Applying Model-Based Techniques for Aerospace Projects in Accordance with DO-178C, DO-331, and DO-333

Ulrich Eisemann

Applying Model-Based Techniques for Aerospace Projects in Accordance with DO-178C, DO-331, and DO-333

U. Eisemann, dSPACE GmbH, Paderborn, Germany

1. INTRODUCTION

The main difference between the new standard for software development in civil aviation, DO-178C (see [1]), and its predecessor, DO-178B, is that the new one has standard supplements that provide a greater scope for using new software development methods. The most important standard supplements are DO-331 (see [2]) on model-based development and model-based verification and DO-333 (see [3]) on the use of formal methods, such as model checking and abstract interpretation. These key software design techniques offer enormous potential for making software development in the aerospace sector highly efficient. At the same time, they not only maintain the high quality and observe the safety requirements for software, but actually improve them. These methods are seamlessly integrated in the development scheme of DO-178C, which is shown in a simplified form in Figure 1.

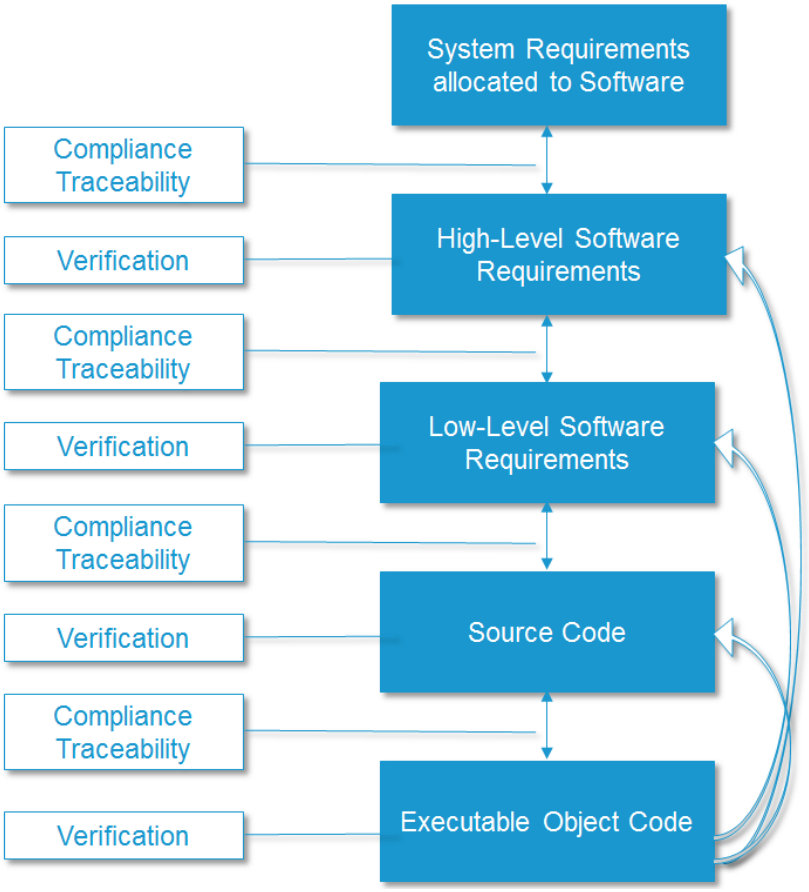

Figure 1: Important design and verification activities according to DO-178C (architecture design and verification have been omitted).

2. OVERVIEW OF THE MODEL-BASED TOOL CHAIN FOR DO-178C, DO-331, AND DO-333

This article describes how to use a model-based tool chain including Simulink®/Stateflow® ([4]), dSPACE TargetLink® ([5]), and tools by BTC Embedded Systems ([6]) to develop software right up to DO-178C Level A by using the standard supplements DO-331 and DO-333, see [7].

The main components of the tool chain are:

- Simulink/TargetLink for the graphical, model-based development environment
- BTC EmbeddedSpecifier (optional) for formalizing requirements
- TargetLink for automatic production code generation
- BTC EmbeddedTester and BTC EmbeddedValidator for meeting different verification and testing objectives

The above tools cover the following essential steps in the software development process in accordance with DO-178C/DO-331, see Figure 2:

- Specifying high-level requirements in the form of Simulink/TargetLink models, which then constitute specification models according to DO-331
- High-level requirements in the form of specification models can also be developed by using EmbeddedSpecifier, which is particularly attractive for converting existing textual requirements to formal requirements
- Representing low-level requirements in the form of Simulink/TargetLink models, which thereby constitute design models according to DO-331
- Automatically generating source code with TargetLink in order to convert the Simulink/TargetLink design models directly to high-quality ANSI C code
- Achieving various verification objectives of DO-331 via model-based testing with BTC EmbeddedTester and BTC EmbeddedValidator to efficiently create requirements-based test cases, automatically execute them in the Simulink/TargetLink environment, and determine coverage metrics such as MC/DC at the code level and model coverage to determine requirements coverage.
- Using BTC EmbeddedValidator for model checking as a formal method in the sense of DO-333 to demonstrate that the models comply with formalized requirements developed using BTC EmbeddedSpecifier.

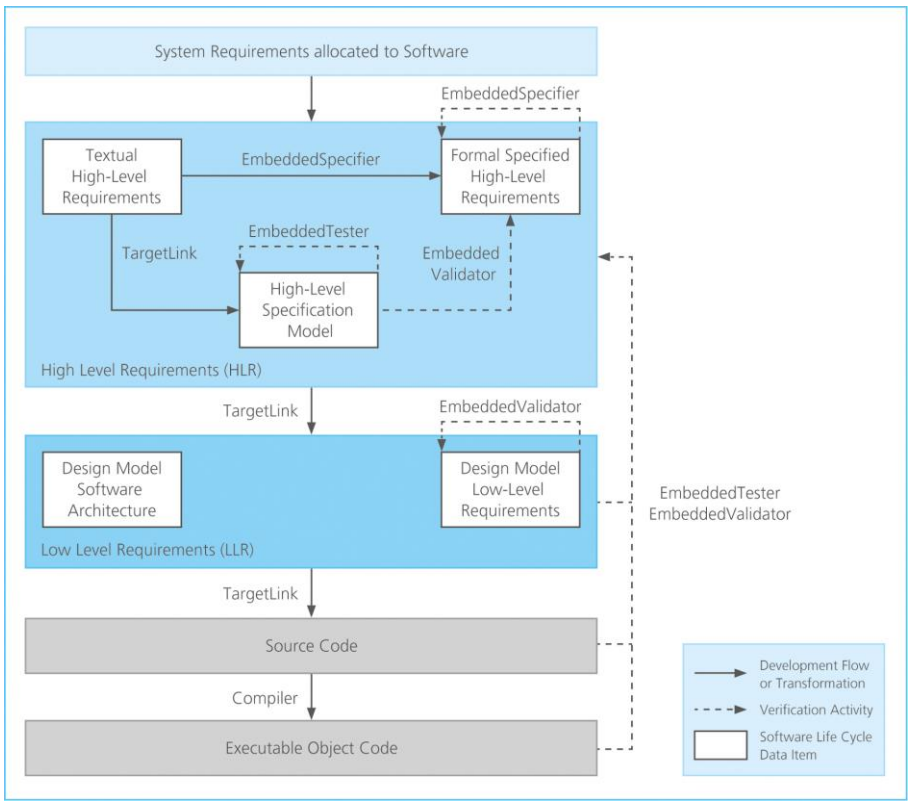

Figure 2: Model-based design tool chain for DO-178C/DO-331-compliant development.

3. MODELS AS A DOOR-OPENER FOR EFFICIENT SOFTWARE DEVELOPMENT

A key milestone for efficient and high-quality software development is representing requirements and specifications by models according to DO-331. A transition from purely textual to formalized requirements in the form of models opens up a wealth of possibilities for automated analysis, source code generation and verification, as will be demonstrated in this article.

Software requirements in DO-178C exist in two different forms, either as

• **High-level requirements (HLR)**

Simply put, they describe what the software is supposed to do but not how it should do it ("black box" view of the software). HLRs are derived from requirements for the entire system or subsystem, which are set up in the system process, e.g., as described in ARP 4754.

• **Low-level requirements (LLR)**

They describe the inner workings of the software, i.e., how the software is supposed to implement the HLRs ("white box" view of the software) and they must comply with the HLRs. It must be possible to translate LLRs directly into source code, which is later translated into the executable object code.

Models in the sense of DO-331 can now be used to represent requirements on these two levels (models for software architecture, e.g., in the form of UML models, are not discussed here but are widely used in practice and also covered by DO-331). In the case of HLRs, DO-331 speaks of specification models, whereas models for representing LLRs are referred to as design models.

3.1. Specification Models for High-Level Requirements

For some functional HLRs, it is best to use dedicated tools, such as BTC EmbeddedSpecifier, that support specific formalized patterns for expressing requirements. This is particularly the case when requirements are not written as a set of mathematical equations or “pseudo code” but come in the form of rather simple signal dependencies and conditions. BTC EmbeddedSpecifier is specifically geared towards requirements of such a form. Therefore, a very generic requirements design pattern in the form of a state machine is used which includes trigger conditions for the requirement as well as actions (see Figure 3). Timing dependencies can be expressed in the requirements design pattern as well. The tool is particularly helpful for the step-by-step transformation of informal textual requirements to formalized patterns in an intuitive process. For this purpose, informal signal names are replaced by actual interfaces in Simulink/TargetLink models and the meaning of the text is translated into the requirements pattern. The tool thereby simplifies the transition from informal to formal requirements with a clearly defined syntax and semantics, which lets users later use verification methods like model checking and automatic test case generation. Moreover, due to the formal nature of the requirements pattern, there is a precise definition for “requirements coverage” to identify whether developed test cases have covered all requirements.

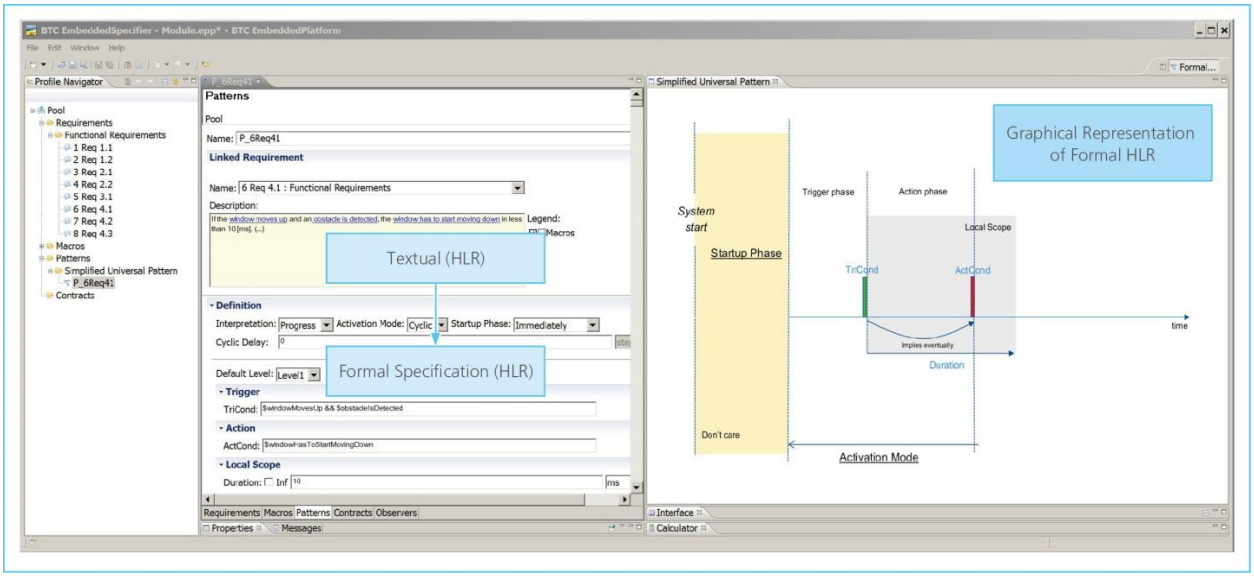

Figure 3: Converting informal textual requirements into formalized requirements.

HLRs can also be expressed by block diagrams and state diagrams in the form of Simulink/TargetLink models (see Figure 4). This is particularly useful if such models are already available from the system process. TargetLink automatically makes sure that only a “safe subset” of Simulink and Stateflow is used during the design process to avoid problems with certain modeling constructs. Moreover, compliance with company-specific standards for specification models can be verified by using automatic style checkers.

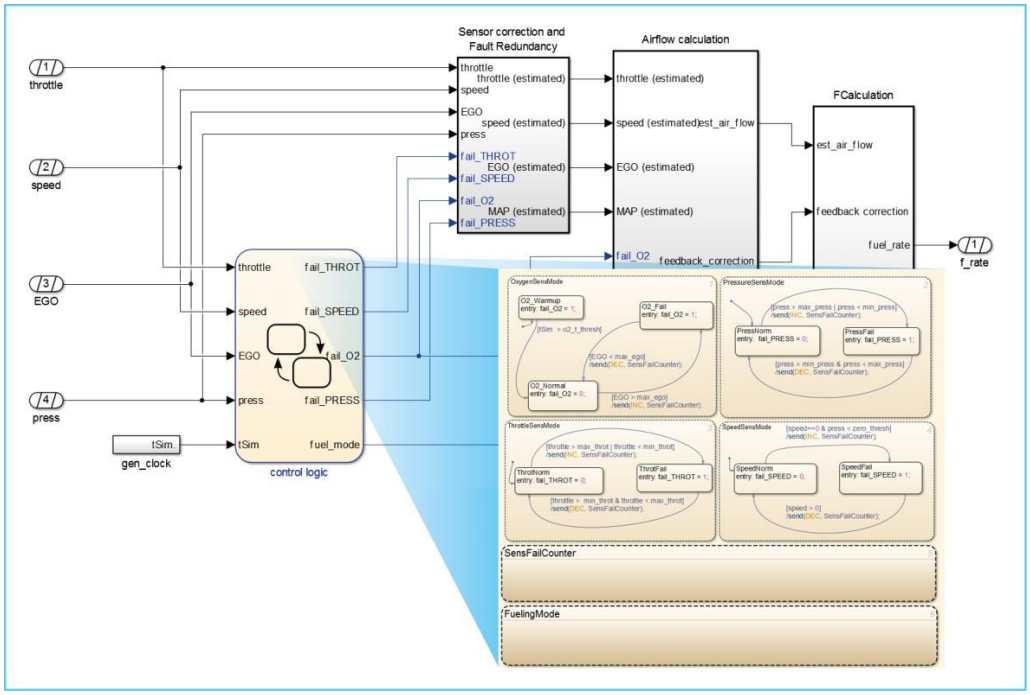

Figure 4: Simulink/TargetLink specification models for representing high-level software requirements.

3.2. Design Models for Low-Level Requirements

By nature, representing low-level requirements (LLRs) in the form of Simulink/TargetLink models is particularly popular, as source code can be generated from them automatically in subsequent steps (Figure 5). Such design models not only describe the actual functionality but also the necessary details of the software, such as internal data structures, distribution across different functions, control flow information, and, in some cases, fixed-point representations.

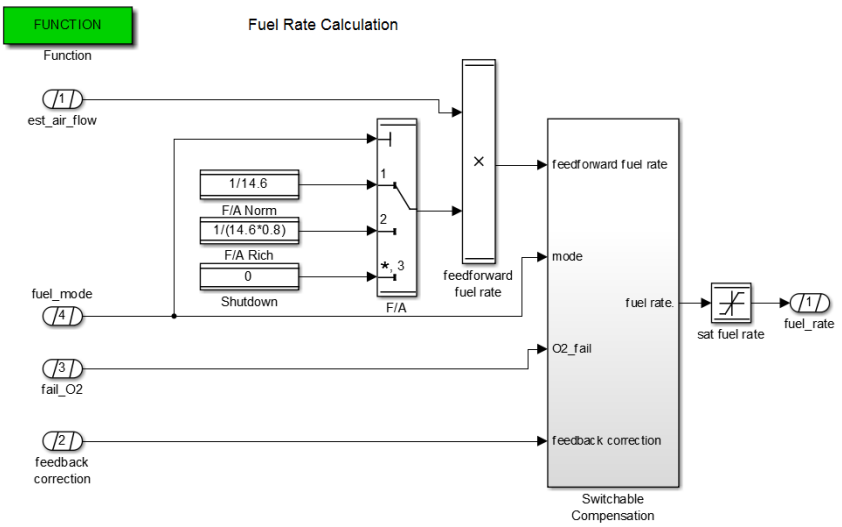

Figure 5: Design models in Simulink/TargetLink represent low-level requirements.

4. GENERATING SOURCE CODE FROM DESIGN MODELS

Design models representing low-level requirements offer a direct route to creating the source code – by using automatic code generation instead of manual coding. Automatic code generators, such as TargetLink, that transform Simulink/TargetLink models directly into ANSI C code are far superior in terms of quality and reliability compared to human programmers. The source code they produce

- Is generated deterministically
- Is very readable and suitable for review. This is ensured by extensive source code commenting, easily understandable symbol names and the use of a proper subset of the C programming language.
- Can be traced back directly to the individual parts of the design model from which the code was generated.
- Contains references to the requirements of the design model. This increases requirement traceability, which represents an important part of any software development process.
- Is highly configurable in terms of how to generate code, e.g., to meet company-specific coding guidelines and to integrate easily with the interfaces of the software architecture, library functions and legacy code.
- Is approximately as efficient as handwritten code. Individual optimizations on a detailed level let users specify the extent to which the code will be optimized, ranging from completely unoptimized to fully optimized code.

In addition to generating the actual source code, the code generator can generates other items, thereby assuring consistency between all the artifacts. Typical examples include:

- Additional documentation files adjusted specifically to special purposes
- Files with traceability information in order to support reviews and analysis of the source code
- Information on the requirements implemented by the source code to build a requirements traceability matrix

Using TargetLink for automatic production code generation from Simulink/Stateflow models has been widely used for many years, especially in the automotive sector, but also in DO-178B Level A projects (see [8]).

5. INNOVATIVE TECHNIQUES FOR VERIFICATION

The great advantages of using models to specify requirements (HLRs and LLRs) are evident not only in automatic production code generation but also in other areas, such as verification. Particularly important activities are:

1. Verifying that models for HLRs and LLRs accurately implement the requirements from which they were developed
2. Verifying that models and source code comply with specific modeling guidelines and coding guidelines
3. Verifying that the source code exactly implements the requirements contained in the design model
4. Verifying that the executable object code implements HLRs and LLRs

The model-based approach combined with the formalization of the requirements in this case provides very powerful mechanisms to facilitate the verification steps, as will be shown in the following subsections.

5.1. Verifying Requirement-Model Compliance

A combination of model simulation, coverage analysis, and test case generation is especially well-suited for verifying the compliance between the requirements and the models that implement them. DO-178B and DO-178C state that test cases have to be created solely on the basis of requirements whose definitions directly include the required result. If a requirement is itself expressed as a model, e.g., a Simulink/TargetLink model, techniques for automatic test case generation, such as those provided by BTC EmbeddedTester, can be used to automatically generate test cases from the specification model (see [10]). If high-level requirements were expressed formally in BTC EmbeddedSpecifier, then automatic test case generation can also be used for those. Naturally, test cases can also be developed manually or semi-automatically based on the requirements. To check the compliance of a model against the requirements from which it was developed,

the previously generated or developed test cases/stimulus values are executed with the model by means of model simulation. The Simulink/TargetLink/BTC environment makes it easy to run the model simulations and check the simulation results against the previously developed test cases. Moreover, model coverage analysis is used to investigate whether the various model elements are all covered completely (Figure 6) in order to identify undesired functionality in the model.

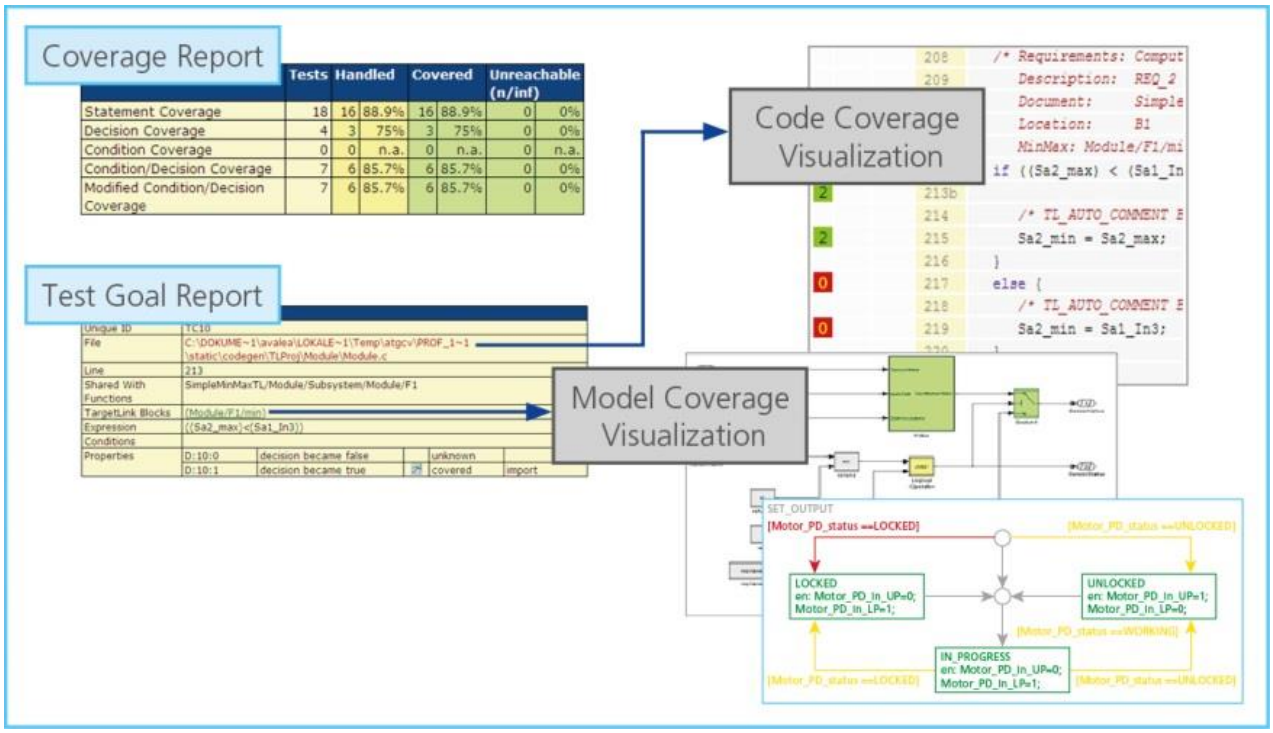

Figure 6: Model coverage and code coverage reports.

Although the simulation capabilities of models help verify the compliance between requirements and models, the traditional execution of even a high number of test cases is always incomplete in a certain sense. If a proof is desired or all requirements are to be verified simultaneously during all simulation runs, different verification methods can be used as long as the requirements were specified formally in BTC EmbeddedSpecifier. By using the model checking functionality of BTC EmbeddedValidator, users can prove or disprove compliance between the formally specified requirements and the design model. In particular, the model checking engine of EmbeddedValidator is fully capable of handling not just boolean and integer data types but also floating-point design models. Model checking is one of the techniques considered in the formal methods supplement DO-333 of DO-178C. Providing a proof of correctness is of course very appealing, but the usual limitations of model checking need to be considered, like a potential explosion of the state space with increasing complexity of the design model. If a model checking approach is not feasible, formal requirements specifications can nonetheless be very helpful to generate requirement observers, which verify all requirements simultaneously during all simulation runs.

5.2. Compliance of Models and Code with Modeling and Coding Standards

In order to check whether models comply with their respective modeling standards, style checkers such as MXAM from Model Engineering Solutions (see [11]) or the Simulink Model Advisor are typically used. These tools support automatic guideline checking in order to simplify review activities on the different models. The compliance of the generated source code with coding guidelines is usually verified by using static source code analysis tools that are available on the market.

5.3. Compliance of Source Code with Design Models

In order to check that the automatically generated source correctly implements the design models from which code was generated, developers can use reviews and analyses, which is also facilitated by TargetLink. In

order to simplify code reviews, traceability information as well as hyperlinks between design model elements and the respective generated source code lines are provided. In addition, further analyses can be fully or partially automated.

5.4. Compliance of Executable Object Code with High-Level and Low-Level Requirements

A typical way to verify that the executable object code implements HLRs and LLRs according to Figure 1 is to execute it on the target platform. TargetLink provides powerful mechanisms for this in the form of processor-in-the-loop simulation, in which the automatically generated code is translated directly by the target compiler and executed on an evaluation board with the target processor (Figure 7). Testing is performed in the Simulink/TargetLink environment and can be directly compared with the results of a model simulation (model-in-the-loop simulation). This test design lets users reuse all test cases that were developed manually or generated automatically. BTC EmbeddedTester provides a powerful environment for performing the required tests, including the automatic comparison of model simulation and processor-in-the-loop simulation. The associated test reports are created completely automatically. BTC EmbeddedTester also supports code coverage analysis (MC/DC coverage, condition coverage, etc.), see Figure 6.

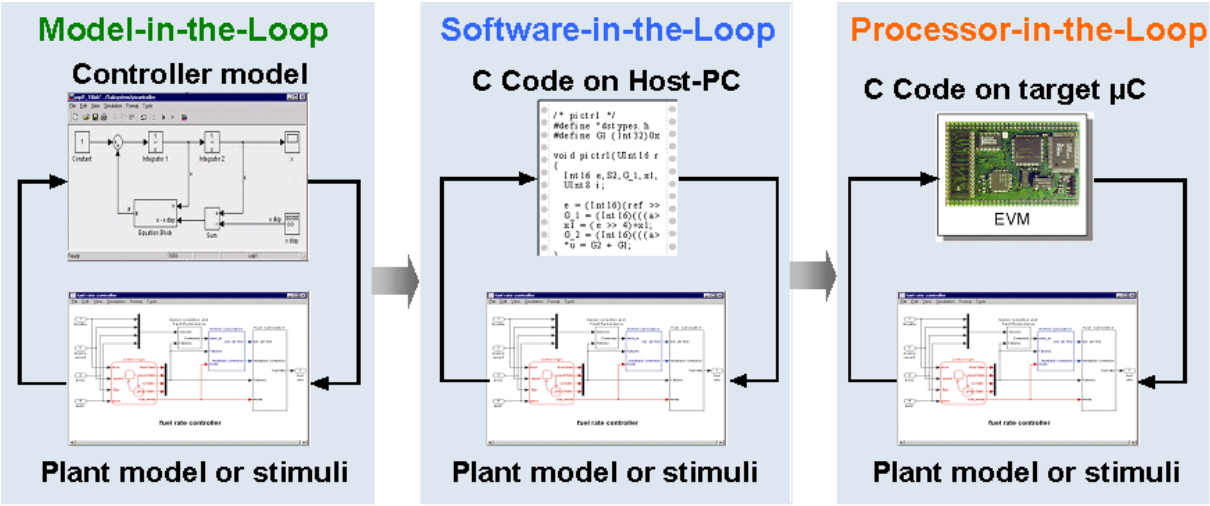

Figure 7: Model simulation and tests in different simulation modes.

6. TOOL QUALIFICATION ASPECTS, LIMITATIONS AND EFFECTIVENESS OF THE TOOL CHAIN

The proposed tool chain is based on the Simulink/Stateflow modeling environment and TargetLink for automatic source code generation. But a code generator as powerful as TargetLink is very hard to qualify as a criteria 1 tool according to DO-330 (see [9]), the Software Tool Qualification Considerations supplement to DO-178C. Therefore, the idea is to use an unqualified code generator and instead perform verification steps as required by DO-178C/DO-331. However, the verification steps greatly benefit especially from the use of verification tools provided by BTC, be it for the source code or the object code. This means that tools like EmbeddedTester (criteria 3 tool) and the model checking engine in EmbeddedValidator (criteria 2 tool) would have to be qualified to claim certification credits for the application of the tools.

Regarding the application of the tool chain, the following primary aspects and limitations should be considered:

- Since the whole tool chain makes extensive use of Simulink/Stateflow, its application is primarily attractive where this environment seems most appropriate judged by the nature of the algorithms and the

software that is to be developed. Where algorithms are best described by block diagrams and state diagrams, the tool chain can boost efficiency.

- Another limitation lies of course in the optional application of model checking to verify the compliance between models and the requirements from which the models were developed. The larger the model and the more complicated the requirement, the less likely it is that a model checking engine will be able to provide a proof in time.

It should be noted that production code generators do not constitute a weakness in the tool chain. On the contrary, a modern production code generator like TargetLink produces code much more reliably and with much higher quality than a human programmer could.
In summary, the tools provide powerful mechanisms not only for requirements definition and design, but especially for the various verification steps, making it possible to meet the objectives of DO-178C quite easily and with less effort than by using conventional techniques.

References

[1] RTCA DO-178C “Software Considerations in Airborne Systems and Equipment Certification”. RTCA, Inc., 2011
[2] RTCA DO-331 “Model-Based Development and Verification Supplement to DO-178C and DO-278A”. RTCA, Inc., 2011
[3] RTCA DO-333 “Formal Methods Supplement to DO-178C and DO-278A”. RTCA, Inc., 2011
[4] MATLAB/Simulink/Stateflow, www.mathworks.com
[5] dSPACE TargetLink, www.dspace.com/go/targetlink
[6] BTC EmbeddedSpecifier, BTC EmbeddedTester, BTC EmbeddedValidator, www.btc-es.de
[7] “TargetLink – Model-Based Development and Verification of Airborne Software“, dSPACE GmbH Paderborn, July 2014
[8] Aloui, Andreas, “C Code Reaches New Heights at Nord-Micro”, dSPACE Newsletter, 1/2002
[9] RTCA DO-330 “Software Tool Qualification Considerations” Supplement to DO-178C and DO-278A”. RTCA, Inc., 2011
[10] H.J. Holberg, U. Brockmeyer, “Significant Quality and Performance Gains through Fully Automated Back-to-Back Testing”, EmbeddedWorld 2010
[11] “Model Examiner”, www.model-engineers.com