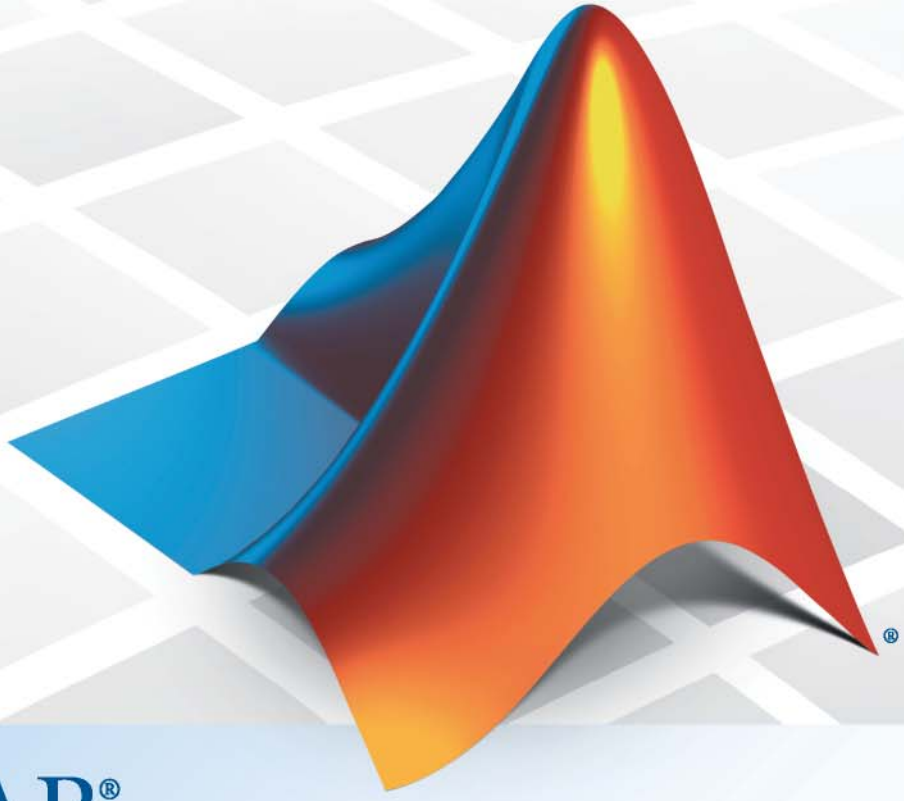
IEC Certification Kit 1

Application-Specific Verification and Validation of Models and Generated C and C++ Code



MATLAB® & SIMULINK®

MathWorks®

Accelerating the pace of engineering and science

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

IEC Certification Kit Application-Specific Verification and Validation of Models and Generated C and C++ Code

Revision History

March 2008	Online only	New for Version 1.1 (Applies to Release 2008a)
October 2008	Online only	Revised for Version 1.2 (Applies to Release 2008b)
March 2009	Online only	Revised for Version 1.3 (Applies to Release 2009a)
September 2009	Online only	Revised for Version 1.4 (Applies to Release 2009b)
April 2010	Online only	Revised for Version 1.4.1 (Applies to Release 2009bSP1)
March 2010	Online only	Revised for Version 1.5 (Applies to Release 2010a)
September 2010	Online only	Revised for Version 1.6 (Applies to Release 2010b) Renamed as *Application-Specific Verification and Validation of Models and Generated C Code*
March 2011	Online only	Revised for Version 1.6.1 (Applies to Release 2010bSP1) Renamed as *Application-Specific Verification and Validation of Models and Generated C and C++ Code*
September 2012	Online only	Revised for Version 1.6.2 (Applies to Release 2010bSP2)

Contents

1 Introduction

2 Workflow for Application- Specific Verification and Validation of Models and Generated C and C++ Code

1

Introduction

Overview

This document describes a workflow for application-specific verification and validation of models and generated C and C++ code developed using Model-Based Design with production code generation. Users of the Real-Time Workshop® Embedded Coder™ software shall carry out this workflow as part of the overall IEC 61508 or ISO 26262 software safety lifecycle. Model-Based Design enables automatic generation of production-quality code from executable graphical models that you can deploy onto embedded systems. Simulink® products from MathWorks® have become an accepted standard for Model-Based Design. Simulink, Simulink® Fixed Point™, and Stateflow® software support graphical modeling with time-based block diagrams and event-based state machines. Real-Time Workshop Embedded Coder software supports code generation for embedded systems.

If you are deploying generated code in safety-related applications, in addition to developing a model and generating code, you must apply appropriate measures and techniques to verify and validate the model and generated code. If you apply the measures and techniques in an application-specific manner, they serve the purpose of translation validation [3]. A successful translation validation indicates that output of the code generator and compiler and linker tool chain are correct for the instance of the design under consideration.

The workflow presented in this document describes a translation validation process intended to comply with applicable requirements of the overall software safety lifecycle defined by IEC 61508-3 and ISO 26262-6, as they relate to verification and validation of models and generated code. The workflow addresses risk levels that are typical for automotive applications (that is, SIL 1 – SIL 3 according to IEC 61508 and ASIL A – ASIL D according to ISO 26262).

Completing the verification and validation workflow is considered to be equivalent to the use of a certified code generation tool chain consisting of a code generator, compiler, and linker to develop the application under consideration (IEC 61508-3 clause 7.4.4.3).

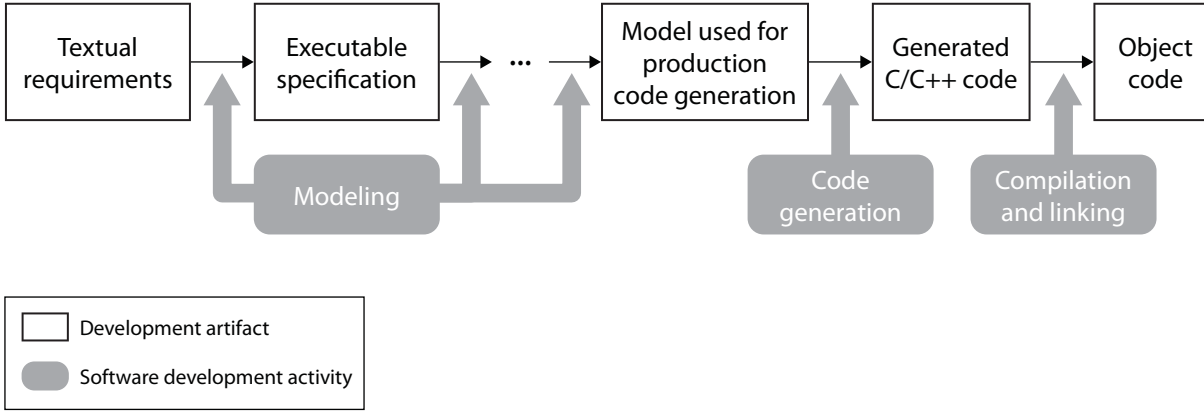
The document is organized as follows:

- "Model-Based Design for Embedded Application Software" on page 1-4 provides a brief overview on Model-Based Design for embedded application software.
- Chapter 2, "Workflow for Application- Specific Verification and Validation of Models and Generated C and C++ Code" describes the workflow for application-specific verification and validation of application software deployed in safety-related embedded systems.
- Chapter 3, "Additional Considerations" covers considerations such as tailoring and bug reporting.
- Appendix A, "Workflow Overview" summarizes the workflow in a tabular way.
- Appendix B, "Conformance Demonstration Template" contains a template that can be used to demonstrate conformance with the applicable requirements of the IEC 61508 and ISO 26262 standards.
- Appendix C, "Contextualization" contextualizes the application software development with respect to IEC 61508 terminology and concepts.
- Appendix D, "Workshare Considerations" provides examples of the division of responsibility between an OEM and a supplier when implementing the workflow for application-specific verification and validation of models and generated C and C++ code.
- Appendix E, "References" provides normative and bibliographical references for this document.
- "Glossary" on page Glossary-1 covers abbreviations and definitions of terms used in this document.

Disclaimer While adhering to the recommendations in this document will reduce the risk that an error is introduced during development and not be detected, it is not a guarantee that the system being developed will be safe. Conversely, if some of the recommendations in this document are not followed, it does not mean that the system being developed will be unsafe.

Model-Based Design for Embedded Application Software

During the development of embedded application software, you can use graphical modeling with Simulink, Simulink Fixed Point, and Stateflow to conceptualize the functionality you want to implement. By using this modeling paradigm, you can model the application software to be developed using time-based block diagrams and event-based state machines. Such a model of the application software can be simulated (executed) within the Simulink environment. The model serves as the primary representation of the application software throughout the development process, specifying functionality and design information and serving as a source for automated code generation with Real-Time Workshop Embedded Coder. In practice, this model evolution is characterized by a step-wise transformation of the application software model from an early executable specification into a model suitable for production code generation, and then finally into production-quality C or C++ code. To accomplish the transformation, you must enhance the model by adding design information and implementation details. The development process becomes the successive refinement of models followed by automatic code generation and compilation and linking, as shown in the following figure.



Model-Based Design Process: From Requirements to Code[1]

You can use the modeling notations offered by Simulink, Simulink Fixed Point, and Stateflow to achieve seamless, consistent, and efficient modeling

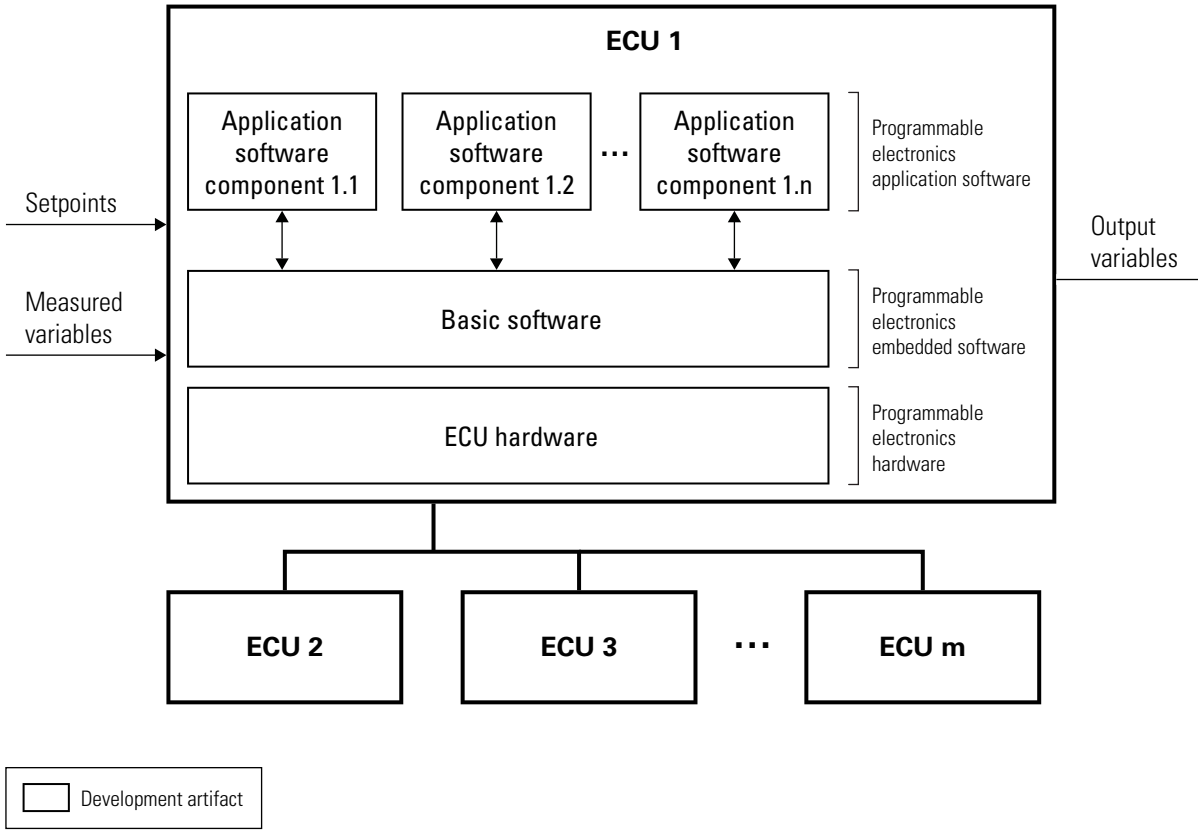
1. Solid arrows in the figure indicate the succession of software development activities.

throughout the modeling process. You can conduct different verification and validation activities at the model level (that is, early in the software lifecycle, before source code becomes available).

You can combine an application software component, developed using the preceding Model-Based Design process, with other application software components to build the application software for an embedded system[2]. The combination of basic software[3], the embedded system's hardware, and the application software provides the functionality of the embedded system. The following figure shows such a layered architecture by using the example of an automotive electronic control unit (ECU).

2. Referred to as *programmable electronics application software* in IEC 61508-3 terminology.

3. Referred to as *programmable electronics embedded software* in IEC 61508-3 terminology.



Overview of an Automotive Embedded System

2

Workflow for Application-Specific Verification and Validation of Models and Generated C and C++ Code

Workflow Overview

To fulfill objectives of IEC 61508-3 and ISO 26262-6 related to software development processes, verifying and validating the application software under development (translation validation) is required regardless of the tool chain you use.

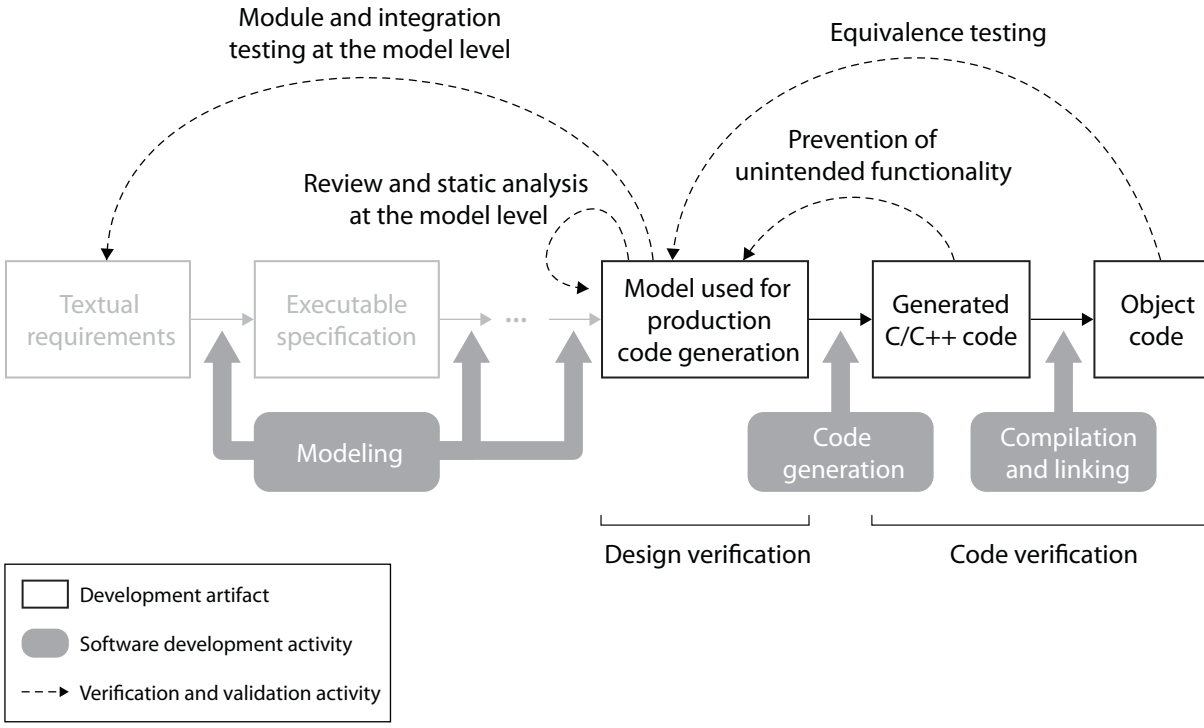
The workflow for application-specific verification and validation of models and generated C and C++ code outlined in this chapter divides the translation validation process into two steps:

1. Design verification: Demonstrate that the model used for production code generation behaves correctly with respect to its requirements.
2. Code verification: Demonstrate equivalence between the model and the corresponding object code.

The first step combines suitable verification and validation techniques at the model level, as described in "Verification and Validation at the Model Level (Design Verification)" on page 2-4. The second step mainly relies on behavioral and structural comparison between the model and the generated code, as described in "Verification and Validation at the Code Level (Code Verification)" on page 2-8.

This two-step approach allows you to complete necessary verification and validation activities, for the most part, at the model level. You can take advantage of the fact that you can reuse model-level tests that are required to verify the generated code.

The following figure shows the suggested translation validation workflow. This workflow is concerned with verifying and validating the model used for production code generation, the generated source code, and the executable object code.



Overview of the Workflow for Application-Specific Verification and Validation of Models and Generated C and C++ Code[4]

The responsibility for carrying out this workflow can rest with multiple parties, e.g., can be shared between an OEM and its supplier. The division of responsibility shall be documented.

Additional considerations to take into account when implementing the workflow are discussed in Chapter 3, “Additional Considerations”.

4. Other development artifacts are grayed out.

Verification and Validation at the Model Level (Design Verification)

In this section...

Overview

The following subsections describe a verification and validation procedure for the model used for production code generation that you can use to verify and validate the application software model before generating any code.

By using a combination of verification and validation activities at the model level, design verification serves the purpose to demonstrate that the model used for production code generation fulfills its requirements and contains no unintended functionality. An integral part of the design verification workflow is dynamic execution of the model by means of systematic simulation (model testing).

The combination of model review, static analyses, and module testing at the model level provides assurance that a model component satisfies its specification and contains no unintended functionality (that is, the model component is verified).

Reviews and Static Analyses at the Model Level

Each model component (each model subsystem considered to be a module) shall be reviewed. Model reviews should ensure readability, understandability, and testability[5].

5. Conformance with the specified requirements is being ensured by using model tests (see "Module and Integration Testing at the Model Level" on page 2-6).

Note Model reviews can be facilitated by using System Design Description reports or Web views generated by Simulink® Report Generator™. The System Design Description report describes a system design represented by a Simulink model.

Only modeling constructs that are suited for production code generation shall be used in a model used for production code generation.

Note For production code generation, it is assumed that the system target file being used is `ert.tlc`, `autosar.tlc`, or a system target file derived from `ert.tlc` or `autosar.tlc`.

Note The Simulink Block Support tables for Real-Time Workshop® and Real-Time Workshop Embedded Coder summarize code generation support for Simulink blocks. The Simulink® Verification and Validation™ Model Advisor check **"Check for questionable constructs"**, in the **By Task > Modeling Standards for IEC 61508** folder, identifies blocks not supported by code generation or not recommended for deployment.

A modeling standard (also known as design standard) shall be used for the development of safety-related application software. The modeling standard should specify good modeling practice, proscribe unsafe language features (for example, undefined language features or unstructured designs) and specify procedures for model documentation. The modeling standard shall be reviewed as fit for purpose.

Note The Simulink documentation includes "MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB®, Simulink, and Stateflow" (MAAB Guidelines) and "Modeling Guidelines for High-Integrity Systems" (High-Integrity Guidelines). The MAAB guidelines are intended to primarily improve style aspects, whereas the High-Integrity Guidelines focus on aspects specific for safety-related applications. These guidelines can be used to derive organization- or project-specific modeling standards.

As far as feasible, manual reviews shall be supported by automated static analyses on model level.

Note Adherence to a modeling standard can be partially enforced using Model Advisor checks provided with the Simulink, Real-Time Workshop, and Simulink Verification and Validation products. The Model Advisor folders **By Task > Modeling Standards for MAAB** and **By Task > Modeling Standards for IEC 61508** provided with the Simulink Verification and Validation product contain Model Advisor checks that support the MAAB guidelines and IEC 61508 respectively. The Simulink Verification and Validation product provides an API for building a custom set of checks that can be applied with the checks provided by MathWorks to produce a single analysis report. The Simulink Verification and Validation product also provides a Model Advisor Configuration Editor that can be used to configure Model Advisor checks and folders graphically.

The results of reviews and static analyses at the model level shall be documented. The procedures for corrective action on failure of reviews and analyses shall be specified.

Module and Integration Testing at the Model Level

Model component testing is intended to expose the behavior of the system and to permit a comparison with the specification. Each model component shall be tested as specified in the model component test specification. The module tests shall show that each model component performs its intended function and does not perform unintended functions.

Test vectors for testing at the model level shall be derived systematically and shall include test vectors being derived from the software requirements specification according to established criteria.

The results of module testing at the model level shall be documented.

The procedures for corrective action on failure of tests shall be specified.

Module integration testing shall be performed as module testing is completed.

Model integration stages shall be tested as specified in the model integration test specification. The tests shall show that all model components interact correctly to perform their intended function and do not perform unintended functions.

Note In some Model-Based Design approaches, an integrated model of all application software components is available in all modeling stages. In those cases, the fully integrated model of the application software is the only relevant integration stage.

The results of integration testing at the model level shall be documented, stating the test results, and whether the objectives and criteria of the test have been met. If there is a failure, the reasons for the failure shall be documented.

During model integration, any modification or change to the model shall be subject to an impact analysis to determine all modules impacted and to identify necessary reverification and redesign activities.

Verification and Validation at the Code Level (Code Verification)

In this section...

Overview

The following subsections describe a verification and validation procedure for generated code that you can use to verify and validate the generated C or C++ code and its derived executable.

By combining equivalence testing[6, 7] between the model used for production code generation and its derived executable with measures to show the absence of unintended functionality, code verification serves the purpose to demonstrate that the execution semantics of the model is being preserved during code generation.

Successful equivalence testing indicates that the output of Real-Time Workshop Embedded Coder and the compiler and linker toolchain are correct for the design instance being tested.

Equivalence Testing Model Versus Code

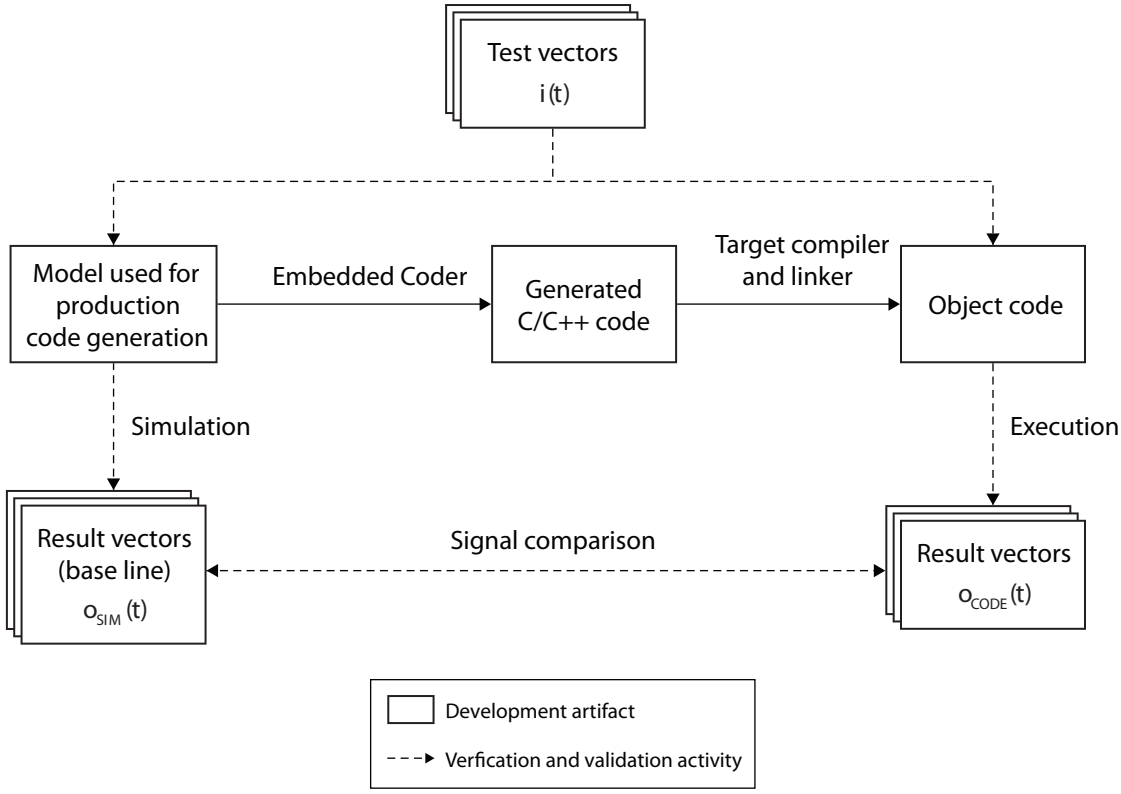
To demonstrate functional equivalence between the model and the generated code, equivalence testing shall be carried out as specified in the equivalence test specification.

In the given context, equivalence testing the model versus the generated code refers to the stimulation of both the model used for production code

6. Referred to as back-to-back testing in ISO 26262-6 terminology.

7. Not to be confused with equivalence class testing according to IEC 61508-3 tables B.2 and B.3.

generation and the executable derived from the generated code with identical test vectors. In this setting, Real-Time Workshop Embedded Coder software and the compiler and linker toolchain are assumed to be a black box. The validity of the translation process (whether or not the semantics of the model has been preserved) is determined by comparing the system responses (result vectors of the model and the generated code) resulting from stimulation with identical timed test vectors $i(t)$. More precisely, the simulation results of the model used for production code generation $o_{SIM}(t)$ are compared with the execution results of the generated and compiled production code $o_{CODE}(t)$ shown in the following figure.



Equivalence Testing Approach

As far as feasible, equivalence testing shall be automated by tools.

Tools used for equivalence testing shall be validated (for example, by applying positive and negative dynamic tests).

The results of equivalence testing shall be documented.

The procedures for corrective action on failure of tests shall be specified.

The following sections provide detailed information on the equivalence testing procedure:

- "Equivalence Test Vector Generation" on page 2-10
- "Equivalence Test Execution" on page 2-13
- "Signal Comparison" on page 2-16

Equivalence Test Vector Generation

A valid translation requires execution of the object code to exhibit the same observable effects as model simulation for any given set of test vectors.

Since complete testing is impossible for complexity reasons, stimuli (test vectors) shall be sufficient to cover the different structural parts of the model.

To assess the model coverage achieved, at Safety-Integrity Levels (SIL) 2 and above [4], as well as at all Automotive Safety Integrity Levels (ASIL), some test coverage metric shall be visible. The extent and scope of structural model coverage needs to be increased for the higher SILs and ASILs. The following table[8] lists coverage metrics that can be used to analyze model coverage.

8. Tables are in the style of the IEC 61508-3 software safety integrity tables. The meaning of the symbols is as follows: ++ … The technique or measure is highly recommended for this SIL. If this technique or measure is not used, then the rationale behind not using it should be detailed during safety planning and agreed to by the assessor; + … The technique or measure is recommended for this SIL, but is a lower recommendation than a ++ recommendation; o … The technique or measure has no recommendation for or against being used.

Model Coverage Analysis

Technique or Measure		IEC 61508			ISO 26262			
		SIL1	SIL2*	SIL3	ASIL A	ASIL B	ASIL C	ASIL D
1a	Decision coverage	o	++	++	+	++	++	++
1b	Condition coverage	o	+	+	+	+	+	+
1c	Modified condition/decision coverage	o	+	+	+	+	+	++
2	Lookup table coverage	o	o	+	o	o	+	+

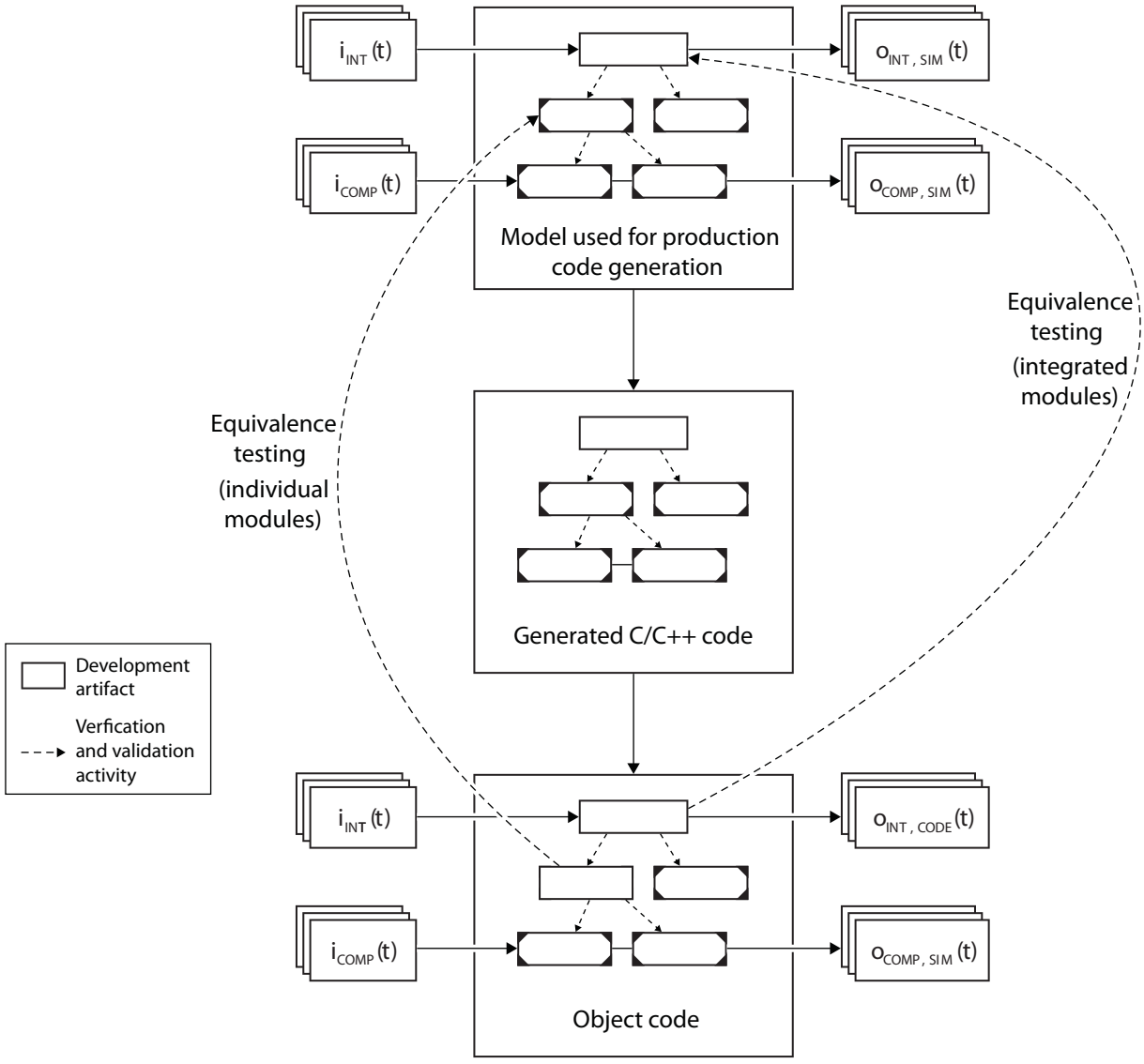
* For SIL 2 coverage, analysis can be focused on complex algorithms/structures.

Note The Simulink Verification and Validation software can be used to perform model coverage analysis.

Note Lookup table coverage can be used as an additional means to ensure coverage for models containing Lookup Table blocks.

Test vectors resulting from requirements based testing (see "Module and Integration Testing at the Model Level" on page 2-6) can be reused for equivalence testing.

The following figure illustrates equivalence testing of individual and integrated modules. $i_{COMP}(t)$ and $o_{COMP}(t)$ refer to the test vectors and result vectors for model components respectively; $i_{INT}(t)$ and $o_{INT}(t)$ refer to the test vectors and result vectors for the integration stages, respectively.



Equivalence Testing of Individual and Integrated Modules

If the coverage achieved with the existing test vectors is not sufficient, additional test vectors shall be created.

If full coverage for the selected metrics cannot be achieved, the uncovered parts shall be assessed and justification for uncovered parts shall be provided.

Note In practice, the set of test vectors can be iteratively extended using model coverage analysis until the mandated level of model coverage has been achieved.

Note Simulink® Design Verifier™ can be used to create additional test vectors for equivalence testing.

Equivalence Test Execution

The test vectors for equivalence testing shall be used to stimulate both the model used for production code generation and the executable derived from the generated code.

The resulting object code shall be tested in an execution environment that corresponds as far as possible to the target environment to which the code will be deployed.

Note

- The resulting object code can be executed on the target processor or on a target-like processor, for example, by means of a processor-in-the-loop (PIL) simulation (PIL verification), or simulated by means of an instruction set simulator (ISS) for the target processor (ISS verification). If feasible, PIL verification is preferred.
- To facilitate PIL verification, the PIL connectivity API provided with the Real-Time Workshop Embedded Coder product can be leveraged. This API allows to integrate custom Top-model and Model block PIL implementations by integrating custom build, download and communications processes.
- To automate equivalence testing, the Code Generation Verification API provided with the Real-Time Workshop Embedded Coder product can be leveraged.

9. This option has no effect when the compiler and target CPU both represent floating-point zero with the integer bit pattern 0.

Note Some optimization and code generation settings may affect the consistency between simulation and code generation. To ensure consistency between simulation and PIL verification / ISS verification:

- On the **Optimization** pane of the Configuration Parameters dialog box, clear the **"Remove root level I/O zero initialization"** check box, or set the parameter `ZeroExternalMemoryAtStartup` to `on`.
- On the **Optimization** pane of the Configuration Parameters dialog box, clear the **"Remove internal data zero initialization"** check box, or set the parameter `ZeroInternalMemoryAtStartup` to `on`.
- On the **Optimization** pane of the Configuration Parameters dialog box, clear the **"Use memset to initialize floats and doubles to 0.0"**[9] check box, or set the parameter `InitFltsAndDblsToZero` to `on`.
- On the **Optimization** pane of the Configuration Parameters dialog box, clear the **"Remove code from floating-point to integer conversions that wraps out-of-range values"**[10] check box, or set the parameter `EfficientFloat2IntCast` to `off`.
- On the **Optimization** pane of the Configuration Parameters dialog box, clear the **"Remove code that protects against division arithmetic exceptions"** check box, or set the parameter `NoFixptDivByZeroProtection` to off.
- On the **Real-Time Workshop > Interface** pane of the Configuration Parameters dialog box, clear the **"Terminate function required"** check box, or set the parameter `IncludeMdlTerminateFcn` to `off`.

If the execution of the resulting object code is not carried out in the target environment, differences between the testing environment and the target environment shall be analyzed to ensure that they do not adversely alter the results.

10. This option does not affect consistency between simulation and code generation for in-range values.

If the model for the application software contains functionality used for simulation, but not for production code generation[11] the user shall take additional measures to ensure that modeled parts and their replacements exhibit the same behavior with respect to functionality, timing, and so on.

Signal Comparison

After test execution, the result vectors (simulation results) of a model $o_{SIM}(t)$ shall be compared with the execution results of the generated code $o_{CODE}(t)$.

The simulation results of the model are used as baseline. They are matched with the result vectors obtained from execution of the object code.

Even in the case of a correct translation of a Simulink and Stateflow model into C or C++ code, one cannot always expect identical behavior (equality). Possible reasons include limited precision of floating point numbers, quantization effects when using fixed point arithmetic, and differences between compilers. So, the definition of correctness must be based on sufficiently similar behavior (sufficient similarity).

A suitable signal comparison algorithm shall be selected that is able to tolerate differences between the result vectors representing the system responses $o_{SIM}(t)$, and $o_{CODE}(t)$.

Note Simple comparison algorithms, such as absolute difference, can be used, as well as more elaborated algorithms, such as the difference matrix method [2]. To automate the inspection and comparison of logged result data, the Simulation Data Inspector tool provided with Simulink can be used.

Two result vectors are sufficiently similar if their difference with respect to a given comparison algorithm is less or equal a given threshold. The selection of the comparison algorithm and the definition of the threshold value depend on the application under consideration and need to be documented.

11. For example, a Stateflow® model of the scheduler will be replaced by the actual scheduler of the real-time operating system.

Note Simulink Fixed Point allows performing bit-true simulations of model portions implemented using fixed-point arithmetic to observe the effects of limited range and precision on designs built with Simulink and Stateflow. When used with Real-Time Workshop Embedded Coder, Simulink Fixed Point enables generating pure integer C code from these model portions. The generated code is in bit-true agreement with the model used for production code generation, ensuring that the implemented design will perform exactly as it did during simulation.

Sufficient similarity serves as a basis for the definition of functional equivalence. Stating the correct model-to-code translation, a model and the code generated from it are functionally equivalent if the simulation of the model and the execution of the executable derived from the generated code lead to sufficiently similar result vectors and both are stimulated with identical test vectors.

Prevention of Unintended Functionality

By the use of at least one of the following measures it shall be demonstrated that the generated C or C++ code does not perform any unintended function:

- Model and code coverage comparison
- Traceability review

Prevention of Unintended Functionality

Technique or Measure		IEC 61508			ISO 26262			
		SIL1	SIL2	SIL3	ASIL A	ASIL B	ASIL C	ASIL D
1a	Model and code coverage comparison	o	+	++	o	+	++	++
1b	Traceability review	o	+	++	o	+	++	++

If model and code coverage comparison is being used, structural code coverage shall be measured during equivalence testing and compared with the structural model coverage.

Note During a SIL or PIL simulation, code coverage metrics for the generated source code can be collected using a third-party tool. Real-Time Workshop Embedded Coder supports the BullseyeCoverage™ tool from Bullseye Testing Technology™.

To be meaningful, structural coverage metrics comparable with each other should be used at the model and code level, respectively.

Note According to [1], decision coverage on model level and branch coverage (C1) on code level, which is also sometimes termed decision coverage, can be used in combination.

Discrepancies between model and code coverage with respect to comparable metrics shall be assessed. If the code coverage achieved is less than the model coverage, unintended functionality could have been introduced.

If a traceability review is being used, a limited review of the generated C or C++ code shall be performed to ensure that all parts of the generated source code can be traced back to the model.

Note Auto generated Simulink block / Stateflow object comments can be used to include tracing information in the generated code.

Note The Traceability Report section of the Real-Time Workshop Embedded Coder code generation report helps to provide a complete mapping between model blocks and the generated code. It lists eliminated and virtual blocks versus traceable blocks. Selecting the **"Code-to-model"**traceability option generates hyperlinks within the displayed source code to view the blocks or subsystems from which the code was generated. Selecting the **"Model-to-code"** traceability option allows you to view the generated code for any block in the model.

Note The traceability matrix export feature provided with the IEC Certification Kit can be used to export customized traceability information into a Microsoft® Excel® spreadsheet. Traceability information can be analyzed and commented in Excel®. For more information, see "Generating a Traceability Matrix" in the *IEC Certification Kit User's Guide*.

Nontraceable code shall be assessed.

The results of demonstrating the absence of unintended functionality in the generated code shall be documented.

The procedures for corrective action in case of the detection of unintended functionality shall be specified.

Hand Coded Portions within the Generated Code

If the generated C or C++ code contains hand-coded portions[12], the user shall take additional measures to verify and validate these code portions according to IEC 61508-3 or ISO 26262-6, respectively. Special attention shall be paid to the interfaces between hand code and generated code.

Note The necessary verification and validation activities for the hand-coded portions may vary depending on the extent of the hand code.

The hand code shall be traceable to the software requirements specification.

The results of verifying and validating the hand code shall be documented.

The procedures for corrective action in case of the detection of errors shall be specified.

12. For example, code introduced by user S-functions.

3

Additional Considerations

When implementing the workflow for application-specific verification and validation, consider

Configuration Management and Revision Control

Revision control information (version numbers) shall be used to identify the artifacts to be verified or validated.

At SIL 2 and above, configuration management shall be applied to the artifacts to be verified or validated [4]. At all ASILs, configuration management shall be applied to the artifacts to be verified or validated, as well as all other work products specified in ISO 26262 or in this document.

The following tables lists measures and techniques for configuration management and revision control.

Configuration Management and Revision Control

Technique/Measure	IEC 61508			ISO 26262			
	SIL1	SIL2	SIL3	ASIL A	ASIL B	ASIL C	ASIL D
1 Inclusion of revision control information into models and generated code	+	++	++	++	++	++	++
2 Computer aided configuration management	o	+	++	++	++	++	++

Note You can use a Simulink Model Info block to display revision control information about a model as an annotation block in the model. The Model Info block can show revision control information embedded within the model or information maintained by an external revision control or configuration management system [5].

Note If you use a source control system to manage files, you can use the Source Control Interface to interface with the system to perform source control actions from within MATLAB, Simulink, and Stateflow [5].

Adequate Competency of the Project Team

Those carrying out modeling, code generation, and related translation validation activities shall be competent for the activities undertaken.

Installation Integrity and Release Compatibility

The tool user shall validate any modifications or additions made to shipping products.

Note You can use the `ver` command in MATLAB to display the current versions of MATLAB, Simulink, Real-Time Workshop Embedded Coder, and other MathWorks products.

The Summary section of the Real-Time Workshop Embedded Coder code generation report lists version and date information, and Simulink model settings.

Note You can use installation integrity techniques to verify the integrity of the installed products.

The tool user shall ensure that the product versions used to create the model are compatible with the product versions used for code generation and related translation validation activities.

Note The version of Simulink software last used to modify the model is contained in the Model section of the model file.

You can find information on saving a model in earlier formats, opening a model originally created in an older version of Simulink, or updating a model to be compatible with the current Simulink version in the Simulink documentation.

Bug Reporting

The tool user shall assess bug report information provided by the tool vendors and comply with the recommendations and workarounds, if applicable.

Issues with MathWorks products shall be reported.

Bug report information shall also be assessed by the tool user on a regular basis, after deployment of the application under development. This is necessary to carry out corrective actions if deployed applications are affected by bugs in the tools identified after deployment.

Note You can use the bug reports section of the MathWorks web site `www.mathworks.com/support/bugreports` to view and report bugs related to MathWorks products.

Deviation from the Reference Workflow

In some instances it might be necessary to deviate from the reference workflow for application-specific verification and validation explained in this document. For example, due to legacy processes and tools, an organization might choose to rely solely on code coverage instead of model and code coverage.

- A formal deviation procedure shall be used to document and justify deviations from the workflow rather than an individual developer having discretion to deviate at will.

- The requirements for code verification can be adjusted in consultation with the assessor if an appropriate application-independent test suite for Real-Time Workshop Embedded Coder is used as an additional means to verify and validate the code generator.

Note Possible credits for using such a test suite include reducing or waiving the requirements related to the prevention of unintended functionality and/or the validation of equivalence testing tools.

Integration with the Software Safety Lifecycle

The application-specific verification and validation activities shall be integrated with the overall software safety lifecycle for the application under consideration.

A

Workflow Overview

Objectives, Prerequisites, and Work Products

Activity	Objective	Prerequisites	Work Products
Verification and Validation at the Model Level (Design Verification)	To demonstrate that the model used for production code generation fulfills its requirements and contains no unintended functionality	• Software requirements specification • Model for production code generation • Modeling standard • Model component test specification • Model integration test specification • Procedures for corrective action	• Model review report • Model component test results • Model integration test results • Verified and tested model components • Verified and tested model for production code generation
Verification and Validation at the Code Level (Code Verification)	To demonstrate that the execution semantics of the model is being preserved during code generation	• Model used for production code generation • Source code • Object code • Equivalence test specification • Procedures for corrective action	• Equivalence test results • Results of measures demonstrating the absence of unintended functionality • Verified and tested source code • Verified and tested object code

B

Conformance Demonstration Template

To justify that you have correctly selected and satisfied the requirements outlined in this document, it is necessary to provide a documented assessment. The IEC Certification Kit product provides an editable Conformance Demonstration Template that can be used to demonstrate conformance with the parts of IEC 61508-3 or ISO 26262-6 covered in this document. The template can be found at:

matlabroot/toolbox/qualkits/iec/rtwec/r2010bSP2/certkitiec_rtwec_cdt.rtf

For each technique or measure:

- In the third column, state to what degree you applied the technique or measure for the application under consideration by using one of the phrases Used, Used to a limited degree, or Not used.
- In the fourth column, state how you used the technique or measure in the application under consideration. If the reference workflow includes alternative means for compliance, indicate what variant you used. In addition, enter a reference to the document (for example, test report or review documentation) that satisfies the requirement.

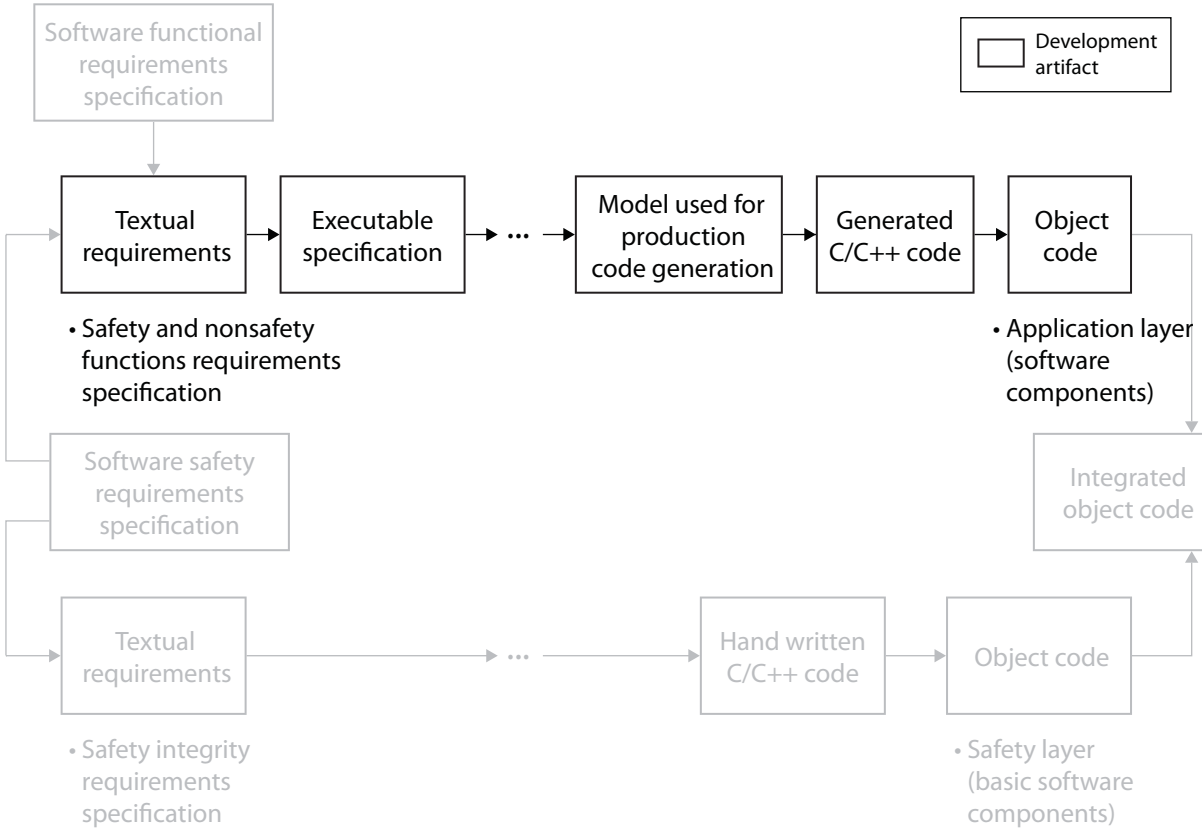
C

Contextualization

This section contextualizes concepts and workflows for application software development presented in this document with respect to terminology and concepts used in the IEC 61508-3 software safety lifecycle, as shown in the figure below.

This document assumes that the application layer that implements safety functions and that might implement nonsafety functions is developed using Model-Based Design. Since the safety layer that realizes the safety integrity requirements is beyond the scope of this document, no assumption is made about the software development paradigm used for the safety layer. Typically, the safety layer would be implemented with traditional development approaches, using hand coding. However, it is not excluded that basic software components implementing the safety layer are also (partially) developed by using Model-Based Design.

The integrated object code will be validated against the software safety requirements specification and, if applicable, against the relevant parts of the software functional specification, as part of the software safety validation process.



Development of Application and Safety Layers

D

Workshare Considerations

Overview

The responsibility for carrying out the workflow for application-specific verification and validation of models and generated C and C++ code can rest with multiple parties, e.g., can be shared between an OEM and its supplier.

This section addresses additional considerations that apply when application software development is distributed, and provides examples for the division of responsibility between an OEM and a supplier when implementing the workflow.

Distributed Development

During application software development, the responsibility for verifying and validating models and generated C or C++ code can rest with multiple parties, e.g., can be shared between an OEM and its supplier. Distributed development might lead to deviations from the reference workflow.

When using distributed development:

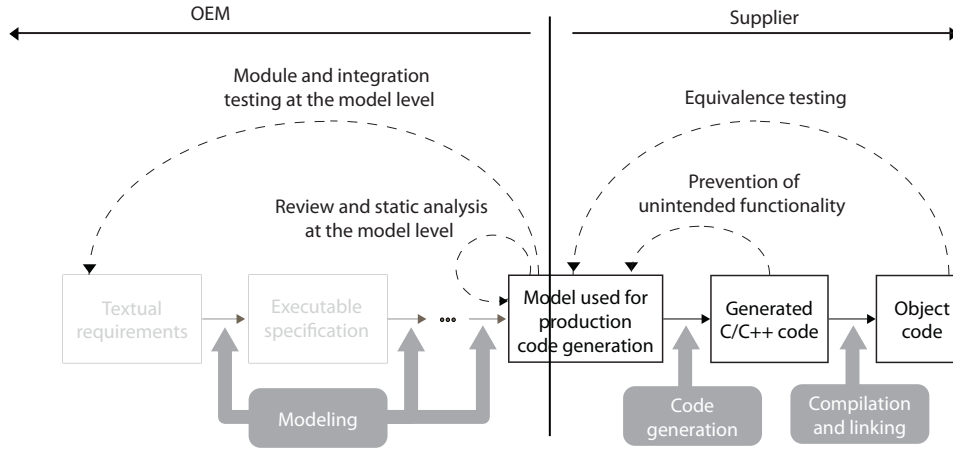
- The workshare and division of responsibility shall be agreed upon and documented.

Note The workshare and division of responsibility can be documented within the Development Interface Agreement (DIA) document.

- Deviations from the reference workflow shall be documented and justified as outlined in “Deviation from the Reference Workflow” on page 3-4.
- Each party is responsible for complying with the tool certification or qualification requirements for the tools they are using.

Workshare Examples

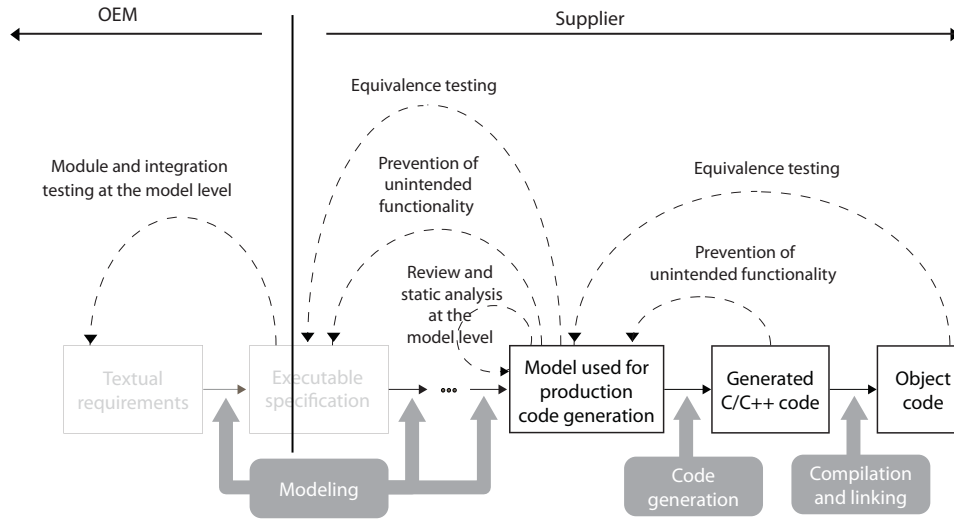
In Example 1 below, the OEM is responsible for creating the model used for production code generation and carries out the model verification portion of the reference workflow. The OEM hands over the model used for production code generation to the supplier. The supplier is responsible for creating the source code and the object code. The supplier carries out the code verification portion of the reference workflow.



Example 1: OEM/Supplier Workshare

In Example 2, the OEM creates the executable specification and carries out the testing activities at the model level. The OEM hands over the executable specification to the supplier. The supplier is responsible for all subsequent stages of the model elaboration process, as well as for creating the source code and object code. The executable specification and the model used for production code generation are compared using equivalence testing and additional measures to prevent unintended functionality are carried out.

In Example 2, the code verification portion of the reference workflow remains unchanged and is carried out by the supplier, while the model verification portion deviates from the reference workflow.



Example 2: OEM/Supplier Workshare

Usage of the Conformance Demonstration Template

You can adapt the Conformance Demonstration Template to demonstrate compliance in the case of distributed development by:

- Adding or modifying rows in the conformance demonstration template to account for deviations from the techniques and measures described in the reference workflow.
- Augmenting the resulting, updated conformance demonstration template with additional columns containing a responsibility assignment matrix for the parties involved.

The responsibility assignment matrix indicates for each listed Technique / Measure which party is Responsible (R), Accountable (A), Consulted (C), or Informed (I). The following figure illustrates the structure of an augmented conformance demonstration template.

	Technique / Measure	Associated Requirements	Used / Used to a limited degree / Not used	Interpretation in this application, Evidence	Workshare / Responsibilities	
					OEM Responsible / Accountable / Consulted / Informed	Supplier Responsible / Accountable / Consulted / Informed
1	Model review (See "Reviews and Static Analyses at the Model Level")	• Inclusion of all model components			R, A	
2	Adherence to modeling standard (See "Reviews and Static Analyses at the Model Level")	• Designation of a modeling standard • Review the modeling standard as fit for purpose • Restriction to modeling constructs suited for production code generation • Evidence for using the modeling standard			R, A	

Conformance Demonstration Template with Responsibility Assignment Matrix

E

References

Normative References

IEC 61508-1:2010. International Standard IEC 61508 Functional safety of electrical/electronic/programmable electronic safety related systems – Part 1: General requirements. Second edition, 2010

IEC 61508-3:2010. International Standard IEC 61508 Functional safety of electrical / electronic / programmable electronic safety-related systems – Part 3: Software requirements. Second edition, 2010

ISO 26262-6:2010. ISO 26262-6 Road vehicles — Functional safety — Part 6: Product development: software level. Final Draft International Standard, 2010

ISO 26262-8:2010. ISO 26262-8:2010 Road vehicles — Functional safety — Part 8: Supporting processes. Final Draft International Standard, 2010

References

[1] A. Baresel, M. Conrad, S. Sadeghipour, J. Wegener: The Interplay between Model Coverage and Code Coverage. 11. Europ. Int. Conf. on Software Testing, Analysis and Review (EuroSTAR 03), Amsterdam, NL, Dec. 2003.

[2] M. Conrad, S. Sadeghipour, H.-W. Wiesbrock: Automatic Evaluation of ECU Software Tests. SAE 2005 Transactions, Journal of Passenger Cars - Mechanical Systems, SAE International, March 2006

[3] A. Pnueli, M. Siegel, E. Singerman: Translation Validation. Lecture Notes in Computer Science, Vol. 1384, pp. 151-166. Springer, 1998

[4] D. J. Smith, K. G. L. Simpson: Functional Safety – A straightforward guide to applying IEC 61508 and related standards. 2nd edition, Elsevier Butterworth-Heinemann, 2005

[5] G. Walker, J. Friedman, and R. Aberg: Configuration Management of the Model-Based Design Process, Proc. SAE World Congress 2007, Detroit, MI, USA, April 2007

Glossary

Activities at the code level
: Verification, validation, and testing activities carried out or based on the generated code for the embedded application software or the executable derived from the generated code.

Activities at the model level
: Verification, validation, and testing activities carried out or based on the model of the embedded application software.

API
: Application Programming Interface

ASIL
: Automotive Safety Integrity Level

AUTOSAR
: AUTomotive Open System ARchitecture

Embedded application software
: Software components carrying out control and monitoring functions in an embedded system.

Executable specification
: Model of the embedded application software used in the early phases of software development to conceptually anticipate the functionality to be implemented. Executable, graphical model representing the initial stage of the model evolution process.

  In the course of the software lifecycle design information and implementation details will be added and resulting in the model used for production code generation.

IDE
: Integrated Development Environment

ISS
: Instruction Set Simulator

Model component

Subsystem of the model of the embedded application software considered to be a module.

Model used for production code generation

Model of the embedded application software used as input for production code generation, using Real-Time Workshop Embedded Coder. Executable, graphical model representing the final stage of the model evolution process.

The model used for production code generation might use fixed-point or floating-point arithmetic. Typically, fixed-step solvers, which represent a necessary prerequisite for efficient code generation, are used for simulating the model used for production code generation.

The model used for production code generation typically implements safety and nonsafety (standard) functions, but does not implement safety-integrity requirements.

PIL

Processor-In-the-Loop

SIL

Safety-Integrity Level or Software-In-the-Loop.

Index

N

P

R

S

T

U

V

W