: **Software Tool for Safety Related Development**

Model(s): **SCADE Test Environment**

Parameters: The tool is qualified to be used in safety-related development according to ISO26262:2018.

The report listed below is a mandatory part of the certificate

Tested according to: ISO 26262-8:2018

The product was tested on a voluntary basis and complies with the essential requirements. The certification mark shown above can be affixed on the product. It is not permitted to alter the certification mark in any way. In addition the certification holder must not transfer the certificate to third parties. See also notes overleaf.

Test report no.: AV95047C
Valid until: 2025-04-27

Date, 2020-04-28 (Guido Neumann)

TÜV®

Page 1 of 1
TÜV SÜD Product Service GmbH • Certification Body • Ridlerstraße 65 • 80339 Munich • Germany

G.6 SCADE LifeCycle Model Change for SCADE Suite Certificate

ZERTIFIKAT ◆ CERTIFICATE ◆ 認證證書 ◆ СЕРТИФИКАТ ◆ CERTIFICADO ◆ CERTIFICAT

TÜV SÜD
Product Service

CERTIFICATE

No. Z10 055460 0021 Rev. 00

Holder of Certificate: **ANSYS France SAS**
15 Place Georges Pompidou
78180 Montigny-le-Bretonneux
FRANCE

Certification Mark:

Product: **Software Tool for Safety Related Development**

Model(s): **SCADE Lifecycle Model Change for SCADE Suite**

Parameters: The tool SCADE Lifecycle Model Change for SCADE Suite, classified as TCL 3 according to ISO26262-8:2018, is qualified to be used in safety-related development for any ASIL.

The report listed below is a mandatory part of the certificate

Tested according to: ISO 26262-8:2018

The product was tested on a voluntary basis and complies with the essential requirements. The certification mark shown above can be affixed on the product. It is not permitted to alter the certification mark in any way. In addition the certification holder must not transfer the certificate to third parties. This certificate is valid until the listed date, unless it is cancelled earlier. All applicable requirements of the testing and certification regulations of TÜV SÜD Group have to be complied.
For details see: www.tuvsud.com/ps-cert

Test report no.: AM97064C
Valid until: 2026-06-23

Date, 2021-06-25

(Guido Neumann)

Page 1 of 1
TÜV SÜD Product Service GmbH • Certification Body • Ridlerstraße 65 • 80339 Munich • Germany

TÜV®

ANSYS, Inc.
Southpointe
2600 Ansys Drive
Canonsburg, PA 15317
U.S.A.
724.746.3304
ansysinfo@ansys.com

Any and all ANSYS, Inc. brand, product, service and feature names, logos and slogans are registered trademarks or trademarks of ANSYS, Inc. or its subsidiaries in the United States or other countries. All other brand, product, service and feature names or trademarks are the property of their respective owners.

© 2022 ANSYS, Inc. All Rights Reserved.

Visit www.ansys.com for more information.