

Code Generation Verification – Assessing Numerical Equivalence between Simulink Models and Generated Code

Mirko Conrad, Tom Erkkinen, Thomas Maier-Komor, Guido Sandmann, Marty Pomeroy

Abstract

With the advent of Model-Based Design and production code generation, the ability to demonstrate that the generated object code correctly implements the model used for code generation, i.e. to show the correctness of the model-to-code transformation result, becomes inherent.

Various approaches to verify the model-to-code translation result in practice, i.e., to efficiently and effectively assess the correctness of translating real world Simulink models into embedded C code, have been proposed and applied by engineering professionals. Unfortunately, many of these approaches do not exploit the advantages of Model-Based Design and more or less carry out the translation validation of generated code in the same manner as for hand-written code.

In this paper, the authors will present *Code Generation Verification (CGV)* an automated, testing-based approach to assess the numerical equivalence between Simulink models and the generated code. In the CGV approach, each individual model-to-code translation is followed by a verification phase to assess that the input (i.e., the Simulink model used for code generation) and the output (i.e., the target code produced during code generation and compilation) of this translation produce the same numerical results when stimulated with identical inputs.

The paper describes the proposed verification approach, and illustrates it by using Simulink and Real-Time Workshop Embedded Coder. It also discusses this approach in the context of high-integrity application development by embedding it into code verification workflows for IEC 61508 and ISO 26262.

1. Introduction

The increasing utilization of *embedded software* in automotive applications resulted in an increasing complexity that has proven difficult to manage with conventional design approaches. *Model-Based Design* has become an established software engineering paradigm for the development of embedded automotive software because of its capability to address the corresponding software complexity and productivity challenges. These gains are now being realized broadly, including in high-integrity applications.

The advent of Model-Based Design has introduced a new artifact into the software life cycle between the requirements and the code: the *model*. In a typical Model-Based Design workflow, an initial *executable graphical model* representing the software component under development is refined and augmented until it becomes the blueprint for the final implementation through automatic *code generation*.

Regardless of the software engineering paradigm utilized, the automotive software being developed needs to be *verified and validated* to detect errors and gain confidence in its correct functioning. In principle, verification and validation of embedded software developed using Model-Based Design and code generation could be carried out in the same way as for manually written code. But such an approach would not leverage the benefits of Model-Based Design regarding process efficiency. Introducing executable models into the embedded software lifecycle brings an opportunity to apply automatic verification and validation techniques earlier in the software lifecycle, at a higher level of abstraction with test reuse. With respect to process efficiency, engineers should make the most of this opportunity.

Translation validation [1] is considered a preferred means to carry out verification and validation of generated code in an engineering context [2, 3, 4]. We use the term translation validation to refer to verification / validation approaches where each individual model-to-code translation is followed by a validation phase that verifies the target code produced on this translation (i.e. code generation / compilation) properly implements the submitted source model. The authors are proposing an approach, where dynamic *testing for numerical equivalence* between models and generated code constitutes the core part of the translation validation process. From the authors' experience, a testing-based translation validation approach is a scalable solution in an engineering context. It affords greater implementation flexibility in terms of code optimization options and target hardware selection.

Simulink® [5] is a widely used Model-Based Design environment. It supports the description and development of models in a graphical block-diagram language. Simulink models can also include state-machine diagrams using *Stateflow*® [6], or imperative code written in *Embedded MATLAB*TM [7]. The Simulink environment provides extensive capabilities for simulation. Embedded code generation from Simulink models is supported by *Real-Time Workshop*® *Embedded Coder*TM [8]. The code generator translates models into production-quality C code that can be deployed onto embedded systems. Simulink is extensively used in different application domains. According to [9], 50% of the behavioural models for automotive controls are designed using the Simulink environment. Due to its popularity, Simulink is used throughout this paper to illustrate the proposed approach.

The paper presents *Code Generation Verification (CGV)* an automated, testing-based translation validation approach to assess the numerical equivalence between Simulink models and the generated code. The paper describes the proposed verification approach, and illustrates it using Simulink and Real-Time Workshop Embedded Coder. It also discusses this approach in the context of high-integrity application development by embedding it into code verification workflows for IEC 61508 [10] and ISO 26262 [11].

2 Translation Validation using CGV

The need to demonstrate the correctness of the model-to-code transformation process can be approached from two directions. In principle, correctness can be shown upfront for all possible translation runs (translator verification) or it can be shown for each individual translation run that is of interest (translation validation). This paper focuses on the translation validation aspect. The use of systematic dynamic testing techniques based on models to carry out translation validation (*translation testing*) has been proposed e.g. in [2, 4, 12].

The *Code Generation Verification (CGV) approach* utilizes translation testing to demonstrate that the execution semantics of the model are preserved during production code generation, compilation and linking (Fig 1).

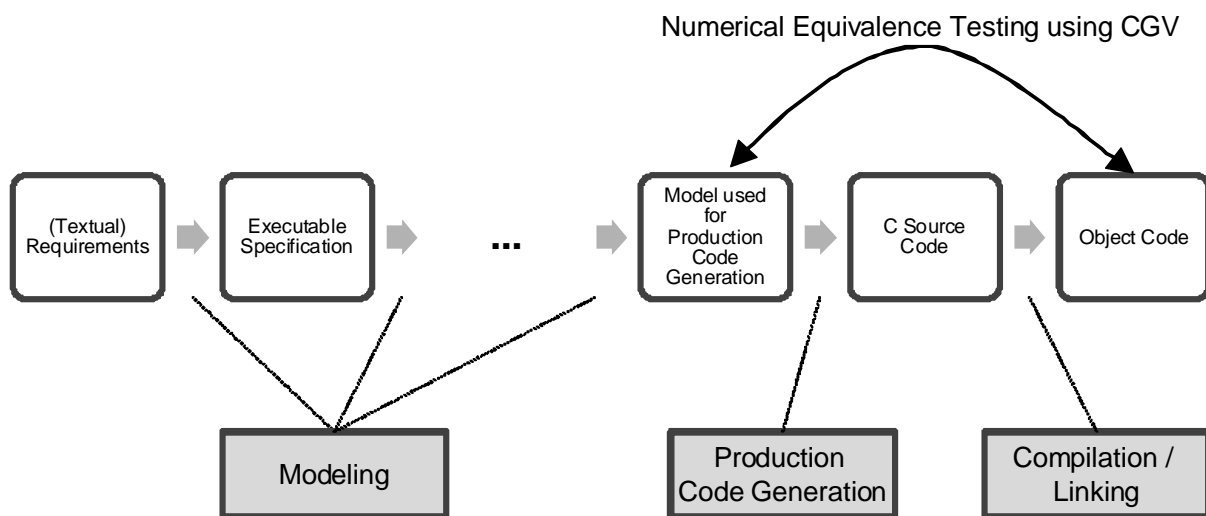


Fig 1: Model-Based Design with Numerical Equivalence Testing

Equivalence testing is performed to demonstrate numerical equivalence between the model used for production code generation (M_{PCG}) and the executable derived from the generated, cross-compiled C code. This type of equivalence testing between the model used and the resulting object code (also known as *comparative testing* or *back-to-back testing*) constitutes the backbone of the proposed translation testing approach¹.

Equivalence testing refers to the execution of the model used for code generation and the object code derived from it via code generation and compilation with the same input stimuli (test vectors) followed by a numerical comparison of the outputs (result vectors). The validity of the translation process, i.e. whether or not the semantics of the model have been preserved during code generation, compilation and linking, is determined by comparing the *system reactions* (result vectors, which are the outputs resulting from stimulation with identical timed test vectors $i(t)$) of the model and the generated code. More precisely, the simulation results of the model used for

¹ Numerical equivalence testing can should be combined with other code verification activities e.g. to show the absence of unintended functionality, the absence of run-time errors or conformance to a coding standard such as MISRA-C.

production code generation $o_{M_PCG}(t)$ are compared with the execution results of the generated and compiled production code $o_{CODE}(t)$.

Fig. 2 summarizes the suggested approach. In depth discussions of equivalence testing procedures can be found in [13, 14].

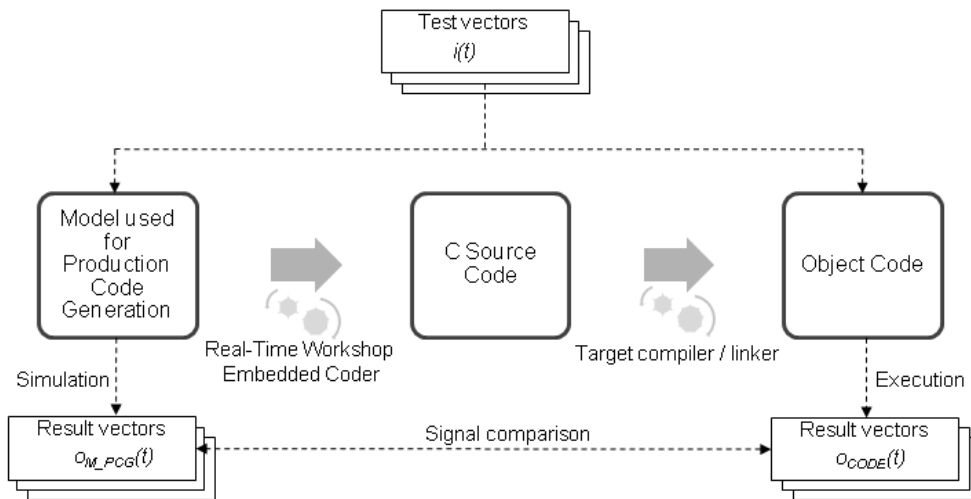


Fig. 2: Numerical equivalence testing approach

Numerical equivalence testing is unique in that the expected outputs for the test vectors do not have to be provided [15]. This makes equivalence testing ideally suited to automation.

The following subsections provide detailed information on the equivalence testing procedure.

2.1 Test Vector Design

A valid model-to-code-translation requires that the execution of the object code exhibits the same observable results as the simulation of the model for any given set of test vectors. Since complete testing is impossible due to complexity reasons, a suitable subset of stimuli (test vectors) needs to be selected and used. The chosen subset should sufficiently cover the different structural entities of the model, i.e. achieve sufficient model coverage.

Test vectors gained from requirements-based testing at the model level can be reused for equivalence testing, cf. [12]. To assess the structural coverage reached by using requirements-based test vectors, model coverage metrics should be utilized.

If the coverage achieved with the existing test vectors is not sufficient w.r.t. the selected model coverage metrics, additional test vectors that execute the model elements not covered shall be created. In practice, the user can iteratively extend the set of test vectors using model coverage analysis until the chosen level of model coverage has been achieved. If full coverage for the selected metric(s) cannot be achieved, the uncovered parts shall be assessed and justification for uncovered parts shall be provided.

Tool support: Model coverage analysis for Simulink models can be performed by using the Model Coverage Tool in Simulink® Verification and Validation™ [16]. The test generation capability of Simulink® Design Verifier™ [17] can be used to create additional test vectors for equivalence testing. Bidirectional traceability between requirements and test case can be analyzed using the Requirements Management Interface of Simulink Verification and Validation.

2.2 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 the code will be deployed to. The resulting object code can be either executed on the target processor or on a target-like processor, e.g. by means of a Processor-in-the-Loop simulation (PIL verification), or simulated by means of an instruction set simulator for the target processor (ISS verification). If feasible, PIL verification is the preferred approach.

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 in order to make sure that they do not adversely alter the results.

Tool Support: The code generation verification (CGV) capability of Real-Time Workshop Embedded Coder can be used to execute a model and the generated code in a Software-in-the-Loop (SIL) or PIL environment. The MATLAB script in Fig. 3 illustrates the usage of CGV to compare simulation and SIL results². After configuring the model to be compatible with SIL testing, the model is stimulated using pre-existing input data and model parameter values. In the second execution of the same model, input data and parameters are used for SIL testing.

```
cgvModel = 'rtwdemo_cgv';
load_system(cgvModel);

cgvCfg = cgV.Config(cgvModel, 'connectivity', 'sil', 'SaveModel',
'on');
cgvCfg.configModel();

% Executing the model
cgvSim = cgV.CGV(cgvModel, 'connectivity', 'sim');
cgvSim.addInputData(1, [cgvModel '_data']);
cgvSim.addPostLoadFiles({[cgvModel '_init.m']});
cgvSim.setOutputDir('cgv_output');
simresult = cgvSim.run();
% Executing the generated code
cgvSil = cgV.CGV(cgvModel, 'Connectivity', 'sil');
cgvSil.addInputData(1, [cgvModel '_data']);
cgvSil.addPostLoadFiles({[cgvModel '_init.m']});
```

² The script is based on the R2010a release of the Simulink product family.

```

cgvSil.setOutputDir('cgv_output');
silresult = cgvSil.run();

% Checking for successful execution of both runs
if ~simresult || ~silresult2
    error('Execution of model failed.');
```

Fig. 3: Equivalence test execution using CGV (code example)

2.3 Result Vector Comparison

After test execution, the result vectors (*simulation results*) of the model $O_{M_PCG}(t)$ shall be compared with the execution results of the generated code $O_{CODE}(t)$. The simulation results of the model M_{PCG} are used as baseline. They are compared with the result vectors obtained from execution of the object code.

Even in the case of a correct translation of a Simulink model into C code, one cannot always expect identical behavior (equality). Possible reasons include limited precision of floating point numbers, target optimized code constructs, quantization effects when using fixed point math and compiler dependent behavior. So, the comparison may need to be able to tolerate limited differences between the result vectors, i.e. it has to be based on sufficiently similar behavior (sufficient similarity).

The user needs to select a suitable *signal comparison algorithm* that is able to tolerate differences between the result vectors representing the system responses $O_{M_PCG}(t)$ and $O_{CODE}(t)$. There is a broad variety of potential comparison algorithms ranging from elementary algorithms like absolute or relative differencing to more elaborate ones such as the difference matrix method, see [18] for an overview of algorithms to consider.

Two result vectors are *sufficiently similar* if their difference w.r.t. a given comparison algorithm is less than or equal to a user-defined threshold. The selection of the comparison algorithm and the definition of the threshold value depend on the application under consideration or even on the characteristics of a given output signal (comparing boolean outputs, for example, may require different algorithms and thresholds than comparing floating point outputs with potentially denormalized values). In practice, an initial automatic assessment with low thresholds and a subsequent manual review of the discrepancies has been successful.

Tool support: Simulink® Fixed Point™ [19] allows one to perform bit-true simulations of model portions implemented using fixed point math 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 generation of pure integer C code from these model portions. The generated code can be assessed to determine if the implemented design will perform as it did during model simulation. Comparing fixed-point models with fixed-point code simplifies signal comparison activities.

Tool support: Comparison algorithms can be implemented in general-purpose programming or scripting languages such as MATLAB or be implemented in dedicated signal comparison tools.

MEval [20] provides a variety of predefined algorithms for result vector comparison including the difference matrix method. CGV provides an API for signal comparison as illustrated in Fig. 4. After defining the tolerance threshold, simulation results and SIL execution results are compared using these tolerances.

```

simData = cgvSim.getOutputData(1);
silData = cgvSil.getOutputData(1);

signalList = {'simData.ErrorsInjected.Data'};
toleranceList = {'absolute', 0.5};
cgv.CGV.createToleranceFile('localtol', signalList, toleranceList);
cgv.CGV.compare(simData, silData, 'Plot', 'mismatch', ...
'Tolerancefile', 'localtol');

```

Fig. 4: Signal comparison using CGV (code example)

To assess the mismatches, the user can plot and visually inspect the mismatching signals. Fig. 5 shows a signal comparison plot provided by CGV. The upper part of the figure window shows the result vectors $O_{M_PCG}(t)$ and $O_{CODE}(t)$, the lower part their absolute difference as well as the threshold used. You can also modify the script to output the names and more details about the mismatched signals.

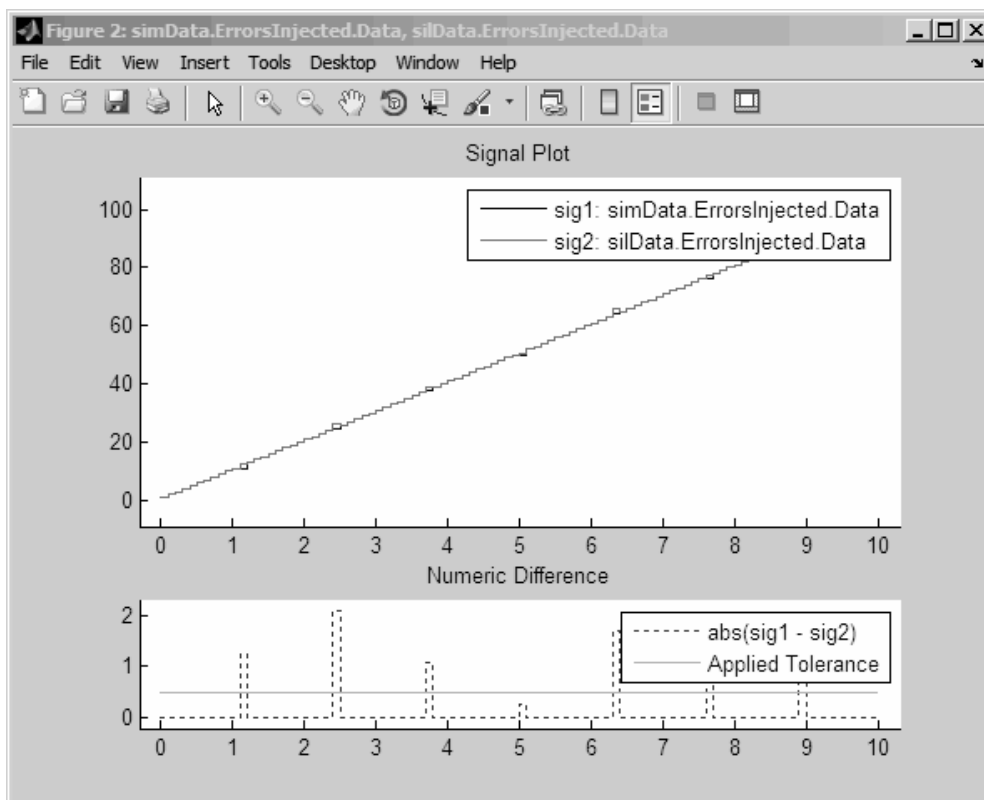


Fig. 5: Graphical signal comparison using CGV

3 Integration into IEC 61508 and ISO 26262 Workflows

As part of developing embedded, high-integrity automotive software, verification and validation methods and tools need to be reconciled with safety standards such as IEC 61508 and ISO 26262.

The proposed translation validation approach using CGV can be used to instantiate an overarching *workflow for the verification and validation of models and generated code* that satisfies the requirements of the ISO 26262 or IEC 61508 standards. This overarching workflow consists of two major phases:

1. *Design verification*: This phase combines suitable verification and validation techniques at the model level to demonstrate that the model is well-formed and meets all requirements.
2. *Code verification*: This phase utilizes equivalence testing (as described above) and other techniques to demonstrate equivalence between the model and the generated code compiled into an executable.

The main constructive development activities as well as the corresponding verification and validation activities of the workflow for the verification and validation of models and generated code are summarized in Fig. 6; [21, 22] provide more detailed discussions. The workflow was used as a cornerstone in the certification of Real-Time Workshop Embedded Coder by TÜV SÜD and is available as part of the IEC Certification Kit [23] product.

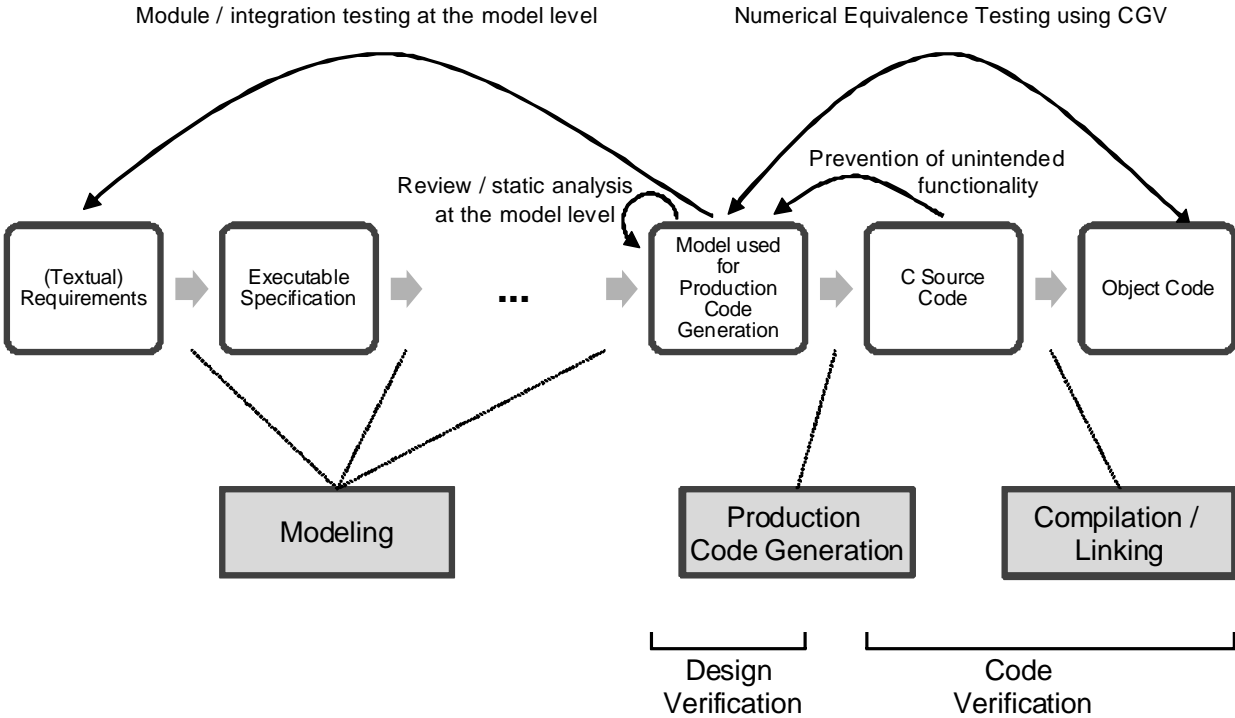


Fig. 6: Workflow for the verification and validation of models and generated code

4 Summary

The increasing usage of Model-Based Design with production code generation calls for new methods and tools for verifying and validating generated code in an efficient and effective manner. Model-Based Design allows for new verification and validation approaches utilizing the existence of executable graphical models as well as multiple executable artifacts which could be checked for equivalence.

To address this need, MathWorks developed Code Generation Verification (CGV) an automated, testing-based approach to assess the numerical equivalence between Simulink models and the generated code. This paper illustrated a CGV-based translation validation approach for embedded automotive software generated from Simulink models.

Users can leverage the APIs provided by CGV within their existing testing tools. This allows a seamless adoption of these capabilities.

References

- [1] Pnueli, A.; Siegel, M.; Singerman, E.: Translation Validation. Proc. 4. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98). Lisbon, Portugal, 1998; pp. 151-166
- [2] Edwards, P.D.: The Use of Automatic Code Generation Tools in the Development of Safety-Related Embedded Systems. Proc. Vehicle Electronic Systems, ERA Report 99-0484, 1999.
- [3] Toeppe, S.; Ranville, S.; Bostic, D.; Wang Y.: Practical Validation of Model Based Code Generation for Automotive Applications. 18. AIAA/IEEE/SAE Digital Avionics System Conference, 1999.
- [4] Burnard, A.: Verifying and Validating Automatically Generated Code, Int. Automotive Conference (IAC '04) Stuttgart, Germany, 2004; pp. 71-78.
- [5] The MathWorks, Inc. Simulink® – www.mathworks.com/products/simulink
- [6] The MathWorks, Inc. Stateflow® – www.mathworks.com/products/stateflow
- [7] The MathWorks, Inc. Embedded MATLAB™ - www.mathworks.com/products/featured/embeddedmatlab
- [8] The MathWorks, Inc. Real-Time Workshop® Embedded Coder™ - www.mathworks.com/products/rtwembedded
- [9] Helmerich, A., Koch, N., Mandel, L. et al.: Study of worldwide trends and R&D programs in embedded systems in view of maximising the impact of a technology platform in the area. Report for the European Commission, Brussels, Belgium, 2005
- [10] IEC 61508. International Standard Functional safety of electrical/ electronic/ programmable electronic safety-related systems. First edition, 1998-2000.
- [11] ISO/DIS 26262-8:2009. Draft International Standard Road vehicles — Functional safety. 2009
- [12] Conrad, M.: Modell-basierter Test eingebetteter Software im Automobil: Auswahl und Beschreibung von Testszenerarien. PhD Thesis, Deutscher Universitätsverlag, Wiesbaden, Germany, 2004.

- [13] Stürmer, I.; Conrad, M.: Test Suite Design for Code Generation Tools. 18. IEEE Int. Conf. on Automated Software Engineering (ASE '03), Montreal, Canada, 2003
- [14] Stürmer, I.; Conrad, M.; Dörr, H.; Pepper, P.: Systematic Testing of Model-Based Code Generators. IEEE Transactions on Software Engineering, Sep 2007, pp. 622-634
- [15] Aldrich, W.: Coverage Analysis for Model-Based Design Tools. TCS 2001.
- [16] The MathWorks, Inc. Simulink® Verification and Validation™ – www.mathworks.com/products/simverification
- [17] The MathWorks, Inc. Simulink® Design Verifier™ – www.mathworks.com/products/slidesignverifier
- [18] Conrad, M.; Sadeghipour, S.; Wiesbrock, H.-W.: Automatic Evaluation of ECU Software Tests. SAE 2005 Transactions, Journal of Passenger Cars - Mechanical Systems, SAE International, Mar. 2006
- [19] The MathWorks, Inc. Simulink® Fixed Point™ – www.mathworks.com/products/simfixed
- [20] IT Power Consultants. MEval – www.itpower.de/28-1-MEval.html
- [21] Conrad, M.: Testing-based translation validation of generated code in the context of IEC 61508. Form. Methods Syst. Des. 35, 3 (Dec. 2009), 389-401
- [22] Conrad, M.; Sandmann, G.: A verification and Validation Approach for IEC 61508 Applications. SAE World Congress 2009, Detroit, Michigan, USA, 2009
- [23] The MathWorks, Inc. IEC Certification Kit – www.mathworks.com/products/iec-61508

Authors:

Dr.-Ing. Mirko Conrad,	The MathWorks, Inc., Natick, MA, USA
Tom Erkinen,	The MathWorks, Inc., Novi, MI, USA
Dr.-Ing. Thomas Maier-Komor,	The MathWorks GmbH, Munich, Germany
Guido Sandmann,	The MathWorks GmbH, Munich, Germany
Marty Pomeroy,	The MathWorks, Inc., Natick, MA, USA