Aspects of a Consistent Modeling Environment for DO-331 Design Model Development of Flight Control Algorithms

Markus Hochstrasser, Simon P. Schatz, Kajetan Nürnberger, Markus Hornauer, Stephan Myschik and Florian Holzapfel

1 Introduction

Within various projects, the *Institute of Flight System Dynamics* at *TU München* developed a modular flight control software for usage in applications ranging from manned autopilot functions to unmanned operations from ground. The controllers were developed using a model-based approach with *MATLAB*, *Simulink*, and *Stateflow*. To guarantee high-quality models and code, and to pave the way for future certification, a model-based development process based on industry standards proposed for airborne software was developed.

In the chosen development approach according to DO-178C [19] and DO-331 [20], *Simulink* and *Stateflow* models step in the place of “Software Low-Level Requirements” and “Software Architecture”. The “Design Models”, as they are called by DO-331 in this context, are directly used to generate ANSI C “Source Code”. The use of *MATLAB*, *Simulink*, and *Stateflow* in such a context requires a special setup,

M. Hochstrasser (✉) · S.P. Schatz · K. Nürnberger · M. Hornauer · F. Holzapfel
Institute of Flight System Dynamics, Technische Universität München, Munich, Germany
e-mail: markus.hochstrasser@tum.de

S.P. Schatz
e-mail: simon.p.schatz@tum.de

K. Nürnberger
e-mail: kajetan.nuernberger@tum.de

M. Hornauer
e-mail: markus.hornauer@tum.de

F. Holzapfel
e-mail: florian.holzapfel@tum.de

S. Myschik
Chair of Flight Mechanics and Flight Control, Universität der Bundeswehr München, Neubiberg, Germany
e-mail: stephan.myschik@unibw.de

B. Dołęga et al. (eds.), *Advances in Aerospace Guidance, Navigation and Control*, https://doi.org/10.1007/978-3-319-65283-2_4

preparation, and limitation of the tool capabilities. The required documentation and configuration is in general created during the Software Planning Process and in the following denoted as “Modeling Environment”. Estrada [6] emphasizes the importance of an environment, but does not discuss the details of its content.

The Modeling Environment describes a package of settings, libraries, and templates that are made available to developers in order to support them in implementing models and generate code that is safe and compliant to the defined process. The difficulty in setting up such an environment originates from the close connection between Design Model and Source Code and the standards both have to fulfill. Since code generation is normally done by a single function call, the Design Model not only describes the design itself, but also the appearance of the generated code. A well-prepared Modeling Environment can significantly improve the results of static code analysis and standard compliance checking in subsequent steps and reduce the remaining effort as shown in [9].

Another challenge is that model developers usually do not have the experience of a C programmer, but are responsible for the generated safety-critical code and its compliance to standards as well. Thus, the fulfillment is mainly determined by the quality and consistency of the conventions and settings from the provided Modeling Environment.

The Modeling Environment normally depends on multiple factors and may vary from company to company. However, the presented basic set is required in most of the projects. Since the content may be influenced by the chosen development process, the application to be developed and the hardware on which the application shall be executed, Sect. 2 introduces the applied model-based development process and Sect. 3 gives a short overview about the flight control computer hardware and the controller design. Based on this, Sect. 4 contains a detailed discussion of the suggested Modeling Environment.

2 Model-Based Software Development Process

The objective was to develop a controller application in alignment with guidance material for a CS-23 aircraft. In Book 2 of CS-23 [7], the relevant paragraph CS 23.1309 “Equipment, systems and installations” does not provide acceptable means of compliance for digital flight control systems, but EASA Certification Review Items refer to FAA Advisory Circular AC 23.1309-1E [8], which itself refers DO-254 [18] and DO-178B/C [19] as guidance material for software and hardware development.

With DO-178C, RTCA published additional supplements addressing modern techniques for software development like model-based approaches (DO-331 [20]) or formal methods (DO-333 [21]). Especially model-based development became popular in the recent years, since it provides means to implement complex software, simulate the models in an early development stage, and automatically generate code.

Tool vendors for model-based development software try to provide a consistent tool chain, allowing the fulfillment of DO-331 objectives in an effective and mostly automated way. Most popular are the workflows provided by *MathWorks* [32] and *Esterel* [5].

For the given project, the *Institute of Flight System Dynamics* decided to setup a development process based on the *MathWorks DO Qualification Workflow* [32] for applications up to DAL B. The workflow proposes to use *Simulink* and *Stateflow* to implement the Design Model, replacing Software Low-level Requirements as well as Software Architecture from the conventional DO-178C process. The approach coincides with DO-331 MB.1.6.3 Example 1 and is also presented in [4, 17].

C Source Code is directly generated from the Design Model using *Embedded Coder* (EC). Since the code generator is not shipped with a Tool Qualification Kit, *MathWorks* provides *Simulink Code Inspector* to automate code review and verify compliance of the Source Code with the Design Model as well as traceability. Using SLCI restricts features of *Simulink* and *Stateflow* to a robust and safe subset.

The workflow and tool chain at the institute subsequent to the generation of Source Code is described in [10]. The interplay with a system design process for control algorithm development is outlined in [11].

Using a Design Model and directly generating code out of it brings the DO-178C Design Process closer to the Coding Process. The work to be done in the Coding Process reduces to a single function call of the coder. The fulfillment of Source Code objectives from Table MB.A-5 now can only be influenced by the Design Model. The *Simulink* and *Stateflow* models thus have to address rules from a Software Design Standard, a Model Standard but also a Code Standard.

3 Hardware and Software Context

3.1 Flight Control Computer Hardware

The addressed controller runs on a single flight control computer (FCC). This is possible due to the system architecture of the research aircraft and the safety concept of the experimental flight control system [1].

The flight control computer has been developed together with industry partners regarding certification aspects.[1] The Power PC 32-bit architecture has a clock speed of 533 MHz with a double-precision floating point unit and works in Big-Endian mode. It also has a high-speed interface to two Cortex M-I/O-processors, providing various UART, ARINC-825, and ARINC-429 interfaces along with multiple discrete IOs.

[1]http://www.fsd.mw.tum.de/infrastructure/gnc-subsystems/ [Cited on 6 January 2017].

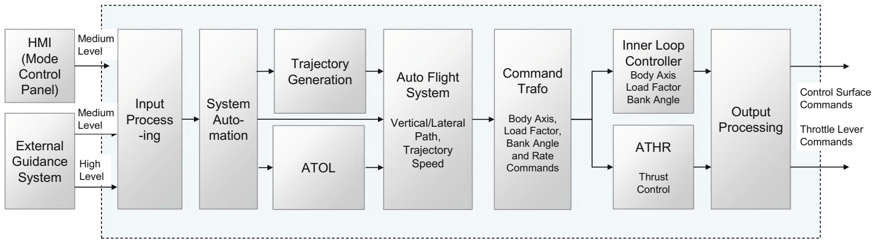

Fig. 1 Structure of the modular integrated flight control system

3.2 Controller Design Considerations

3.2.1 Controller Architecture

Figure 1 shows a simplified version of the cascaded controller and the identified software modules. Input and output processing modules contain reorganization of the incoming and outgoing data. Core of the algorithm is an aircraft dependent baseline innerloop controller, whose input commands are the load factors. The auto flight system contains a trajectory/path controller and a controller for well-known autopilot functionalities like attitude, altitude, heading, or speed hold. Further information about the auto flight module and the integrated trajectory/path controller is provided by [14, 15, 22]. For high-level commands, e.g., waypoints, the controllable path is calculated by the trajectory generation module as described in [23, 25, 26]. Additionally, there exists a module for automatic landing and take off (ATOL). All mode switching as well as the startup behavior is implemented in the system automation module as outlined in [16].

3.2.2 Software Modules and Interfaces

The controller is concurrently developed by various developers that are working on and are responsible for clearly defined software modules. A software module consists of a set of Simulink Models and dependent data passing the software lifecycle together. A software module has an own revision index, a defined interface and separately allocated requirements. The schematic draft of the system in Fig. 1 also outlines the most important software modules.

3.2.3 Floating-Point Arithmetic

Since the FCC has a double-precision floating point unit, this arithmetic is preferred. Although real necessity for double-precision is only given when dealing with WGS-84 waypoint positions, it was decided to honor simplicity and perform all calculations with double-precision.

Understanding generated shared utility functions (see Sect. 4.2.4) and treatment of special floating point quantities (NaN, Inf,...) requires a deeper look into floating point arithmetic. The FCC partly supports floating point arithmetic standardized by IEEE-754:2008 [12]. Here, the *binary64* format with base two, an exponent length of ten bytes and a mantissa length of 52 bytes (plus one bit for the sign) is used. This complies with the internal floating point representation and arithmetic of *MATLAB*.[2]

The treatment of special floating point quantities is shortly addressed in the following section. The internal floating point exception handling as specified in IEEE-754 is not considered.

3.2.4 Interface to C Framework

The model-based controller is embedded in a conventionally developed C framework. The top-level model of the controller application, as simplified in Fig. 2, provides one in- and outport for every incoming and every outgoing physical interface of the FCC. The port data types are designed as Simulink Bus objects, which are translated to C structures during code generation. The coder settings therefor are described in Sect. 4.2.1.

The input and output buses have sub-buses for every readable/writable message, which itself contain elements for every message payload parameter. Additionally, every message bus has an update indication, notifying that a new message has been received, or a message as to be sent, respectively.

The execution of the whole software is as follows: Every iteration the surrounding C-Framework decodes incoming messages and copies the received data into the exposed C structures of the generated code. After that, it calls the step function of the application. As soon as the step function has finished, the framework copies the data from the outgoing C structures into the messages to be sent.

It is also visible in Fig. 2 that the incoming data is rearranged into functional structures in system "datain" at first (e.g., into `sensors_in`). This system additionally performs basic input monitoring like saturation to a specific range. From this point on, no special floating point quantity shall be in the software any more and no subsequent software module shall introduce such.

4 Modeling Environment for Safety-Critical Software with Simulink and Stateflow

This chapter summarizes the most important aspects, which a consistent Modeling Environment for *MATLAB Release 2016b* must fulfill to create safety-critical software. For other releases, variations are possible. The Modeling Environment is

[2]http://de.mathworks.com/help/matlab/matlab_prog/floating-point-numbers.html [Cited on 09/05/2016].

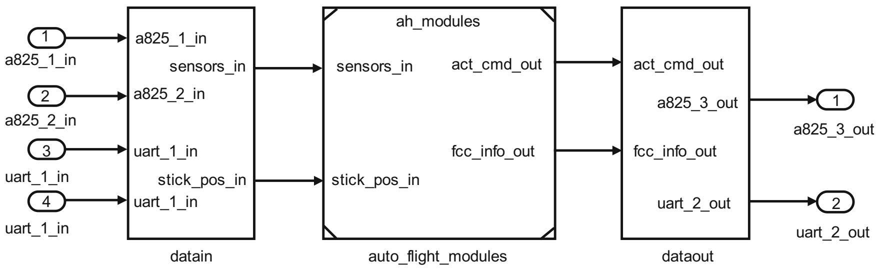

Fig. 2 Schematic draft of the top-level controller model

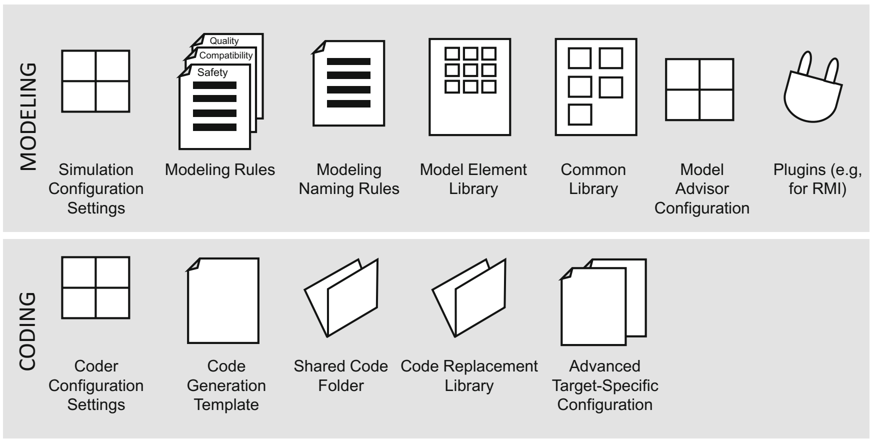

Fig. 3 Modeling Environment provided to developers at the *Institute of Flight System Dynamics*. The whole package is under version control and its content must be read or installed by the developers in Simulink prior to implementation

defined during the software planning process and mainly documented in the Software Model Standard (DO-331 MB.11.23 “Model Standards”) with respect to

- existing modeling rules,
- standards for generated code,
- tools used in the process,
- strategies for integration, and
- planned target hardware.

The Modeling Environment used at the institute is depicted in Fig. 3. For this paper, the Modeling Environment is divided into a modeling and coding part. In reality, the border is indistinct. For example, *Simulink* uses a single configuration file/object, mixing options for simulation and for code generation with EC. This is similar for modeling rules that influence the model, but with respect to the generated code.

Section 4.1 introduces the part of the Modeling Environment that primarily impacts the Design Model in its function as Software Low-Level Requirements and Software Architecture. Section 4.2 focuses on details relevant for code generation.

4.1 Environment for Design and Simulation

4.1.1 Modeling Rules

Basis for the finally applied set of rules in the presented application were the *MAAB Control Algorithm Modeling Guidelines* [27], the *MathWorks High-Integrity System Modeling Guidelines* [30], and the *MathWorks Code Generation Guidelines* [29].

This basic set was reduced by removing duplicates resulting from the independent historical origin of the rules. Additionally, restrictions imposed by subsequent tools, especially SLCI, allowed the exclusion of further guidelines (e.g., since no *Embedded MATLAB Charts* and *Functions* are supported, all rules on *MATLAB* code could be excluded). Instead and supplemental to this basic collection, custom rules have been defined. Table 1 lists a summary of the applicable modeling rules at the institute. In some cases, the custom rules replace or overwrite parts of the *MAAB* and *MathWorks* rules. Then, the respective, underlying rules have been marked as not applicable and were redefined.

Not included in Table 1 are rules concerning compatibility. Compatibility rules e.g., for SLCI are documented in the corresponding tool operational requirements. In the case of SLCI, most of the compatibility rules are also shipped with checks.

During the verification process, the rules may either be reviewed or automatically checked using *Simulink Model Advisor*. The collection of Model Advisor checks is provided to the developer within the Modeling Environment as "Model Advisor Configuration" to allow continuous checking before the subsequent verification process is triggered.

A closer look at the set of applicable rules reveals, that not all of them have the same criticality. Rules addressing model appearance are for example necessary to achieve readable models, but do not impact the generated code. In contrast, an integer overflow behavior setting is of much higher importance. Thus, the rules are categorized into three groups:

Table 1 Composition of modeling rules

Source	# Rules original	# Applicable[a]
MAAB (v. 3.0)	109	54
MW HI (R2016b)	101	72
MW CG (R2016b)	12	5
Custom	51	51

[a]For projects at the *Institute of Flight System Dynamics*

Safety-critical Rules These rules describe model settings, architecture, and patterns that significantly contribute to safe code. If these rules are not respected, it is possible that errors are introduced in the code, which are hard to find in subsequent verification processes. Additionally, rules are contained that play an important role in the process and would require a significant restructuring and redefinition. Related Model Advisor checks of this group require a tool qualification, and a failure or warning is not accepted and not justifiable.

Compatibility Rules These rules ensure compatibility with subsequent tools and the process. Incompatibility introduced by the model developer is reliably detected by the affected qualified tools, thus these rules have no direct relevance for safety. However, when incompatibility is discovered late, this may lead to significant rework. This category of rules requires no tool qualification for the checks, a failure is not accepted, but a warning may be formally justified.

Quality Rules Under this category, all non-critical rules are summarized. They do not have a direct impact on safety, or their impact is analyzed in subsequent steps with qualified tools. The rules address readability, maintainability, appearance and comfort settings. A single violation is not critical, but has impact on the quality of the model. Related Model Advisor checks shall not fail, but a warning may be acceptable without formal justification. Quality rules may be summarized to a quality index. Then, an acceptable quality range may be defined and checked. Quality metrics and their automated evaluation are a rising topic for models [24].

4.1.2 Modeling Naming Rules

Beside modeling rules, modeling naming rules have been established. To avoid overloading of the modeling rules, they were defined in a separate document. The naming rules give syntax and semantics for names of files and folders, workspace objects, as well as blocks and signals belonging to the Design Model. The work required for defining proper naming conventions should not be underestimated due to the direct impact of the model on code and the high number of different files and workspace objects to be covered.

The purpose of naming conventions is to support

Compliance to Modeling Rules As described above, the developed set of modeling rules bases on *MAAB*, *MathWorks High Integrity*, and *MathWorks Code Generation Guidelines*. At least the first two define naming rules, which have to be respected.

Integration of Software Modules Since the whole application is split into different software modules, naming conventions must ensure that integration of models and code is possible without conflicts.

For example, all software modules at the institute currently initialize their objects into the base workspace, since SLCI had an incompatibility with *Data Dictionaries* for a long time. Since the model workspace did not support Simulink.Bus objects or Simulink.Parameter and Simulink.Signal objects with explicitly set storage class, the shared base workspace was the only remaining solution. Naming conventions ensure that the software modules do not conflict and overwrite data when loaded together. This special issue was resolved by introducing a short identifier for every separately developed module prepended to all data initialized in the base workspace (e.g. `af_sensor_data_Bus` with prefix `af` for the autopilot module).

In the Source Code, the uniqueness of global identifiers and file names across software modules is directly influenced by naming conventions of the model (e.g. naming of an exported header file). Naming conflicts lead either to mangling in the code generation process, or errors later on during compilation.

Compliance to Coding Rules Coding rules like MISRA C:2012 [33] include guidance on naming, e.g. enforcing uniqueness of identifiers within a specified range of characters. Due to automatic code generation, the naming of workspace objects is the only way to influence MISRA:2012 compliance of the generated code beside tuning the coder settings.

A typical issue is that the coder adds prefixes to identifiers, which leads to longer identifiers in the generated code and may result in violation of MISRA C:2012 rule 5.1 requiring a distinct external identifier within the first 31 characters for C99 (due to this definition, postfixes are in general no issue for MISRA compliance). A bus `sensor_data_Bus` in a model called `flight_control` may introduce the external identifier `flight_ control_rtZsensor_data_Bus`. In general, EC can automatically limit identifier length. This however reduces readability and has limitations.[3] To avoid a violation, both restrictions on the model and bus name length are required. Helpful for figuring out limitations of the coder is also the *MathWorks Embedded Coder Compliance Report* [31].

In newer releases of EC, the treatment of pre- and postfixes has been significantly improved. In some cases, if the length of the generated identifier exceeds the maximum identifier length specified in the coder settings, pre- and postfixes are removed before the original name is shortened.

Readability of Models Workspace object names should have a clear syntax and already provide some information about their type and physical meaning. Best example is the label of a signal line, which may provide a hint of its boolean data type by the postfix `_flg`. However, more important for a controller application is that it states its physical meaning in a clear way. A major part of the modeling naming rules is thus defining the naming of coordinate systems, angles, transformation matrices (e.g., `M_O_B` for M_{OB} as transformation from body-fixed to the north-east-down system), units (e.g., `mDs2` for m/s^2) or the translation of mathematical characters (e.g., `mu` for μ or `x_ddot` for $\ddot{x}$).

Readability of Code As already mentioned, coding standards introduce naming restrictions to ensure compliance with a majority of compilers. Code generators like EC provide functionality to enforce this compliance. However, most of these automatic procedures reduce readability and their usage should be avoided if possible

[3]Limitations on controlling the identifier format are documented in the *Embedded Coder User's Guide* [28] pp. 36–33 "Identifier Format Control Parameters Limitations".

by designing the model correctly. For example, a concept to enforce uniqueness of identifiers is mangling, meaning that similar identifiers are shortened and an arbitrary but unique sequence of characters is appended. For example, when using *Virtual Subsystems*, two similarly named *Unit Delay* blocks may be placed in the same Source Code scope. The created global C variables of two blocks called `int_unit_delay` with a maximum identifier length of 15 (exemplary) and mangling length of 4 would be `int_unit_delay` and `int_unit__h4pj`, which reduces readability and impedes debugging as well as verification later on.

4.1.3 Traceability Plugin

One solution used at the institute to document System Requirements and Software High-Level Requirements is *Polarion REQUIREMENTS*.[4] The application is web-based and runs on a web server. On model-side, the *MathWorks* toolbox *Simulink Verification and Validation*[5] provides the Requirement Management Interface (RMI) to annotate model elements and establish a link to any requirement management tool that provides a corresponding plugin. The plugin for connecting *Polarion* and the RMI is provided by the *Polarion Connector for Simulink*.[6] The plugin must be registered in *Simulink* and is thus included in the Modeling Environment, too.

4.1.4 Model Element Library

The model element library contains atomic blocks and small subsystems, which do not generate separate functions, but are inlined in the code of the higher model level. These subsystems have to be small enough to be testable in the model and C-function, in which they are embedded. The usable atomic blocks are mainly defined by compatibility with SLCI, which comes along with the `slcilib` library. This basic block set was modified to additionally comply with the applicable modeling rules.

4.1.5 Simulation Configuration Settings

The configuration settings mainly relevant for simulation are solver, optimization and diagnostic settings. Most of them are given by the guidelines from *MathWorks*, *MAAB* or SLCI. Only a few remain eligible.

[4]https://polarion.plm.automation.siemens.com/products/polarion-requirements [Cited on 6 January 2017].

[5]http://de.mathworks.com/products/simverification/ [Cited on 6 January 2017].

[6]Polarion Connector for Simulink, see http://extensions.polarion.com/extensions/173-polarion-connector-for-simulink [Cited on 6 January 2017].

Table 2 Model configuration settings

Name	Description
Top-level Model (Coding)	Configuration for the top-level model generating an interface of type void-void (see also Sect. 4.2.1)
Reusable Model (Coding)	A reusable model reference can be used for multiple instances. However, only signals with custom storage class “Simulink Global” are allowed (see also Sect. 4.2.1)
Singleton Model (Coding)	A model reference that can only be embedded once. This model can include “Exported Global” signals (see also Sect. 4.2.1)
Test Harness (Simulation)	Since the test harness encapsulates the whole model-in-the-loop simulation with e.g., the flight dynamic model or actuator models, its configuration has to differ slightly to be compatible

For the presented controller, the solver settings restrict execution to “single rate” and with a discrete fixed-step solver. For diagnostics, conservative settings are chosen. Optimization settings mainly affect the generated code and are thus discussed in the next section.

Although referred to as a single configuration set in here, there are normally multiple model configurations. Currently, four slightly different model configuration settings are used in the presented Modeling Environment as listed in Table 2.

Model configuration settings are set in models as “Configuration References”. This increases maintainability and consistency over all software modules.

4.1.6 Common Library Module

Beside the model element library, a so-called *Common Library* is provided to the developer. This library contains subsystems that are too large to be inlined and are thus reusable model references. These libraries are considered a separate software module, and follow an own software verification process.

Content of the library are subsystems adding robustness to critical function calls, like protected divisions, square roots or integrators protected against windup.

4.2 Environment for Code Generation

The generated Source Code shall comply with ANSI ISO/IEC 9899:1999 [13] and satisfy the rules and directives for autogenerated code (Appendix E) of MISRA C:2012 [33].

As already mentioned, EC is used for code generation. The coder provides hundreds of settings and customization possibilities. The following sections demonstrate the minimum of customization that is required from the viewpoint of a DO-331 software development process. In general, it is advisable to avoid extensive customization, since any deviation from the default configuration may have an impact on tool compatibility and robustness [3].

4.2.1 Coder Configuration Settings

Coder configuration settings are manifold and they are defined as modeling rules, since the border between simulation and coder settings is blurry. Many simulation settings are directly used by the coder (for example the single- or multi-rate or most of the optimization settings).

Code Interface The code interface differs depending on the model. For the top-level model, whose functions are externally called by the surrounding C framework, a simple non-reusable interface of type void-void is generated. This creates structures for input and output, as well as void-void `initialize` and `step` functions. For nested models, either a reusable or a non-reusable interface can be chosen, depending on whether multiple or only a single instance shall be allowed. The code of non-reusable models may be easier to test separately, since signals can be exposed as exported global variables. All functions of nested models pass parameters as individual arguments to avoid the overhead of copying the input variables into a single structure.

Code Packaging Code packaging is set for modularity. However, modular code generation cannot be achieved solely by coder settings. The right usage of built-in custom storage classes is another important point.

Code Style For the code style, a nominal parenthesis level is used as well as nominal casting. Note that "Standard compliant" casting is not yet compatible with SLCI. Comments must be adjusted to provide sufficient traceability information in the code. The identifier control was set to standard values.

Support of Functionality Code is generated for a single rate system. Since a floating point unit is available on the target computer, floating point arithmetic is activated. Further features like complex numbers, infinite numbers or absolute time are disabled, since they are neither required nor would they solely be checkable with SLCI due to the large amount of additional code.

Code Optimization To allow verification with SLCI, advanced code optimizations are disabled by setting `AdvancedOptControl` to `-SLCI`. The optimization group contains the powerful feature "Signal Storage Reuse". Although SLCI is compatible with all of the nested settings, their usage must be analyzed carefully. "Signal Storage Reuse" on the one hand can drastically improve the performance of the generated code and reduce the memory requirements, but on the other hand it can significantly reduce readability and robustness concerning change. For example, a locally reused variable may be named according to its first occurrence. In other contexts, in which it is reused, this name may be misleading and may not fit to the actual meaning of the stored value. Furthermore, if the first occurrence changes its variable name, changes all over the generated code will appear. The actual fix around the first occurrence is hard to find and verify in the code.

4.2.2 Modeling Rules Concerning Coding

The structure of the code is not only influenced by the configuration settings, but also by block or signal line settings.

Input and Output Data types of Atomic Blocks Atomic blocks often allow an implicit type conversion. Implicit type conversions affect readability and should be avoided. The data type should be preserved. The block "Sum of elements" for example allows setting an accumulator data type as well as an output data type. Including the input data type, a conversion between three data types is necessary if set differently. Although supported by EC, this is a SLCI incompatibility in most cases.

Integer Saturation on Overflow Blocks for integer calculations typically provide the option to generate code that saturates on integer overflow. This produces significant extra code and is not verifiable by SLCI. Integer saturation on overflow should thus be deactivated in the block and - if required - modeled separately. Exception of the rule is the Abs block, where hisl_0001 of the High Integrity Guidelines explicitly advises to set the option (due to the differing positive and negative ranges of signed integers). It was observed, that developers are normally not aware of this exception, since they take preset library blocks. The result was dead code for every Abs block in early verification, since they added their own protections.

Subsystem Settings Simulink knows various kinds of subsystems, basically virtual and non-virtual (atomic) subsystems. Virtual subsystems can be considered as visual help. The coder eliminates them. By setting a subsystem to "atomic", the coder is forced to keep the generated code of the subsystem together. Further options allow to specify, how this is achieved, either by a function (`reusable` or `non-reusable`) or by a packaged block of code (`inlined`). Up to release 2016a, SLCI only supported the `inlined` option. Structuring code by functions, which is necessary for testing, was only possible by model references. With Release 2016b, `non-reusable` functions are verifiable, too.

Simulink Data Objects By explicitly specifying the storage class of Simulink Data Objects[7] (Simulink.Bus, Simulink.Signal and Simulink.Parameter) and, for example, their header or source file name, the behavior of the code generator can be influenced.

[7]http://de.mathworks.com/help/simulink/ug/working-with-data-objects.html [Cited on 6 January 2017].

Simulink.Parameter objects should be used for non-scalar or structured parameters to prevent that the coder generates a common data pool [9]. Simulink.Signals for example may be used to expose signals for global access (for testing) or to define reused signals and thus prevent data copies.

4.2.3 Customization of the Coding Process and Advanced Target-Specific Configuration

The coding process, controlled by Target Language Compiler (TLC) files, has not been modified for the presented controller. Slightly customized is only the Code Generation Template (CGT), which allows formatting the high-level organization of the code. The general arrangement of the code remained, only the comment fields were updated to include further necessary information and exclude the timestamp (to ease comparison of revisions with Diff tools and identify real changes). Additionally, a data alignment specification is registered to include alignment information for the compiler.

4.2.4 Replacement of Shared Utility Functions

When generating C-code from a Simulink model, some atomic blocks expand to complex functions. These functions are placed in a shared location, so they are only generated once for all models. They have canonical function and file names differing for every combination of settings in the block mask.

For these shared C functions, the following aspects have to be considered:

1. In the presented case, the Design Model replaces Software Low-Level and Software Architecture. Therefore, the Design Model must be sufficiently descriptive and granular. If an atomic block produces complex code, this argumentation does not hold any more.
2. Requirements cannot be linked into shared utility C-code, since these functions are regenerated every build.
3. SLCI just verifies the call, but not the correctness of the function itself.
4. Shared utility function complicate configuration management if their generation is not controlled. Although the functions have canonical names, the number of C functions, that can be created, is immense, even with a reduced block set (thousands). Limiting the shared utility functions to a dedicated subset of functions is task of the modeling rules, which have to restrict the allowed atomic block settings.

As a consequence of points 1 to 3, it is desirable to replace the generated shared functions by C functions developed and verified along a conventional DO-178C software development process. One way is using so-called *Code Replacement Libraries*. They allow replacing specified function calls in the generated C-code by custom calls. This avoids the generation of the shared utilities.

Not yet solved with this solution is point 4. If the modeling rules are not restrictive enough, shared code beyond the replaced function may be generated. A repeated review for unsupported, additionally generated functions is necessary after building the code.

With Release 2016b, the new EC setting `ExistingSharedCode` was introduced, which allows the specification of an already existing shared code directory. EC then scans the specified directory for the functions it requires during code generation and copies the files to the new shared folder. Identified are the files by their canonical filename. Further, setting the option `UseOnlyExistingSharedCode` allows to automatically abort the coding process, if not all required files are found. Thus, the developer gets direct feedback, whether the model generates unsupported shared functions or not.

For the presented controller, only four auto-generated functions are required. Reason is, that SLCI limits the options e.g., for Lookup Tables and their dimensions. Additionally, the integrated integer overflow protection functionality is disabled for all blocks except the Abs Block (see Sect. 4.2.2).

- `rt_roundd`: Although the ISO C standard [13] defines the `round` function that rounds halfway cases away from zero regardless the floating point rounding direction, EC generates `rt_roundd`, which only bases on `ceil` and `floor` functions. Additionally, it prevents rounding if the units in the last place (ULPs) are equal or greater than 1.0. The `rt_roundd` function has probably been introduced to also ensure calculation equality between simulation and code for non-standard compliant C libraries.
- `rt_modd`: This is a protected version and safe implementation of the double-precision floating point modulo operation generated by the Mathematical Function block. The function is implemented to preserve simulation and code equality for non-standard compliant C math libraries. If the typically used `fmod` function shall be called directly, the `Rem` block setting has to be used. However, even the C standard does not define the behavior for a divisor equal to zero for this function.
- `look1_binlca` and `look2_binlca`: Functions originating from Lookup Table blocks with double-precision input, output and accumulator data type, binary search and linear interpolation. Extrapolation method is "cut" and out-of-range protection is not removed.

4.2.5 Replacement of Compiler-Supplied Libraries

Beside the shared functions, the controller code generated with EC addresses additional standard C functions like `sin` or `memcpy`. Although these basic functions are normally supplied with the compiler, they have to be verified separately and must satisfy DO-178C objectives [2].

Depending on the used libraries, a *Code Replacement Library* must be registered in Simulink before generating code, to replace standard C function calls (e.g., a `sin(...)` to `cert_sin(...)`).

5 Summary

Using *Simulink* and *Stateflow* for implementing a Design Model is not an out-of-the box solution, but requires a planned setup and preparation of the tool. In this paper, aspects of a consistent Modeling Environment were presented in the light of the used hardware, application design considerations, and the chosen model-based development process.

The paper structured the Modeling Environment in two parts, one containing the relevant components concerning the role of the Design Model as Software Design and Software Architecture, and a second part with focus on code generation. For the first part, an overview of the applicable rules was given and model libraries as well as a important configuration settings have been highlighted. The code generation part pointed out the major settings relevant for EC and strategies to deal with code libraries.

The details in this paper should not be seen as the ultimate solution for every company and workflow due to the varying context, but the key aspects can be considered in every safety-critical project.

The close connection between Design Model and Source Code in a code-generation based process emphasizes the importance of presciently planned rules, restrictions, and settings for *MATLAB*, *Simulink*, and *Stateflow*, since any setting in the Design Model may directly impact the Source Code. Making the Modeling Environment consistent may require an iterative process and experience.

The discussion also showed the central role of SLCI. This tool should not only be seen as a sole verification tool since it also plays an important part in defining a safe and robust subset of *MATLAB*, *Simulink*, and *Stateflow* features for safety-critical applications.

The outlined Modeling Environment is a snapshot of the effort taken at the *Institute of System Dynamics*. Further investigation is planned on modular code generation, and optimal settings for improved compatibility with 3rd party tools used along the workflow, like WCET analyzers or special compilers, all coming along with additional restrictions.

References

1. Braun B, Philip S, Peter L, Dambeck J, Holzapfel F (2013) Multi-purpose flying sensor testbed: AIRTEC 2013 aerospace sensors/aerospace testing. Frankfurt a. M., 6 November 2013
2. Certification Authorities Software Team (2004) Position paper CAST-21 - compiler-supplied libraries, January 2004
3. Dillaber E, Kendrick L, Jin W, Reddy V (eds) (2010) Pragmatic strategies for adopting model-based design for embedded applications. SAE Int
4. Erkinnen T, Potter B (2009) Model-based design for DO-178B with qualified tools: AIAA modeling and simulation technologies conference and exhibit. American Institute of Aeronautics and Astronautics Inc, Hyatt Regency McCormick Place, Chicago Illinois

5. Esterel Technologies SA (2015) Efficient development of safe avionics software with DO-178C objectives using SCADE suite: methodology handbook, June 2015
6. Estrada RG, Sasaki G, Dillaber E (2013) Best practices for developing DO-178 compliant software using model-based design. AIAA Infotech@Aerospace. Boston. https://doi.org/10.2514/6.2013-4566
7. European Aviation Safety Agency EASA: certification specifications for normal, utility, aerobatic and commuter category aeroplanes: CS-23 Amendment 3
8. Federal Aviation Administration FAA (2011) System safety analysis and assessment for part 23 airplanes (AC 23.1309-1E) 17 November 2011
9. Hochstrasser M, Hornauer M, Holzapfel F (2016) Formal verification of flight control applications along a model-based development process: a case study. In: DGLR Workshop - Software Safety. München, 05 October 2016. http://www.dglr.de/fileadmin/inhalte/dglr/fb/q3/veranstaltungen/L63_Q34_2016_Software_Safety/2016_DGLR_Workshop_TUM_samoconsult.pdf
10. Hornauer M, Holzapfel F (2011) Model based testing for CS-23 avionic and UAV applications: DGLR workshop 2011. In: DGLR Workshop - Verifikation in der modellbasierten Software-Entwicklung, München
11. Hornauer M, Schuck F, Holzapfel F (2013) Wechselwirkungen zwischen GNC algorithmus und software. In: DGLR Workshop - Durchgängige Entwicklung von GNC Funktionen - vom Algorithmus zur Embedded Software. München
12. IEEE Computer Society (2008) IEEE standard for floating point arithmetic (IEEE 754-2008), August 2008
13. ISO/IEC (1999) Programming languages C - 2nd edn, December 1999
14. Karlsson E, Gabrys A, Schatz SP, Holzapfel F (2016) Dynamic flight path control coupling for energy and maneuvering integrity. In: IEEE control systems society (ed) proceedings of 14th international conference on control, automation, robotics and vision
15. Karlsson E, Schatz SP, Baier T, Dörhöfer C, Gabrys A, Hochstrasser M, Krause C, Lauffs PJ, Mumm NC, Nürnberger K, Peter L, Schneider V, Philip S, Steinert L, Zollitsch AW, Holzapfel F (2016) Automatic flight path control of an experimental DA42 general aviation aircraft. In: IEEE control systems society (ed.) proceedings of 14th international conference on control, automation, robotics and vision
16. Krause C, Holzapfel F (2016) Designing a system automation for a novel UAV demonstrator. In: IEEE control systems society (ed.) proceedings of 14th international conference on control, automation, robotics and vision
17. Potter B (2012) Complying with DO-178C and DO-331 using model-based design
18. RTCA (2000) DO-254 - design assurance guidance for airborne electronic hardware
19. RTCA (2011) DO-178C - software considerations in airborne systems and equipment certification
20. RTCA (2011) DO-331 - model-based development and verification supplement to DO-178C and DO-278A
21. RTCA (2011) DO-333 formal methods supplement to DO-178C and DO-278A
22. Schatz SP, Holzapfel F (2014) Modular trajectory/path following controller using nonlinear error dynamics. In: 2014 IEEE international aerospace electronics and remote sensing technology (ICARES), pp. 157–163. IEEE. https://doi.org/10.1109/ICARES.2014.7024374
23. Schatz SP, Schneider V, Karlsson E, Holzapfel F, Baier T, Dörhöfer C, Hochstrasser M, Gabrys A, Krause C, Lauffs PJ, Mumm NC, Nürnberger K, Peter L, Spiegel P, Steinert L, Zollitsch AW (2016) Flightplan flight tests of an experimental DA42 generation aviation aircraft. In: IEEE control systems society (ed.) proceedings of 14th international conference on control, automation, robotics and vision
24. Scheible J (2012) Automatisierte qualitätsbewertung am beispiel von matlab simulink-modellen in der automobil-domäne. Dissertation, Eberhard Karls Universität Tübingen, Tübingen. https://publikationen.uni-tuebingen.de/xmlui/handle/10900/49708
25. Schneider V, Mumm N, Holzapfel F (2015) Trajectory generation for an integrated mission management system. In: 2014 IEEE international aerospace electronics and remote sensing technology (ICARES). IEEE

26. Schneider V, Piprek P, Schatz SP, Baier T, Dörhöfer C, Hochstrasser M, Gabrys A, Karlsson E, Krause C, Lauffs PJ, Mumm NC, Nürnberger K, Peter L, Spiegel P, Steinert L, Holzapfel F (2016) Online trajectory generation using clothoid segments. In: IEEE control systems society (ed) proceedings of 14th international conference on control, automation, robotics and vision
27. The MathWorks automotive advisory board: MathWorks automotive advisory board control algorithm modeling guidelines using MATLAB, Simulink, and Stateflow: R2016b
28. The MathWorks Inc. Embedded coder user's guide: R2016b
29. The MathWorks Inc. Guidelines and factors to consider for code generation: R2016b
30. The MathWorks Inc. Modeling guidelines for high-integrity systems: R2016b
31. The MathWorks Inc. (2014) Embedded coder R2014b - MISRA AC AGC compliance considerations
32. The MathWorks Inc. (2016) DO qualification R2016b: model-based design workflow for DO-178C
33. The Motor Industry Software Reliability Association (2013) MISRA-C:2012 - Guidelines for the use of C language in critical systems, March 2013